Application Techniques

PowerBuilder®

11.0

DOCUMENT ID: DC37774-01-1100-01

LAST REVISED: May 2007

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

Contents

PART 5 PROGRAM ACCESS TECHNIQUES

PART 8 GENERAL TECHNIQUES

PART 9 DEPLOYMENT TECHNIQUES

About This Book

Audience

You should read this book if you are involved in any phase of a client/server, distributed, or Web application development project that uses PowerBuilder®.

How to use this book

This how-to book guides you through programming techniques used to build and deploy PowerBuilder applications and components. It does this by equipping you with collections of techniques for implementing many common application features and advice for choosing those techniques best suited to your requirements.

PowerBuilder is accompanied by sample applications that illustrate some of the issues, features, and techniques you will read about. Examine the components of these applications in PowerBuilder, read the comments in the code, and experiment with real, working examples of what you are trying to learn.

For where to find the sample applications, see Chapter 1, “Using Sample Applications.”

Related documents

For a description of all the books in the PowerBuilder documentation set, see the preface of *PowerBuilder Getting Started*.

Other sources of information

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

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

- The SyBooks CD contains product manuals and is included with your software. The Eclipse-based SyBooks browser allows you to access the manuals in an easy-to-use, HTML-based format.

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

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

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

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

Conventions

The formatting conventions used in this manual are:

Formatting example	Indicates
Retrieve and Update	When used in descriptive text, this font indicates: • Command, function, and method names • Keywords such as true, false, and null • Datatypes such as integer and char • Database column names such as emp_id and f_name • User-defined objects such as dw_emp or w_main
variable or *file name*	When used in descriptive text and syntax descriptions, oblique font indicates: • Variables, such as *myCounter* • Parts of input text requiring substitution, such as *pblname*.`pbd` • File and path names
File>Save	Menu names and menu items are displayed in plain text. The greater than symbol (>) shows you how to navigate menu selections. For example, File>Save indicates “select Save from the File menu.”
`dw_1.Update()`	Monospace font indicates: • Information that you enter in a dialog box or on a command line • Sample script fragments • Sample output fragments

If you need help

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

PART 1

Sample Applications

This part introduces the sample applications provided with PowerBuilder and explains how you use them to learn programming techniques.

CHAPTER 1 Using Sample Applications

About this chapter

This chapter describes how to use PowerBuilder sample applications.

Contents

About the sample applications

PowerBuilder provides sample applications with source code so you can learn and reuse the techniques used in the samples. There are two kinds of samples:

- Samples that you can download from the Web
- Samples you can install from the CD

Samples on the Web

The latest PowerBuilder sample applications and utilities are on the Sybase CodeXchange Web site in the PowerBuilder Samples and Utilities project at http://powerbuilder.codeXchange.sybase.com/. There is a link to this page on the Windows Start menu at Program Files>Sybase>PowerBuilder 11.0>Sybase CodeXchange. If you have not logged in to MySybase, you must log in on the Sybase Universal Login page to access CodeXchange. If you do not have a MySybase account, you can sign up using the link on this page. MySybase is a free service that provides a personalized portal into the Sybase Web site.

These samples are contributed by Sybase employees and users and are updated frequently. They include standalone applications that illustrate specific features of PowerBuilder, including new features such as using Web services, creating EJB clients, and writing visual and nonvisual extensions using the PowerBuilder Native Interface. Most samples include a readme document that explains which features a sample demonstrates and how to download and use it.

Samples on the CD

There are two samples on the CD that you can install when you install PowerBuilder.

Code Examples

The PowerBuilder Code Examples application contains many examples you can use to learn a range of coding techniques. The examples are specifically designed to show you how to use different coding techniques with both old and new PowerBuilder features.

Web DataWindow

To develop an application that uses Web DataWindow® technology, you can use the generic server component that is preinstalled in EAServer and provided as a DLL for use with the Microsoft Component Object Model (COM). You can also create your own HTML generator server component, using the sample Web DataWindow PBL as a model. For more information about the Web DataWindow, see the *DataWindow Programmer's Guide*.

Installing the sample applications

To install the samples from the CD, select the Custom setup type in the PowerBuilder installation program and select Code Examples from the list of components. To install the Code Examples application, select Example Application. To install the Web DataWindow PBL, select Web DataWindow.

Both samples are installed in the *Code Examples* subdirectory in your *PowerBuilder 11.0* directory. The Code Examples application uses a sample SQL Anywhere database called the EAS Demo DB, which is installed in the *Sybase\Shared\PowerBuilder* directory. If you cannot find the *Code Examples* directory or the *EASDEMO110.DB* file, the sample applications and the database may not have been installed.

Opening the sample applications

To open a sample application, you need to create a workspace for the application and then add the target for that application to your workspace.

If there is no target file for an application, you can still add it to your workspace. Select Existing Application from the Target tab of the New dialog box. In the wizard that opens, navigate to the *PowerBuilder 11.0.0\Code Examples* folder, expand the folder and PBL that contains the application you want to examine, and select the application.

The next section contains a procedure that steps you through opening and running the Code Examples application.

Using the Code Examples application

You run the Code Examples application from the development environment.

❖ **To run the Code Examples application:**

1 Select File>New from the menu bar, select Workspace from the Workspace tab, and click OK.

2 Navigate to the *PowerBuilder 11.0\Code Examples\Example App* folder, type a name for the workspace, and click Save.

3 Select Add Target from the pop-up menu for the workspace you just created, navigate to the *PowerBuilder 11.0\Code Examples\Example App* folder, select the *PB Examples* target file, and click Open.

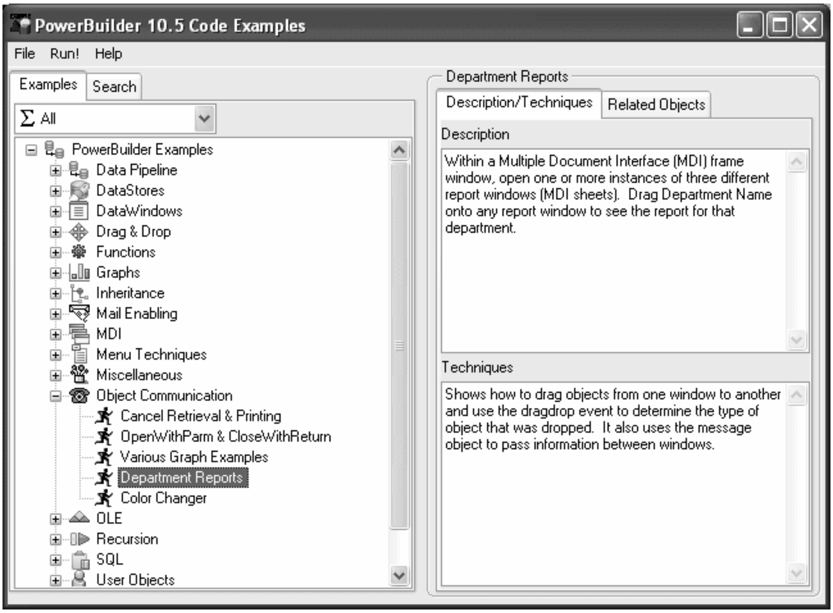
If you expand the target, you will see that the PBL that contains the application and all its supporting PBLs have been added to the workspace.

4 Click the Run button on the PowerBar.

Browsing the examples

When the Code Examples application opens, the left pane contains an expandable tree view listing the categories of examples available. Some examples appear in more than one category. For example, the Business Class example is listed under Inheritance and User Objects. If you are looking for examples showing how to work with a specific feature, such as DataStores or DataWindows, expand that category and look at the example names.

When you select an example in the left pane, a description of the example and the techniques it demonstrates displays on the right:



Within a Multiple Document Interface (MDI) frame window, open one or more instances of three different report windows (MDI sheets). Drag Department Name onto any report window to see the report for that department.

Shows how to drag objects from one window to another and use the dragdrop event to determine the type of object that was dropped. It also uses the message object to pass information between windows.

Finding examples

If you are looking for ways to work with a specific PowerBuilder object class or feature, you can use the categories in the Examples pane and the descriptions to locate examples. If you are looking for examples using a specific event, function, or user-defined object, use the Search pane.

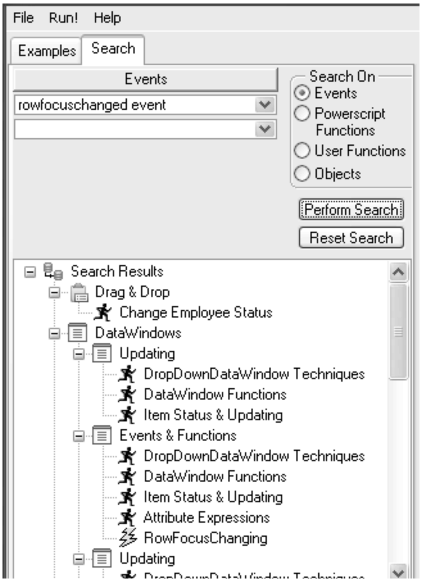
❖ **To search for a function, event, or object:**

1 Click the Search tab in the Code Examples main window.

2 Select a radio button in the Search On group box.

3 Select the item you want in the drop-down list and click Perform Search.

The names of all the examples that use the function, event, or object you searched for display:



Events

rowfocuschanged event

Search On

● Events

○ Powerscript Functions

○ User Functions

○ Objects

Running and examining examples

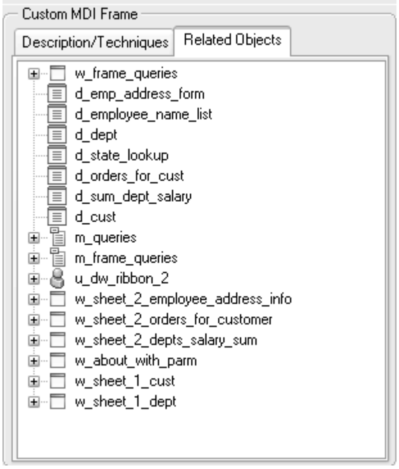
Once you have located an example that performs some processing you want to learn about, you can run it to examine how it works and look at the code (and copy it if you want to).

Running an example

To run the highlighted example, double-click it or click Run!. You can get Help on how to use the example and what it does by clicking the Help button on the example's main window.

Examining the code

To see all the objects used in an example, click the Related Objects tab on the right pane and click the plus signs to expand items:



w_frame_queries
d_emp_address_form
d_employee_name_list
d_dept
d_state_lookup
d_orders_for_cust
d_sum_dept_salary
d_cust
m_queries
m_frame_queries
u_dw_ribbon_2
w_sheet_2_employee_address_info
w_sheet_2_orders_for_customer
w_sheet_2_depts_salary_sum
w_about_with_parm
w_sheet_1_cust
w_sheet_1_dept

Double-click the icon for a script or function to examine it.

Using examples in the development environment

Running the Code Examples application and looking at the code for an example gives you a lot of information, but if you open objects in the examples in the development environment, you can examine them in more depth.

For example, you can open any object in a painter, examine the inheritance hierarchy in the Browser, and step through an example in the Debugger. You can even copy objects to your own application in the Library painter or copy code fragments to the Script view.

The libraries in the Code Examples application are organized by object type. For example, *pbexamd1.pbl* and *pbexamd2.pbl* contain DataWindow objects. This makes it easy to find the objects that are referenced as examples later in this book. If you expand the sample libraries in the List view in the Library painter, the comments tell you what each object is used for.

PART 2

Language Techniques

This part presents a collection of programming techniques you can use to take advantage of PowerBuilder object-oriented features and PowerScript® language elements, including the ClassDefinition object.

CHAPTER 2

Selected Object-Oriented Programming Topics

About this chapter

This chapter describes how to implement selected object-oriented programming techniques in PowerBuilder.

Contents

Terminology review

Classes, properties, and methods

In object-oriented programming, you create reusable **classes** to perform application processing. These classes include **properties** and **methods** that define the class's behavior. To perform application processing, you create **instances** of these classes. PowerBuilder implements these concepts as follows:

- **Classes** PowerBuilder objects (such as windows, menus, window controls, and user objects)
- **Properties** Object variables and instance variables
- **Methods** Events and functions

The remaining discussions in this chapter use this PowerBuilder terminology.

Fundamental principles

Object-oriented programming tools support three fundamental principles: inheritance, encapsulation, and polymorphism.

Inheritance Objects can be derived from existing objects, with access to their visual component, data, and code. Inheritance saves coding time, maximizes code reuse, and enhances consistency. A descendent object is also called a **subclass**.

Encapsulation An object contains its own data and code, allowing outside access as appropriate. This principle is also called *information hiding*. PowerBuilder enables and supports encapsulation by giving you tools that can enforce it, such as access and scope. However, PowerBuilder itself does not require or automatically enforce encapsulation.

Polymorphism Functions with the same name behave differently, depending on the referenced object. Polymorphism enables you to provide a consistent interface throughout the application and within all objects.

Visual objects

Many current applications make heavy use of object-oriented features for visual objects such as windows, menus, and visual user objects. This allows an application to present a consistent, unified look and feel.

Nonvisual objects

To fully benefit from PowerBuilder's object-oriented capabilities, consider implementing class user objects, also known as nonvisual user objects:

Standard class user objects Inherit their definitions from built-in PowerBuilder system objects, such as Transaction, Message, or Error. The nvo_transaction Transaction object in the Code Examples sample application is an example of a subclassed standard class user object. Creating customized standard class user objects allows you to provide powerful extensions to built-in PowerBuilder system objects.

Custom class user objects Inherit their definitions from the PowerBuilder NonVisualObject class. Custom class user objects encapsulate data and code. This type of class user object allows you to define an object class from scratch. The u_business_object user object in the Code Examples sample application is an example of a custom class user object. To make the most of PowerBuilder's object-oriented capabilities, you must use custom class user objects. Typical uses include:

- **Global variable container** The custom class user object contains variables and functions for use across your application. You encapsulate these variables as appropriate for your application, allowing access directly or through object functions.
- **Service object** The custom class user object contains functions and variables that are useful either in a specific context (such as a DataWindow) or globally (such as a collection of string-handling functions).

- **Business rules** The custom class user object contains functions and variables that implement business rules. You can either create one object for all business rules or create multiple objects for related groups of business rules.
- **Distributed computing** The custom class user object contains functions that run on a server or cluster of servers.

  For more information, see Part 6, “Distributed Application Techniques.”

PowerBuilder techniques

PowerBuilder provides full support for inheritance, encapsulation, and polymorphism in both visual and nonvisual objects.

Creating reusable objects
In most cases, the person developing reusable objects is not the same person using the objects in applications. This discussion describes defining and creating reusable objects. It does not address usage.

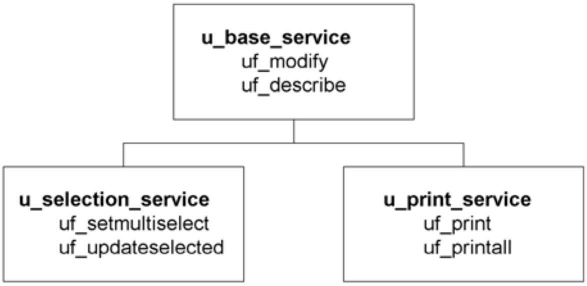
Implementing inheritance

PowerBuilder makes it easy to create descendent objects. You implement inheritance in PowerBuilder by using a painter to inherit from a specified ancestor object.

For examples of inheritance in visual objects, see the w_employee window and u_employee_object in the Code Examples sample application.

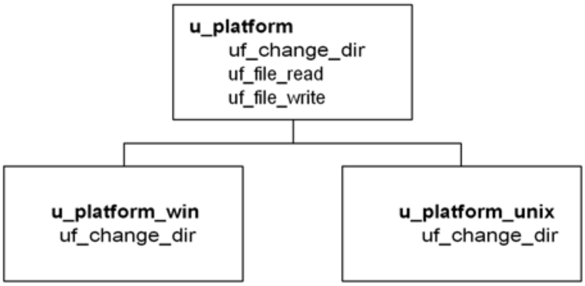
Example of ancestor service object One example of using inheritance in custom class user objects is creating an ancestor service object that performs basic services and several descendent service objects. These descendent objects perform specialized services, as well as having access to the ancestor’s services:

Figure 2-1: Ancestor service object



Example of virtual function in ancestor object Another example of using inheritance in custom class user objects is creating an ancestor object containing functions for all platforms and then creating descendent objects that perform platform-specific functions. In this case, the ancestor object contains a **virtual function** (uf_change_dir in this example) so that developers can create descendent objects using the ancestor's datatype.

Figure 2-2: Virtual function in ancestor object



For more on virtual functions, see "Other techniques" on page 17.

Implementing encapsulation

Encapsulation allows you to insulate your object's data, restricting access by declaring instance variables as private or protected. You then write object functions to provide selective access to the instance variables.

One approach One approach to encapsulating processing and data is as follows:

- Define instance variables as public, private, or protected, depending on the desired degree of outside access. To ensure complete encapsulation, define instance variables as either private or protected.
- Define object functions to perform processing and provide access to the object's data.

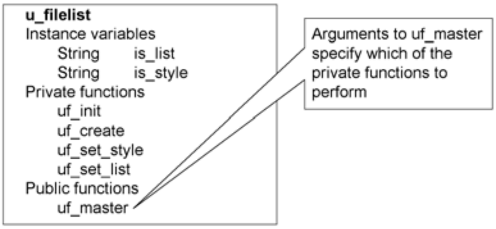
Table 2-1: Defining object functions

To do this	Provide this function	Example
Perform processing	uf_do_*operation*	uf_do_retrieve (which retrieves rows from the database)
Modify instance variables	uf_set_*variablename*	uf_set_style (which modifies the is_style string variable)
Read instance variables	uf_get_*variablename*	uf_get_style (which returns the is_style string variable)
(Optional) Read boolean instance variables	uf_is_*variablename*	uf_is_protected (which returns the ib_protected boolean variable)

Another approach Another approach to encapsulating processing and data is to provide a single entry point, in which the developer specifies the action to be performed:

- Define *instance variables* as private or protected, depending on the desired degree of outside access
- Define private or protected *object functions* to perform processing
- Define a single *public function* whose arguments indicate the type of processing to perform

Figure 2-3: Defining a public function for encapsulation



For an example, see the uo_sales_order user object in the Code Examples sample application.

Distributed components

When you generate an application server component, public functions are available in the interface of the generated component and you can choose to make public instance variables available. Private and protected functions and variables are never exposed in the interface of the generated component.

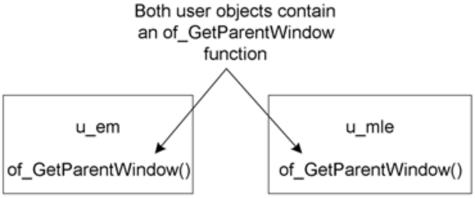
For more information, see Part 6, "Distributed Application Techniques."

Implementing polymorphism

Polymorphism refers to a programming language's ability to process objects differently depending on their datatype or class. Polymorphism means that functions with the same name behave differently depending on the referenced object. Although there is some discussion over an exact definition for polymorphism, many people find it helpful to think of it as follows:

Operational polymorphism Separate, unrelated objects define functions with the same name. Each function performs the appropriate processing for its object type:

Figure 2-4: Operational polymorphism

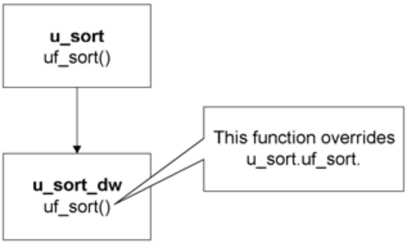


For an example, see the u_external_functions user object and its descendants in the Code Examples sample application.

Inclusional polymorphism Various objects in an inheritance chain define functions with the same name.

With inclusional polymorphism PowerBuilder determines which version of a function to execute, based on where the current object fits in the inheritance hierarchy. When the object is a descendant, PowerBuilder executes the descendent version of the function, overriding the ancestor version:

Figure 2-5: Inclusional polymorphism



For an example, see the u_employee_object user object in the Code Examples sample application.

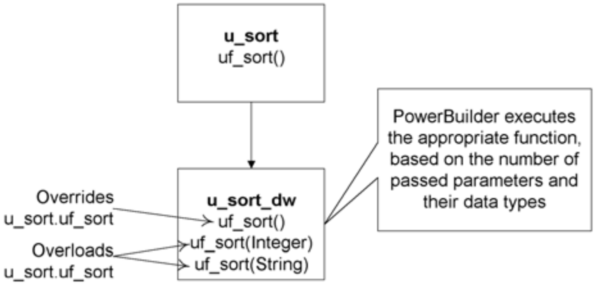
Other techniques

PowerBuilder allows you to implement a wide variety of object-oriented techniques. This section discusses selected techniques and relates them to PowerBuilder.

Using function overloading

In function overloading, the descendent function (or an identically named function in the same object) has different arguments or argument datatypes. PowerBuilder determines which version of a function to execute, based on the arguments and argument datatypes specified in the function call:

Figure 2-6: Function overloading

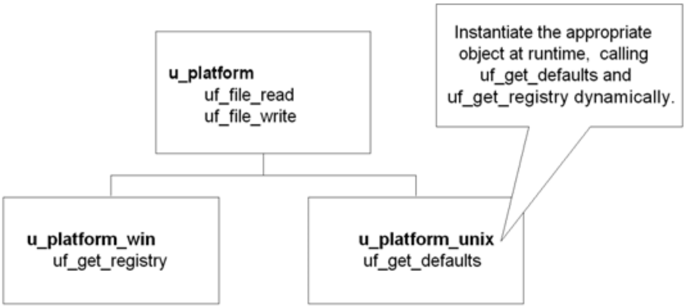


Global functions
Global functions cannot be overloaded.

Dynamic versus static lookup

Dynamic lookup In certain situations, such as when insulating your application from cross-platform dependencies, you create separate descendent objects, each intended for a particular situation. Your application calls the platform-dependent functions dynamically:

Figure 2-7: Dynamic lookup


Instantiate the appropriate object at runtime, calling uf_get_defaults and uf_get_registry dynamically.
u_platform
uf_file_read
uf_file_write
u_platform_win
uf_get_registry
u_platform_unix
uf_get_defaults

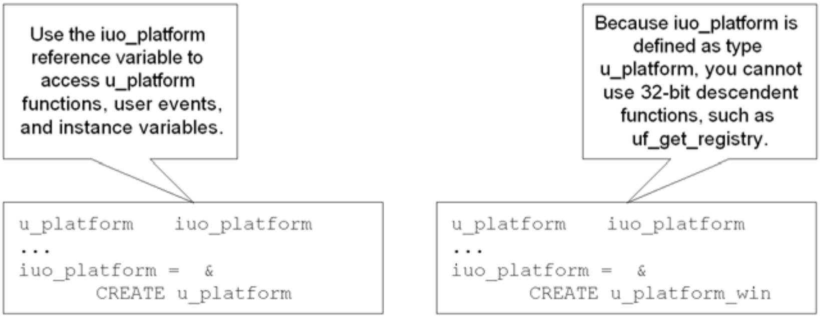
Instantiate the appropriate object at runtime, as shown in the following code example:

```
// This code works with both dynamic and
// static lookup.
// Assume these instance variables
u_platform iuo_platform
Environment ienv_env
...
GetEnvironment(ienv_env)
choose case ienv_env.ostype
   case windows!
      iuo_platform = CREATE u_platform_win
   case windowsnt!
      iuo_platform = CREATE u_platform_win
   case else
      iuo_platform = CREATE u_platform_unix
end choose
```

Although dynamic lookup provides flexibility, it also slows performance.

Static lookup To ensure fast performance, static lookup is a better option. However, PowerBuilder enables object access using the reference variable's datatype (not the datatype specified in a CREATE statement).

Figure 2-8: Static lookup



Use the iuo_platform reference variable to access u_platform functions, user events, and instance variables.

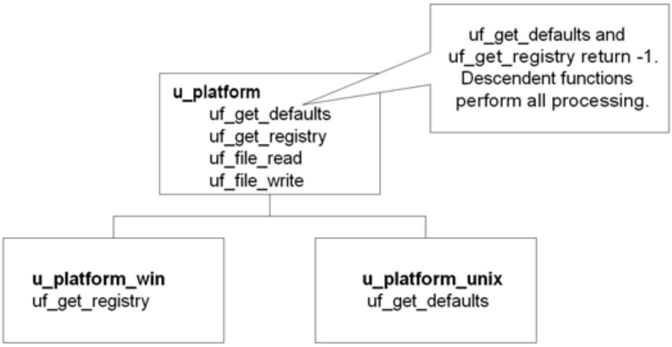
```
u_platform   iuo_platform
...
iuo_platform =  &
      CREATE u_platform
```

Because iuo_platform is defined as type u_platform, you cannot use 32-bit descendent functions, such as uf_get_registry.

```
u_platform   iuo_platform
...
iuo_platform =  &
      CREATE u_platform_win
```

When using static lookup, you must define default implementations for functions in the ancestor. These ancestor functions return an error value (for example, -1) and are overridden in at least one of the descendent objects.

Figure 2-9: Ancestor functions overridden in descendent functions



By defining default implementations for functions in the ancestor object, you get platform independence as well as the performance benefit of static lookup.

Using delegation

Delegation occurs when objects offload processing to other objects.

Aggregate relationship In an aggregate relationship (sometimes called a *whole-part relationship*), an object (called an owner object) associates itself with a service object designed specifically for that object type.

For example, you might create a service object that handles extended row selection in DataWindow objects. In this case, your DataWindow objects contain code in the Clicked event to call the row selection object.

❖ **To use objects in an aggregate relationship:**

1 Create a service object (u_sort_dw in this example).

2 Create an instance variable (also called a reference variable) in the owner (a DataWindow control in this example):

```
u_sort_dw iuo_sort
```

3 Add code in the owner object to create the service object:

```
iuo_sort = CREATE u_sort_dw
```

4 Add code to the owner’s system events or user events to call service object events or functions. This example contains the code you might place in a ue_sort user event in the DataWindow control:

```
IF IsValid(iuo_sort) THEN
   Return iuo_sort.uf_sort()
ELSE
   Return -1
END IF
```

5 Add code to call the owner object's user events. For example, you might create a CommandButton or Edit>Sort menu item that calls the ue_sort user event on the DataWindow control.

6 Add code to the owner object's Destructor event to destroy the service object:

```
IF IsValid(iuo_sort) THEN
   DESTROY iuo_sort
END IF
```

Associative relationship In an associative relationship, an object associates itself with a service to perform a specific type of processing.

For example, you might create a string-handling service that can be enabled by any of your application's objects.

The steps you use to implement objects in an associative relationship are the same as for aggregate relationships.

Using user objects as structures

When you enable a user object's AutoInstantiate property, PowerBuilder instantiates the user object along with the object, event, or function in which it is declared. You can also declare instance variables for a user object. By combining these two capabilities, you create user objects that function as structures. The advantages of creating this type of user object are that you can:

- Create descendent objects and extend them.
- Create functions to access the structure all at once.
- Use access modifiers to limit access to certain instance variables.

❖ **To create a user object to be used as a structure:**

1 Create the user object, defining instance variables only.

2 Enable the user object's AutoInstantiate property by checking AutoInstantiate on the General property page.

3 Declare the user object as a variable in objects, functions, or events as appropriate.

PowerBuilder creates the user object when the object, event, or function is created and destroys it when the object is destroyed or the event or function ends.

Subclassing DataStores

Many applications use a DataWindow visual user object instead of the standard DataWindow window control. This allows you to standardize error checking and other, application-specific DataWindow behavior. The u_dwstandard DataWindow visual user object found in the tutorial library *TUTOR_PB.PBL* provides an example of such an object.

Since DataStores function as nonvisual DataWindow controls, many of the same application and consistency requirements apply to DataStores as to DataWindow controls. Consider creating a DataStore standard class user object to implement error checking and application-specific behavior for DataStores.

CHAPTER 3 Selected PowerScript Topics

About this chapter

This chapter describes how to use elements of the PowerScript language in an application.

Contents

Dot notation

Dot notation lets you qualify the item you are referring to—instance variable, property, event, or function—with the object that owns it.

Dot notation is for objects. You do not use dot notation for global variables and functions, because they are independent of any object. You do not use dot notation for shared variables either, because they belong to an object class, not an object instance.

Qualifying a reference

Dot notation names an object variable as a qualifier to the item you want to access:

objectvariable.item

The object variable name is a qualifier that identifies the owner of the property or other item.

Adding a parent qualifier To fully identify an object, you can use additional dot qualifiers to name the parent of an object, and its parent, and so on:

parent.objectvariable.item

A **parent** object contains the child object. It is not an ancestor-descendent relationship. For example, a window is a control's parent. A Tab control is the parent of the tab pages it contains. A Menu object is the parent of the Menu objects that are the items on that menu.

Many parent levels You can use parent qualifiers up to the level of the application. You typically need qualifiers only up to the window level.

For example, if you want to call the Retrieve function for a DataWindow control on a tab page, you might qualify the name like this:

```
w_choice.tab_alpha.tabpage_a.dw_names.Retrieve()
```

Menu objects often need several qualifiers. Suppose a window w_main has a menu object m_mymenu, and m_mymenu has a File menu with an Open item. You can trigger the Open item's Selected event like this:

```
w_main.m_mymenu.m_file.m_open.EVENT Selected()
```

As you can see, qualifying a name gets complex, particularly for menus and tab pages in a Tab control.

How many qualifiers? You need to specify as many qualifiers as are required to identify the object, function, event, or property.

A parent object knows about the objects it contains. In a window script, you do not need to qualify the names of the window's controls. In scripts for the controls, you can also refer to other controls in the window without a qualifier.

For example, if the window w_main contains a DataWindow control dw_data and a CommandButton cb_close, a script for the CommandButton can refer to the DataWindow control without a qualifier:

```
dw_data.AcceptText()
dw_data.Update()
```

If a script in another window or a user object refers to the DataWindow control, the DataWindow control needs to be qualified with the window name:

```
w_main.dw_data.AcceptText()
```

Referencing objects

There are three ways to qualify an element of an object in the object's own scripts:

- Unqualified:

  ```
  li_index = SelectItem(5)
  ```

  An unqualified name is unclear and might result in ambiguities if there are local or global variables and functions with the same name.

- Qualified with the object's name:

  ```
  li_index = lb_choices.SelectItem(5)
  ```

  Using the object name in the object's own script is unnecessarily specific.

- Qualified with a generic reference to the object:

  ```
  li_index = This.SelectItem(5)
  ```

  The pronoun This shows that the item belongs to the owning object.

This pronoun In a script for the object, you can use the pronoun This as a generic reference to the owning object:

This.*property*

This.*function*

Although the property or function could stand alone in a script without a qualifier, someone looking at the script might not recognize the property or function as belonging to an object. A script that uses This is still valid if you rename the object. The script can be reused with less editing.

You can also use This by itself as a reference to the current object. For example, suppose you want to pass a DataWindow control to a function in another user object:

```
uo_data.uf_retrieve(This)
```

This example in a script for a DataWindow control sets an instance variable of type DataWindow so that other functions know the most recently used DataWindow control:

```
idw_currentdw = This
```

Parent pronoun The pronoun Parent refers to the parent of an object. When you use Parent and you rename the parent object or reuse the script in other contexts, it is still valid.

For example, in a DataWindow control script, suppose you want to call the Resize function for the window. The DataWindow control also has a Resize function, so you must qualify it:

```
// Two ways to call the window function
w_main.Resize(400, 400)
Parent.Resize(400, 400)

// Three ways to call the control's function
Resize(400, 400)
dw_data.Resize(400, 400)
This.Resize(400, 400)
```

GetParent function The Parent pronoun works only within dot notation. If you want to get a reference to the parent of an object, use the GetParent function. You might want to get a reference to the parent of an object other than the one that owns the script, or you might want to save the reference in a variable:

```
window w_save
w_save = dw_data.GetParent()
```

For example, in another CommandButton’s Clicked event script, suppose you wanted to pass a reference to the control’s parent window to a function defined in a user object. Use GetParent in the function call:

```
uo_winmgmt.uf_resize(This.GetParent(), 400, 600)
```

ParentWindow property and function Other tools for getting the parent of an object include:

- **ParentWindow property** – used in a menu script to refer to the window that is the parent of the menu
- **ParentWindow function** – used in any script to get a reference to the window that is the parent of a particular window

For more about these pronouns and functions, see the *PowerScript Reference*.

Objects in a container object

Dot notation also allows you to reach inside an object to the objects it contains. To refer to an object inside a container, use the Object property in the dot notation. The structure of the object in the container determines how many levels are accessible:

control.Object.*objectname*.*property*

control.Object.*objectname*.Object.*qualifier*.*qualifier*.*property*

Objects that you can access using the Object property are:

- DataWindow objects in DataWindow controls
- External OLE objects in OLE controls

These expressions refer to properties of the DataWindow object inside a DataWindow control:

```
dw_data.Object.emp_lname.Border
dw_data.Object.nestedrpt[1].Object.salary.Border
```

No compiler checking For objects inside the container, the compiler cannot be sure that the dot notation is valid. For example, the DataWindow object is not bound to the control and can be changed at any time. Therefore, the names and properties after the Object property are checked for validity during execution only. Incorrect references cause an execution error.

For more information For more information about runtime checking, see “Optimizing expressions for DataWindow and external objects” on page 35.

For more information about dot notation for properties and data of DataWindow objects and handling errors, see the *DataWindow Reference*.

For more information about OLE objects and dot notation for OLE automation, see Chapter 19, “Using OLE in an Application.”

Constant declarations

To declare a constant, add the keyword CONSTANT to a standard variable declaration:

CONSTANT { *access* } *datatype constname* = *value*

Only a datatype that accepts an assignment in its declaration can be a constant. For this reason, blobs cannot be constants.

Even though identifiers in PowerScript are not case sensitive, the declarations shown here use uppercase as a convention for constant names:

```
CONSTANT integer GI_CENTURY_YEARS = 100
CONSTANT string IS_ASCENDING = "a"
```

Advantages of constants

If you try to assign a value to the constant anywhere other than in the declaration, you get a compiler error. A constant is a way of assuring that the declaration is used the way you intend.

Constants are also efficient. Because the value is established during compilation, the compiled code uses the value itself, rather than referring to a variable that holds the value.

Controlling access for instance variables

Instance variables have access settings that provide control over how other objects' scripts access them.

You can specify that a variable is:

- **Public** Accessible to any other object
- **Protected** Accessible only in scripts for the object and its descendants
- **Private** Accessible in scripts for the object only

For example:

```
public integer ii_currentvalue
CONSTANT public integer WARPFACTOR = 1.2
protected string is_starship

// Private values used in internal calculations
private integer ii_maxrpm
private integer ii_minrpm
```

You can further qualify access to public and protected variables with the modifiers PRIVATEREAD, PRIVATEWRITE, PROTECTEDREAD, or PROTECTEDWRITE:

```
public privatewrite ii_averagerpm
```

Private variables for encapsulation

One use of access settings is to keep other scripts from changing a variable when they should not. You can use PRIVATE or PUBLIC PRIVATEWRITE to keep the variable from being changed directly. You might write public functions to provide validation before changing the variable.

Private variables allow you to encapsulate an object's functionality. This technique means that an object's data and code are part of the object itself and the object determines the interface it presents to other objects.

If you generate a component, such as an EAServer, COM+, or application server component, from a custom class user object, you can choose to expose its instance variables in the component's interface, but private and protected instance variables are never exposed.

For more information

For more about access settings, see the chapter about declarations in the *PowerScript Reference*.

For more about encapsulation, see Chapter 2, “Selected Object-Oriented Programming Topics.”

Resolving naming conflicts

There are two areas in which name conflicts occur:

- Variables that are defined within different scopes can have the same name. For example, a global variable can have the same name as a local or instance variable. The compiler warns you of these conflicts, but you do not have to change the names.
- A descendent object has functions and events that are inherited from the ancestor and have the same names.

If you need to refer to a hidden variable or an ancestor’s event or function, you can use dot notation qualifiers or the scope operator.

Hidden instance variables

If an instance variable has the same name as a local, shared, or global variable, qualify the instance variable with its object’s name:

objectname.*instancevariable*

If a local variable and an instance variable of a window are both named *birthdate*, then qualify the instance variable:

```
w_main.birthdate
```

If a window script defines a local variable *x*, the name conflicts with the X property of the window. Use a qualifier for the X property. This statement compares the two:

```
IF x > w_main.X THEN
```

Hidden global variables

If a global variable has the same name as a local or shared variable, you can access the global variable with the scope operator (::) as follows:

::*globalvariable*

This expression compares a local variable with a global variable, both named *total*:

```
IF total < ::total THEN ...
```

Use prefixes to avoid naming conflicts
If your naming conventions include prefixes that identify the scope of the variable, then variables of different scopes always have different names and there are no conflicts.

For more information about the search order that determines how name conflicts are resolved, see the chapters about declarations and calling functions and events in the *PowerScript Reference*.

Overridden functions and events

When you change the script for a function that is inherited, you override the ancestor version of the function. For events, you can choose to override or extend the ancestor event script in the painter.

You can use the scope operator to call the ancestor version of an overridden function or event. The ancestor class name, not a variable, precedes the colons:

```
result = w_ancestor:: FUNCTION of_func(arg1, arg2)
```

You can use the Super pronoun instead of naming an ancestor class. Super refers to the object's immediate ancestor:

```
result = Super:: EVENT ue_process()
```

In good object-oriented design, you would not call ancestor scripts for other objects. It is best to restrict this type of call to the current object's immediate ancestor using Super.

For how to capture the return value of an ancestor script, see "Return values from ancestor scripts" next.

Overloaded functions

When you have several functions of the same name for the same object, the function name is considered to be overloaded. PowerBuilder determines which version of the function to call by comparing the signatures of the function definitions with the signature of the function call. The signature includes the function name, argument list, and return value.

Overloading
Events and global functions cannot be overloaded.

Return values from ancestor scripts

If you want to perform some processing in an event in a descendent object, but that processing depends on the return value of the ancestor event script, you can use a local variable called *AncestorReturnValue* that is automatically declared and assigned the return value of the ancestor event.

The first time the compiler encounters a CALL statement that calls the ancestor event of a script, the compiler implicitly generates code that declares the *AncestorReturnValue* variable and assigns to it the return value of the ancestor event.

The datatype of the *AncestorReturnValue* variable is always the same as the datatype defined for the return value of the event. The arguments passed to the call come from the arguments that are passed to the event in the descendent object.

Extending event scripts

The *AncestorReturnValue* variable is always available in extended event scripts. When you extend an event script, PowerBuilder generates the following syntax and inserts it at the beginning of the event script:

CALL SUPER::*event_name*

You see the statement only if you export the syntax of the object.

Overriding event scripts

The *AncestorReturnValue* variable is available only when you override an event script after you call the ancestor event using the CALL syntax explicitly:

CALL SUPER::*event_name*

or

CALL *ancestor_name*::*event_name*

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

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

ancestor_name::EVENT *event_name* ( )

Example

You can put code like the following in an extended event script:

```
IF AncestorReturnValue = 1 THEN
  // execute some code
ELSE
  // execute some other code
END IF
```

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

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

Types of arguments for functions and events

When you define a function or user event, you specify its arguments, their datatypes, and how they are passed.

There are three ways to pass an argument:

- **By value** Is the default

  PowerBuilder passes a copy of a by-value argument. Any changes affect the copy, and the original value is unaffected.

- **By reference** Tells PowerBuilder to pass a pointer to the passed variable

  The function script can change the value of the variable because the argument points back to the original variable. An argument passed by reference must be a variable, not a literal or constant, so that it can be changed.

- **Read-only** Passes the argument by value without making a copy of the data

  Read-only provides a performance advantage for some datatypes because it does not create a copy of the data, as with by value. Datatypes for which read-only provides a performance advantage are String, Blob, Date, Time, and DateTime.

  For other datatypes, read-only provides documentation for other developers by indicating something about the purpose of the argument.

Matching argument types when overriding functions

If you define a function in a descendant that overrides an ancestor function, the function signatures must match in every way: the function name, return value, argument datatypes, and argument passing methods must be the same.

For example, this function declaration has two long arguments passed by value and one passed by reference:

```
uf_calc(long a_1, long a_2, ref long a_3) &
   returns integer
```

If the overriding function does not match, then when you call the function, PowerBuilder calculates which function matches more closely and calls that one, which might give unexpected results.

Ancestor and descendent variables

All objects in PowerBuilder are descendants of PowerBuilder system objects—the objects you see listed on the System page in the Browser.

Therefore, whenever you declare an object instance, you are declaring a descendant. You decide how specific you want your declarations to be.

As specific as possible

If you define a user object class named uo_empdata, you can declare a variable whose type is uo_empdata to hold the user object reference:

```
uo_empdata uo_emp1
uo_emp1 = CREATE uo_empdata
```

You can refer to the variables and functions that are part of the definition of uo_empdata because the type of *uo_emp1* is uo_empdata.

When the application requires flexibility

Suppose the user object you want to create depends on the user's choices. You can declare a user object variable whose type is UserObject or an ancestor class for the user object. Then you can specify the object class you want to instantiate in a string variable and use it with CREATE:

```
uo_empdata uo_emp1
string ls_objname
ls_objname = ... // Establish the user object to open
uo_emp1 = CREATE USING ls_objname
```

This more general approach limits your access to the object's variables and functions. The compiler knows only the properties and functions of the ancestor class uo_empdata (or the system class UserObject if that is what you declared). It does not know which object you will actually create and cannot allow references to properties defined on that unknown object.

Abstract ancestor object In order to address properties and functions of the descendants you plan to instantiate, you can define the ancestor object class to include the properties and functions that you will implement in the descendants. In the ancestor, the functions do not need code other than a return value—they exist so that the compiler can recognize the function names. When you declare a variable of the ancestor class, you can reference the functions. During execution, you can instantiate the variable with a descendant, where that descendant implements the functions as appropriate:

```
uo_empdata uo_emp1
string ls_objname
// Establish which descendant of uo_empdata to open
ls_objname = ...
uo_emp1 = CREATE USING ls_objname

// Function is declared in the ancestor class
result = uo_emp1.uf_special()
```

This technique is described in more detail in “Dynamic versus static lookup” on page 17.

Dynamic function calls Another way to handle functions that are not defined for the declared class is to use dynamic function calls.

When you use the DYNAMIC keyword in a function call, the compiler does not check whether the function call is valid. The checking happens during execution when the variable has been instantiated with the appropriate object:

```
// Function not declared in the ancestor class
result = uo_emp1.DYNAMIC uf_special()
```

Performance and errors
You should avoid using the dynamic capabilities of PowerBuilder when your application design does not require them. Runtime evaluation means that work the compiler usually does must be done at runtime, making the application slower when dynamic calls are used often or used within a large loop. Skipping compiler checking also means that errors that might be caught by the compiler are not found until the user is executing the program.

Dynamic object selection for windows and visual user objects

A window or visual user object is opened with a function call instead of the CREATE statement. With the Open and OpenUserObject functions, you can specify the class of the window or object to be opened, making it possible to open a descendant different from the declaration’s object type.

This example displays a user object of the type specified in the string s_u_name and stores the reference to the user object in the variable *u_to_open*. Variable *u_to_open* is of type DragObject, which is the ancestor of all user objects. It can hold a reference to any user object:

```
DragObject u_to_open
string s_u_name
s_u_name = sle_user.Text
w_info.OpenUserObject(u_to_open, s_u_name, 100, 200)
```

For a window, comparable code looks like this. The actual window opened could be the class w_data_entry or any of its descendants:

```
w_data_entry w_data
string s_window_name
s_window_name = sle_win.Text
Open(w_data, s_window_name)
```

Optimizing expressions for DataWindow and external objects

No compiler validation for container objects

When you use dot notation to refer to a DataWindow object in a DataWindow control or DataStore, the compiler does not check the validity of the expression:

```
dw_data.Object.column.property
```

Everything you specify after the Object property passes the compiler and is checked during execution.

The same applies to external OLE objects. No checking occurs until execution:

```
ole_1.Object.qualifier.qualifier.property.Value
```

Establishing partial references

Because of the runtime syntax checking, using many expressions like these can impact performance. To improve efficiency when you refer repeatedly to the same DataWindow component object or external object, you can define a variable of the appropriate type and assign a partial reference to the variable. The script evaluates most of the reference only once and reuses it.

The datatype of a DataWindow component object is DWObject:

```
DWObject dwo_column
dwo_column = dw_data.Object.column
```

```
dwo_column.SlideLeft = ...
dwo_column.SlideUp = ...
```

The datatype of a partially resolved automation expression is OLEObject:

```
OLEObject ole_wordbasic
ole_wordbasic = ole_1.Object.application.wordbasic
ole_wordbasic.propertyname1 = value
ole_wordbasic.propertyname2 = value
```

Handling errors

The Error and (for automation) ExternalException events are triggered when errors occur in evaluating the DataWindow and OLE expressions. If you write a script for these events, you can catch an error before it triggers the SystemError event. These events allow you to ignore an error or substitute an appropriate value. However, you must be careful to avoid setting up conditions that cause another error. You can also use try-catch blocks to handle exceptions as described in "Exception handling in PowerBuilder" next.

For information

For information about DataWindow data expressions and property expressions and DWObject variables, see the *DataWindow Reference*. For information about using OLEObject variables in automation, see Chapter 19, “Using OLE in an Application.”

Exception handling in PowerBuilder

When a runtime error occurs in a PowerBuilder application, unless that error is trapped, a single application event (SystemError) fires to handle the error no matter where in the application the error happened. Although some errors can be handled in the system error event, catching the error closer to its source increases the likelihood of recovery from the error condition.

You can use exception-handling classes and syntax to handle context-sensitive errors in PowerBuilder applications. This means that you can deal with errors close to their source by embedding error-handling code anywhere in your application. Well-designed exception-handling code can give application users a better chance to recover from error conditions and run the application without interruption.

Exception handling allows you to design an application that can recover from exceptional conditions and continue execution. Any exceptions that you do not catch are handled by the runtime system and can result in the termination of the application.

PowerBuilder clients can catch exceptions thrown from application server components and recover from them. Components developed with PowerBuilder can also define and throw their own exception types, making them more consistent with other server component types like Java.

Exception handling can be found in such object-oriented languages as Java and C++. The implementation for PowerBuilder is similar to the implementation of exception handling in Java. In PowerBuilder, the TRY, CATCH, FINALLY, THROW, and THROWS reserved words are used for exception handling. There are also several PowerBuilder objects that descend from the Throwable object.

Basics of exception handling

Exceptions are objects that are thrown in the event of some exceptional (or unexpected) condition or error and are used to describe the condition or error encountered. Standard errors, such as null object references and division by zero, are typically thrown by the runtime system. These types of errors could occur anywhere in an application and you can include **catch** clauses in any executable script to try to recover from these errors.

User-defined exceptions

There are also exceptional conditions that do not immediately result in runtime errors. These exceptions typically occur during execution of a function or a user-event script. To signal these exceptions, you create user objects that inherit from the PowerScript Exception class. You can associate a user-defined exception with a function or user event in the prototype for the method.

For example, a user-defined exception might be created to indicate that a file cannot be found. You could declare this exception in the prototype for a function that is supposed to open the file. To catch this condition, you must instantiate the user-defined exception object and then **throw** the exception instance in the method script.

Objects for exception handling support

Several system objects support exception handling within PowerBuilder.

Throwable object type

The object type **Throwable** is the root datatype for all user-defined exception and system error types. Two other system object types, RuntimeError and Exception, derive from Throwable.

RuntimeError and its descendants

PowerBuilder runtime errors are represented in the **RuntimeError** object type. For more robust error-handling capabilities, the RuntimeError type has its own system-defined descendants; but the RuntimeError type contains all information required for dealing with PowerBuilder runtime errors.

One of the descendants of RuntimeError is the NullObjectError type that is thrown by the system whenever a null object reference is encountered. This allows you to handle null-object-reference errors explicitly without having to differentiate them from other runtime errors that might occur.

Error types that derive from RuntimeError are typically used by the system to indicate runtime errors. RuntimeErrors can be caught in a try-catch block, but it is not necessary to declare where such an error condition might occur. (PowerBuilder does that for you, since a system error can happen anywhere anytime the application is running.) It is also not a requirement to catch these types of errors.

Exception object type

The system object **Exception** also derives from Throwable and is typically used as an ancestor object for user-defined exception types. It is the root class for all checked exceptions. **Checked exceptions** are user-defined exceptions that must be caught in a try-catch block when thrown, or that must be declared in the prototype of a method when thrown outside of a try-catch block.

The PowerScript compiler checks the local syntax where you throw checked exceptions to make sure you either declare or catch these exception types. Descendants of RuntimeError are not checked by the compiler, even if they are user defined or if they are thrown in a script rather than by the runtime system.

Handling exceptions

Whether an exception is thrown by the runtime system or by a THROW statement in an application script, you handle the exception by catching it. This is done by surrounding the set of application logic that throws the exception with code that indicates how the exception is to be dealt with.

TRY-CATCH-FINALLY block

To handle an exception in PowerScript, you must include some set of your application logic inside a try-catch block. A try-catch block begins with a TRY clause and ends with the END TRY statement. It must also contain either a CATCH clause or a FINALLY clause. A try-catch block normally contains a FINALLY clause for error condition cleanup. In between the TRY and FINALLY clauses you can add any number of CATCH clauses.

CATCH clauses are not obligatory, but if you do include them you must follow each CATCH statement with a variable declaration. In addition to following all of the usual rules for local variable declarations inside a script, the variable being defined must derive from the Throwable system type.

You can add a TRY-CATCH-FINALLY, TRY-CATCH, or TRY-FINALLY block using the Script view Paste Special feature for PowerScript statements. If you select the Statement Templates check box on the AutoScript tab of the Design Options dialog box, you can also use the AutoScript feature to insert these block structures.

Example

Example catching a system error This is an example of a TRY-CATCH-FINALLY block that catches a system error when an arccosine argument, entered by the application user (in a SingleLineEdit) is not in the required range. If you do not catch this error, the application goes to the system error event, and eventually terminates:

```
Double ld_num
ld_num = Double (sle_1.text)
TRY
   sle_2.text = string (acos (ld_num))
CATCH (runtimeerror er)
   MessageBox("Runtime Error", er.GetMessage())
FINALLY
   // Add cleanup code here
   of_cleanup()
   Return
END TRY
MessageBox("After", "We are finished.")
```

The system runtime error message might be confusing to the end user, so for production purposes, it would be better to catch a user-defined exception—see the example in “Creating user-defined exception types” on page 40—and set the message to something more understandable.

The TRY reserved word signals the start of a block of statements to be executed and can include more than one CATCH clause. If the execution of code in the TRY block causes an exception to be thrown, then the exception is handled by the first CATCH clause whose variable can be assigned the value of the exception thrown. The variable declaration after a CATCH statement indicates the type of exception being handled (a system runtime error, in this case).

CATCH order

It is important to order your CATCH clauses in such a way that one clause does not hide another. This would occur if the first CATCH clause catches an exception of type Exception and a subsequent CATCH clause catches a descendant of Exception. Since they are processed in order, any exception thrown that is a descendant of Exception would be handled by the first CATCH clause and never by the second. The PowerScript compiler can detect this condition and signals an error if found.

If an exception is not dealt with in any of the CATCH clauses, it is thrown up the call stack for handling by other exception handlers (nested try-catch blocks) or by the system error event. But before the exception is thrown up the stack, the FINALLY clause is executed.

FINALLY clause

The FINALLY clause is generally used to clean up after execution of a TRY or CATCH clause. The code in the FINALLY clause is guaranteed to execute if any portion of the try-catch block is executed, regardless of how the code in the try-catch block completes.

If no exceptions occur, the TRY clause completes, followed by the execution of the statements contained in the FINALLY clause. Then execution continues on the line following the END TRY statement.

In cases where there are no CATCH clauses but only a FINALLY clause, the code in the FINALLY clause is executed even if a return is encountered or an exception is thrown in the TRY clause.

If an exception occurs within the context of the TRY clause and an applicable CATCH clause exists, the CATCH clause is executed, followed by the FINALLY clause. But even if no CATCH clause is applicable to the exception thrown, the FINALLY clause still executes before the exception is thrown up the call stack.

If an exception or a return is encountered within a CATCH clause, the FINALLY clause is executed before execution is transferred to the new location.

Creating user-defined exception types

You can create your own user-defined exception types from standard class user objects that inherit from Exception or RuntimeError or that inherit from an existing user object deriving from Exception or RuntimeError.

Inherit from Exception object type

Normally, user-defined exception types should inherit from the Exception type or a descendant, since the RuntimeError type is used to indicate system errors. These user-defined objects are no different from any other nonvisual user object in the system. They can contain events, functions, and instance variables.

This is useful, for example, in cases where a specific condition, such as the failure of a business rule, might cause application logic to fail. If you create a user-defined exception type to describe such a condition and then catch and handle the exception appropriately, you can prevent a runtime error.

Throwing exceptions

Exceptions can be thrown by the runtime engine to indicate an error condition. If you want to signal a potential exception condition manually, you must use the THROW statement.

Typically, the THROW statement is used in conjunction with some user-defined exception type. Here is a simple example of the use of the THROW statement:

```
Exception    le_ex
le_ex = create Exception
Throw le_ex
MessageBox ("Hmm", "We would never get here if" &
   + "the exception variable was not instantiated")
```

In this example, the code throws the instance of the exception le_ex. The variable following the THROW reserved word must point to a valid instance of the exception object that derives from Throwable. If you attempt to throw an uninstantiated Exception variable, a NullObjectError is thrown instead, indicating a null object reference in this routine. That could only complicate the error handling for your application.

Declaring exceptions thrown from functions

If you signal an exception with the THROW statement inside a method script—and do not surround the statement with a try-catch block that can deal with that type of exception—you must also declare the exception as an exception type (or as a descendant of an exception type) thrown by that method. However, you do not need to declare that a method can throw runtime errors, since PowerBuilder does that for you.

The prototype window in the Script view of most PowerBuilder painters allows you to declare what user-defined exceptions, if any, can be thrown by a function or a user-defined event. You can drag and drop exception types from the System Tree or a Library painter view to the Throws box in the prototype window, or you can type in a comma-separated list of the exception types that the method can throw.

Example

Example catching a user-defined exception This code displays a user-defined error when an arccosine argument, entered by the application user, is not in the required range. The try-catch block calls a method, wf_acos, that catches the system error and sets and throws the user-defined error:

```
TRY
   wf_acos()
```

```
CATCH (uo_exception u_ex)
   MessageBox("Out of Range", u_ex.GetMessage())
END TRY
```

This code in the wf_acos method catches the system error and sets and throws the user-defined error:

```
uo_exception lu_error
Double ld_num
ld_num = Double (sle_1.text)
TRY
   sle_2.text = string (acos (ld_num))
CATCH (runtimeerror er)
   lu_error = Create uo_exception
   lu_error.SetMessage("Value must be between -1" &
      + "and 1")
   Throw lu_error
END TRY
```

Integration with EAServer

If you declare exceptions on a method of a user object and deploy the user object as a component to EAServer, the exceptions are translated to IDL (CORBA) as part of the method prototype. This means that PowerBuilder components in EAServer can be defined to throw exceptions that can be handled by any type of EAServer client application.

Other benefits for EAServer applications Another benefit for component development is that you can handle runtime errors in the component. If you do not handle an error, it is automatically translated into an exception and the component stops executing.

PowerBuilder client applications that use EAServer components can handle exceptions thrown by any type of EAServer component. If a Java EAServer component has a method on it that is defined to throw an exception and a PowerBuilder proxy is created to use that component, the method on the PowerBuilder proxy is also declared to throw a user-defined exception. The definition of the user-defined exception is created automatically at the time of the PowerBuilder proxy creation.

For more information about error handling in EAServer clients, see “Handling errors” on page 535.

IDL restrictions Deployment of components to EAServer imposes restrictions on the way you can use exception handling within PowerBuilder. Only the public instance variables defined on the exception type are actually translated to IDL. This is because IDL exceptions cannot have methods declared on them. Therefore if the exception type has methods defined on it, those methods can be called within the execution of the component but cannot be called by client applications that catch the exception thrown.

You must keep this restriction in mind when designing exception objects for distributed applications, exposing all exception information as public instance variables instead of through accessor methods on an exception object.

Two other interface restrictions also apply to exception types of a user object that is deployed as an EAServer component. Instance variables of exceptions on the user object methods cannot have object datatypes. Null data is supported only for instance variables with simple datatypes; if instance variables are structures or arrays, null values for individual elements are not maintained.

Adding flexibility and facilitating object reuse

You can use exception handling to add flexibility to your PowerBuilder applications, and to help in the separation of business rules from presentation logic. For example, business rules can be stored in a non-visual object (nvo) that has:

- An instance variable to hold a reference to the presentation object:

  ```
  powerobject my_presenter
  ```

- A function that registers the presentation object

  The registration function could use the following syntax:

  ```
  SetObject (string my_purpose, powerobject myobject)
  ```

- Code to call a dynamic function implemented by the presentation object, with minimal assumptions about how the data is displayed

  The dynamic function call should be enclosed in a try-catch block, such as:

  ```
  TRY
        my_presenter.Dynamic nf_displayScreen(" ")
     CATCH (Throwable lth_exception)
        Throw lth_exception
  END TRY
  ```

This try-catch block catches all system and user-defined errors from the presentation object and throws them back up the calling chain (to the object that called the nvo). In the above example, the thrown object in the CATCH statement is an object of type Throwable, but you could also instantiate and throw a user exception object:

```
uo_exception luo_exception

TRY
      my_presenter.Dynamic nf_displayScreen(" ")
CATCH (Throwable lth_exception)
      luo_exception = Create uo_exception
      luo_exception.SetMessage & +
      (lth_exception.GetMessage())
      Throw luo_exception
END TRY
```

Code for data processing could be added to the presentation object, to the business rules nvo, or to processing objects called by the nvo. The exact design depends on your business objectives, but this code should also be surrounded by try-catch blocks. The actions to take and the error messages to report (in case of code processing failure) should be as specific as possible in the try-catch blocks that surround the processing code.

There are significant advantages to this type of approach, since the business nvo can be reused more easily, and it can be accessed by objects that display the same business data in many different ways. The addition of exception handling makes this approach much more robust, giving the application user a chance to recover from an error condition.

Using the SystemError and Error events

Error event

If a runtime error occurs, an error structure that describes the error is created. If the error occurs in the context of a connection to a remote server (such as EAServer) then the Error event on the Connection, JaguarORB, DataWindow, or OLE control object is triggered, with the information in the error structure as arguments.

The error can be handled in this Error event by use of a special reference argument that allows the error to be ignored. If the error does not occur in the context described above, or if the error in that context is not dealt with, then the error structure information is used to populate the global error variable and the SystemError event on the Application object is triggered.

SystemError event

In the SystemError event, unexpected error conditions can be dealt with in a limited way. In general, it is not a good idea to continue running the application after the SystemError event is triggered. However, error-handling code can and should be added to this event. Typically you could use the SystemError event to save data before the application terminates and to perform last-minute cleanup (such as closing files or database connections).

Precedence of exception handlers and events

If you write code in the Error event, then that code is executed first in the event of a thrown exception.

If the exception is not thrown in any of the described contexts or the object's Error event does not handle the exception or you do not code the Error event, then the exception is handled by any active exception handlers (CATCH clauses) that are applicable to that type of exception. Information from the exception class is copied to the global error variable and the SystemError event on the Application object is fired only if there are no exception handlers to handle the exception.

Error handling for new applications

For new PowerBuilder applications, the recommended approach for handling errors is to use a try-catch block instead of coding the Error event on Connection, DataWindow, or OLE control objects. You should still have a SystemError event coded in your Application object to handle any uncaught exceptions. The SystemError event essentially becomes a global exception handler for a PowerBuilder application.

Garbage collection and memory management

The PowerBuilder garbage collection mechanism checks memory automatically for unreferenced and orphaned objects and removes any it finds, thus taking care of most memory leaks. You can use garbage collection to destroy objects instead of explicitly destroying them using the DESTROY statement. This lets you avoid runtime errors that occur when you destroy an object that was being used by another process or had been passed by reference to a posted event or function.

A reference to an object is any variable whose value is the object. When the variable goes out of scope, or when it is assigned a different value, PowerBuilder removes a reference to the object and counts the remaining references, and the garbage collection process destroys the object if no references remain.

Garbage collection occurs:

- When the garbage collection interval has been exceeded and the PowerBuilder application becomes idle and
- When you explicitly call the GarbageCollect function.

When PowerBuilder completes the execution of a system-triggered event, it makes a garbage collection pass if the set interval between garbage collection passes has been exceeded. The default interval is 0.5 seconds. Note that this system-triggered garbage collection pass only occurs when the PowerBuilder application is idle, therefore if a long computation or process is in progress when the interval is exceeded, garbage collection does not occur immediately.

You can force immediate garbage collection by invoking the GarbageCollect function. When you use dot notation and OLEObjects, temporary variables are created. These temporary variables are released only during the garbage collection process. You might want to invoke GarbageCollect inside a loop that appears to be causing memory leaks.

The garbage collection pass removes any objects and classes that cannot be referenced, including those containing circular references (otherwise unreferenced objects that reference each other).

Posting events and functions
When you post an event or function and pass an object reference, PowerBuilder adds an internal reference to the object to prevent its memory from being reclaimed by the garbage collector between the time of the post and the actual execution of the event or function. This reference is removed when the event or function is executed.

Exceptions to garbage collection

There are a few objects that are prevented from being collected:

- **Visual objects** Any object that is visible on your screen is not collected because when the object is created and displayed on your screen, an internal reference is added to the object. When any visual object is closed, it is explicitly destroyed.
- **Timing objects** Any Timing object that is currently running is not collected because the Start function for a Timing object adds an internal reference. The Stop function removes the reference.
- **Shared objects** Registered shared objects are not collected because the SharedObjectRegister function adds an internal reference. SharedObjectUnregister removes the internal reference.

Controlling when garbage collection occurs

Garbage collection occurs automatically in PowerBuilder, but you can use functions to force immediate garbage collection or to change the interval between reference count checks. Three functions let you control when garbage collection occurs: GarbageCollect, GarbageCollectGetTimeLimit, and GarbageCollectSetTimeLimit.

For information about these functions, see the *PowerScript Reference*. For an example illustrating their use, see the Code Examples sample application, described in Chapter 1, "Using Sample Applications."

Performance concerns

You can use tracing and profiling to examine the effect of changing the garbage collection interval on performance.

For information about tracing and profiling, see the PowerBuilder *User's Guide*.

Configuring memory management

You can set the PB_POOL_THRESHOLD environment variable to specify the threshold at which the PowerBuilder memory manager switches to a different memory allocation strategy.

When most windows, DataWindows, DataStores, or other PowerBuilder objects are destroyed or reclaimed by the garbage collector, the PowerBuilder heap manager returns the memory allocated for each object to a global memory pool and records its availability on a global free list. The freed memory is not returned to the operating system. When a new object is created, PowerBuilder allocates blocks of memory from the global memory pool (if sufficient memory is available in the global free list) or from the operating system (if it is not) to a memory pool for the object.

When the memory required by an object exceeds 256KB, PowerBuilder uses a different strategy. It allocates subsequent memory requirements from the operating system in large blocks, and returns the physical memory to the operating system when the object is destroyed. It retains the virtual memory to reduce fragmentation of the virtual address space.

For most applications and components, the threshold of 256KB at which PowerBuilder switches to the "large blocks" strategy works well and reduces the memory required by an application when it is working at its peak level of activity. However, if you want to keep the overall physical memory usage of your application as low as possible, you can try setting a lower threshold.

The advantage of setting a low threshold is that the size of the global memory pool is reduced. The application does not retain a lot of memory when it is inactive. The disadvantage is that large blocks of memory are allocated for objects that require more memory than the threshold value, so that when the application is running at its peak of activity, it might use more virtual memory than it would with the default threshold.

Setting a low threshold can be beneficial for long-running client applications that use many short-lived objects, where the client application's memory usage varies from low (when idle) to high (when active). For multithreaded applications, such as servers, a higher threshold usually results in lower virtual memory utilization.

Logging heap manager output

You can record diagnostic ouput from the PowerBuilder heap manager in a file to help you troubleshoot memory allocation issues in your application. The PB_HEAP_LOGFILENAME environment variable specifies the name and location of the file.

If you specify a file name but not a directory, the file is saved in the same directory as the PowerBuilder executable, or, for a PowerBuilder component running on EAServer, to the EAServer *bin* directory.

If you specify a directory that does not exist, the file is not created, or, for a PowerBuilder component running on EAServer, output is written to the EAServer log file (*Jaguar.log*).

By default, the log file is overwritten when you restart PowerBuilder or EAServer. If you want diagnostic output to be appended to the file, set PB_HEAP_LOGFILE_OVERWRITE to false.

You can set the variables in a batch file that launches the application, or as system or user environment variables on the computer or server on which the application or component runs.

For more information about tuning memory management in PowerBuilder and EAServer, see the technical document EAServer/PowerBuilder Memory Tuning and Troubleshooting at http://www.sybase.com/detail?id=1027319.

Efficient compiling and performance

The way you write functions and define variables affects your productivity and your application's performance.

Short scripts for faster compiling

If you plan to build machine code dynamic libraries for your deployed application, keep scripts for functions and events short. Longer scripts take longer to compile. Break the scripts up so that instead of one long script, you have a script that makes calls to several other functions. Consider defining functions in user objects so that other objects can call the same functions.

Local variables for faster performance

The scope of variables affects performance. When you have a choice, use local variables, which provide the fastest performance. Global variables have the biggest negative impact on performance.

Reading and writing text or binary files

You use PowerScript text file functions to read and write text in line mode or text mode, or to read and write binary files in stream mode:

- In *line mode*, you can read a file a line at a time until either a carriage return or line feed (CR/LF) or the end-of-file (EOF) is encountered. When writing to the file after the specified string is written, PowerScript appends a CR/LF.
- In *stream mode*, you can read the entire contents of the file, including any CR/LFs. When writing to the file, you must write out the specified blob (but not append a CR/LF).
- In *text mode*, you can read the entire contents of the file, including any CR/LFs. When writing to the file, you must write out the specified string (but not append a CR/LF).

Reading a file into a MultiLineEdit
You can use stream mode to read an entire file into a MultiLineEdit, and then write it out after it has been modified.

Understanding the position pointer

When PowerBuilder opens a file, it assigns the file a unique integer and sets the position pointer for the file to the position you specify—the beginning, after the byte-order mark, if any, or end of the file. You use the integer to identify the file when you want to read the file, write to it, or close it. The position pointer defines where the next read or write will begin. PowerBuilder advances the pointer automatically after each read or write.

You can also set the position pointer with the FileSeek or FileSeek64 function.

File functions

These are the built-in PowerScript functions that manipulate files:

Table 3-1: PowerScript functions that manipulate files

Function	Datatype returned	Action
FileClose	Integer	Closes the specified file
FileDelete	Boolean	Deletes the specified file
FileEncoding	Encoding enumerated type	Returns the encoding used in the file
FileExists	Boolean	Determines whether the specified file exists
FileLength	Long	Obtains the length of a file with a file size of 2GB or less
FileLength64	LongLong	Obtains the length of a file of any size
FileOpen	Integer	Opens the specified file
FileRead	Integer	Reads from the specified file (deprecated)
FileReadEx	Long	Reads from the specified file
FileSeek	Long	Seeks to a position in a file with a file size of 2GB or less
FileSeek64	LongLong	Seeks to a position in a file of any size
FileWrite	Integer	Writes to the specified file (deprecated)
FileWriteEx	Long	Writes to the specified file

Encoding

The last argument in the FileOpen function lets you create an ANSI, UTF-8, UTF-16LE (Little Endian), or UTF16-BE (Big Endian) file.

The *encoding* argument, like all arguments of the FileOpen function except the file name, is optional. You need only specify it if you want to create a new text file with Unicode encoding. If the *filename* argument refers to a file that does not exist, the FileOpen function creates the file and sets the character encoding specified in the *encoding* argument.

By default, if the file does not exist and the encoding argument is not specified, PowerBuilder opens a file with ANSI encoding. This ensures compatibility with earlier versions of PowerBuilder.

The FileRead and FileWrite functions cannot read more than 32,766 bytes at a time. The FileReadEx and FileWriteEx functions can write an unlimited number of bytes at a time.

CHAPTER 4

Getting Information About PowerBuilder Class Definitions

About this chapter

This chapter explains what class definition information is and how it is used, and presents some sample code. Developers of tools and object frameworks can use class definition information for tasks such as producing reports or defining objects with similar characteristics. You do not need to use class definition information if you are building typical business applications.

Contents

Overview of class definition information

A ClassDefinition object is a PowerBuilder object that provides information about the class of another PowerBuilder object. You can examine a class in a PowerBuilder library or the class of an instantiated object. By examining the properties of its ClassDefinition object, you can get details about how a class fits in the PowerBuilder object hierarchy.

From the ClassDefinition object, you can discover:

- The variables, functions, and events defined for the class
- The class’s ancestor
- The class’s parent
- The class’s children (nested classes)

Related objects
The ClassDefinition object is a member of a hierarchy of objects, including the TypeDefinition, VariableDefinition, and ScriptDefinition objects, that provide information about datatypes or about the variables, properties, functions, and event scripts associated with a class definition.

For more information, see the Browser or *Objects and Controls*.

Definitions for instantiated objects For each object instance, a ClassDefinition property makes available a ClassDefinition object to describe its definition. The ClassDefinition object does not provide information about the object instance, such as the values of its variables. You get that information by addressing the instance directly.

Definitions for objects in libraries An object does not have to be instantiated to get class information. For an object in a PowerBuilder library, you can call the FindClassDefinition function to get its ClassDefinition object.

Performance Class definition objects may seem to add a lot of overhead, but the overhead is incurred only when you refer to the ClassDefinition object. The ClassDefinition object is instantiated only when you call FindClassDefinition or access the ClassDefinition property of a PowerBuilder object. Likewise, for properties of the ClassDefinition object that are themselves ClassDefinition or VariableDefinition objects, the objects are instantiated only when you refer to those properties.

Terminology

The class information includes information about the relationships between objects. These definitions will help you understand what the information means.

object instance

A realization of an object. The instance exists in memory and has values assigned to its properties and variables. Object instances exist only when you run an application.

class

A definition of an object, containing the source code for creating an object instance. When you use PowerBuilder painters and save an object in a PBL, you are creating class definitions for objects. When you run your application, the class is the datatype of object instances based on that class. In PowerBuilder, the term *object* usually refers to an instance of the object. It sometimes refers to an object’s class.

system class

A class defined by PowerBuilder. An object you define in a painter is a descendant of a system class, even when you do not explicitly choose to use inheritance for the object you define.

parent

The object that contains the current object or is connected to the object in a way other than inheritance. This table lists classes of objects and the classes that can be the parents of those objects:

Table 4-1: Classes of objects and their parents

Object	Parent
Window	The window that opened the window. A window might not have a parent. The parent is determined during execution and is not part of the class definition.
Menu item	The menu item on the prior level in the menu. The item on the menu bar is the parent of all the items on the associated drop-down menu.
Control on a window	The window.
Control on user object	The user object.
TabPage	The Tab control in which the TabPage is defined or in which it was opened.
ListViewItem or TreeViewItem	The ListView or TreeView control.
Visual user object	The window or user object on which the user object is placed.

child

A class that is contained within another parent class. Also called a nested class. For the types of objects that have a parent and child relationship, see parent.

ancestor

A class from whose definition another object is inherited. See also descendant.

descendant

An object that is inherited from another object and that incorporates the specifics of that object: its properties, functions, events, and variables. The descendant can use these values or override them with new definitions. All objects you define in painters and store in libraries are descendants of PowerBuilder system classes.

inheritance hierarchy

An object and all its ancestors.

collapsed hierarchy

A view of an object class definition that includes information from all the ancestors in the object's inheritance tree, not just items defined at the current level of inheritance.

scalar

A simple datatype that is not an object or an array. For example, Integer, Boolean, Date, Any, and String.

instance variable and property

Built-in properties of PowerBuilder system objects are called properties, but they are treated as instance variables in the class definition information.

Who uses PowerBuilder class definitions

Most business applications do not need to use class definition information. Code that uses class definition information is written by groups that write class libraries, application frameworks, and productivity tools.

Although your application might not include any code that uses class definition information, tools that you use for design, documentation, and class libraries will. These tools examine class definitions for your objects so that they can analyze your application and provide feedback to you.

Scenarios Class information might be used when developing:

- A custom object browser
- A tool that needs to know the objects of an application and their relationships

  The purpose might be to document the application or to provide a logical way to select and work with the objects.
- A CASE tool that deconstructs PowerBuilder objects, allows the user to redesign them, and reconstructs them

  To do the reconstruction, the CASE tool needs both class definition information and a knowledge of PowerBuilder object source code syntax.
- A class library in which objects need to determine the class associated with an instantiated object, or a script needs to know the ancestor of an object in order to make assumptions about available methods and variables

Examining a class definition

This section illustrates how to access a class definition object and how to examine its properties to get information about the class, its scripts, and its variables.

Getting a class definition object

To work with class information, you need a class definition object. There are two ways to get a ClassDefinition object containing class definition information.

For an instantiated object in your application

Use its ClassDefinition property.

For example, in a script for a button, this code gets the class definition for the parent window:

```
ClassDefinition cd_windef
cd_windef = Parent.ClassDefinition
```

For an object stored in a PBL

Call FindClassDefinition.

For example, in a script for a button, this code gets the class definition for the window named w_genapp_frame from a library on the application's library list:

```
ClassDefinition cd_windef
cd_windef = FindClassDefinition("w_genapp_frame")
```

Getting detailed information about the class

This section has code fragments illustrating how to get information from a ClassDefinition object called cd_windef.

For examples of assigning a value to cd_windef, see "Getting a class definition object."

Library

The LibraryName property reports the name of the library a class has been loaded from:

```
s = cd_windef.LibraryName
```

Ancestor

The Ancestor property reports the name of the class from which this class is inherited. All objects are inherited from PowerBuilder system objects, so the Ancestor property can hold a ClassDefinition object for a PowerBuilder class. The Ancestor property contains a null object reference when the ClassDefinition is for PowerObject, which is the top of the inheritance hierarchy.

This example gets a ClassDefinition object for the ancestor of the class represented by cd_windef:

```
ClassDefinition cd_ancestorwindef
cd_ancestorwindef = cd_windef.Ancestor
```

This example gets the ancestor name. Note that this code would cause an error if cd_windef held the definition of PowerObject, because the Ancestor property would be NULL:

```
ls_name = cd_windef.Ancestor.Name
```

Use the IsValid function to test that the object is not NULL.

This example walks back up the inheritance hierarchy for the window w_genapp_frame and displays a list of its ancestors in a MultiLineEdit:

```
string s, lineend
ClassDefinition cd
lineend = "~r~n"

cd = cd_windef
s = "Ancestor tree:" + lineend

DO WHILE IsValid(cd)
   s = s + cd.Name + lineend
   cd = cd.Ancestor
LOOP

mle_1.Text = s
```

The list might look like this:

```
Ancestor tree:
w_genapp_frame
window
graphicobject
powerobject
```

Parent

The ParentClass property of the ClassDefinition object reports the parent (its container) specified in the object's definition:

```
ClassDefinition cd_parentwindef
cd_parentwindef = cd_windef.ParentClass
```

If the class has no parent, ParentClass is a null object reference. This example tests that ParentClass is a valid object before checking its Name property:

```
IF IsValid(cd_windef.ParentClass) THEN
   ls_name = cd_windef.ParentClass.Name
END IF
```

Nested or child classes

The ClassDefinition object's NestedClassList array holds the classes the object contains.

NestedClassList array includes ancestors and descendants
The NestedClassList array can include classes of ancestor objects. For example, a CommandButton defined on an ancestor window and modified in a descendent window appears twice in the array for the descendent window, once for the window and once for its ancestor.

This script produces a list of the controls and structures defined for the window represented in cd_windef.

```
string s, lineend
integer li
lineend = "~r~n"

s = s + "Nested classes:" + lineend

FOR li = 1 to UpperBound(cd_windef.NestedClassList)
   s = s + cd_windef.NestedClassList[li].Name &
      + lineend
NEXT
mle_1.Text = s
```

This script searches the NestedClassList array in the ClassDefinition object cd_windef to find a nested DropDownListBox control:

```
integer li
ClassDefinition nested_cd

FOR li = 1 to UpperBound(cd_windef.NestedClassList)
   IF cd_windef.NestedClassList[li].DataTypeOf &
      = "dropdownlistbox" THEN
      nested_cd = cd_windef.NestedClassList[li]
      EXIT
   END IF
NEXT
```

Class definitions for object instances as distinct from object references
Getting a ClassDefinition object for an instantiated object, such as an ancestor or nested object, does not give you a reference to instances of the parent or child classes. Use standard PowerBuilder programming techniques to get and store references to your instantiated objects.

Getting information about a class's scripts

This section has code fragments illustrating how to get script information from a ClassDefinition object called cd_windef.

For examples of assigning a value to cd_windef, see "Getting a class definition object" on page 55.

List of scripts

The ScriptList array holds ScriptDefinition objects for all the functions and events defined for a class. If a function is overloaded, it will appear in the array more than once with different argument lists. If a function or event has code at more than one level in the hierarchy, it will appear in the array for each coded version.

This example loops through the ScriptList array and builds a list of script names. All objects have a few standard functions, such as ClassName and PostEvent, because all objects are inherited from PowerObject:

```
string s, lineend
integer li
ScriptDefinition sd
lineend = "~r~n"

FOR li = 1 to UpperBound(cd_windef.ScriptList)
   sd = cd_windef.ScriptList[li]
   s = s + sd.Name + " " + lineend
NEXT
mle_1.Text = s
```

This example amplifies on the previous one and accesses various properties in the ScriptDefinition object. It reports whether the script is a function or event, whether it is scripted locally, what its return datatype and arguments are, and how the arguments are passed:

```
string s, lineend
integer li, lis, li_bound
ScriptDefinition sd
lineend = "~r~n"
FOR li = 1 to UpperBound(cd_windef.ScriptList)
   sd = cd_windef.ScriptList[li]
   s = s + sd.Name + " "

   CHOOSE CASE sd.Kind
   CASE ScriptEvent!
      // Events have three relevant properties
      // regarding where code is defined
      s = s + "Event, "
```

```
       IF sd.IsScripted = TRUE then
         s = s + "scripted, "
      END If
      IF sd.IsLocallyScripted = TRUE THEN
         s = s + "local, "
      END IF
      IF sd.IsLocallyDefined = TRUE THEN
         s = s + "local def,"
      END IF

   CASE ScriptFunction!
      // Functions have one relevant property
      // regarding where code is defined
      s = s + "Function, "
      IF sd.IsLocallyScripted = TRUE THEN
         s = s + "local, "
      END IF
   END CHOOSE

   s = s + "returns " + &
       sd.ReturnType.DataTypeOf + "; "
   s = s + "Args: "

   li_bound = UpperBound(sd.ArgumentList)
   IF li_bound = 0 THEN s = s + "None"

   FOR lis = 1 to li_bound
   CHOOSE CASE sd.ArgumentList[lis]. &
      CallingConvention
      CASE ByReferenceArgument!
      s = s + "REF "
      CASE ByValueArgument!
      s = s + "VAL "
      CASE ReadOnlyArgument!
      s = s + "READONLY "
      CASE ELSE
      s = s + "BUILTIN "
   END CHOOSE

   s = s + sd.ArgumentList[lis].Name + ", "
   NEXT

   s = s + lineend
NEXT
mle_1.text = s
```

Where the code is in the inheritance hierarchy You can check the IsLocallyScripted property to find out whether a script has code at the class's own level in the inheritance hierarchy. By walking back up the inheritance hierarchy using the Ancestor property, you can find out where the code is for a script.

This example looks at the scripts for the class associated with the ClassDefinition cd_windef, and if a script's code is defined at this level, the script's name is added to a drop-down list. It also saves the script's position in the ScriptList array in the instance variable *ii_localscript_idx*. The DropDownListBox is not sorted, so the positions in the list and the array stay in sync:

```
integer li_pos, li

FOR li = 1 to UpperBound(cd_windef.ScriptList)
   IF cd_windef.ScriptList[li].IsLocallyScripted &
      = TRUE
   THEN
      li_pos = ddlb_localscripts.AddItem( &
         cd_windef.ScriptList[li].Name)
      ii_localscript_idx[li_pos] = li
   END IF
NEXT
```

Matching function signatures

When a class has overloaded functions, you can call FindMatchingFunction to find out what function is called for a particular argument list.

For an example, see FindMatchingFunction in the *PowerScript Reference*.

Getting information about variables

This section has code fragments illustrating how to get information about variables from a ClassDefinition object called cd_windef. For examples of assigning a value to cd_windef, see "Getting a class definition object" on page 55.

List of variables Variables associated with a class are listed in the VariableList array of the ClassDefinition object. When you examine that array, you find not only variables you have defined explicitly but also PowerBuilder object properties and nested objects, which are instance variables.

This example loops through the VariableList array and builds a list of variable names. PowerBuilder properties appear first, followed by nested objects and your own instance and shared variables:

```
string s, lineend
integer li
VariableDefinition vard
lineend = "~r~n"

FOR li = 1 to UpperBound(cd_windef.VariableList)
   vard = cd_windef.VariableList[li]
   s = s + vard.Name + lineend
NEXT
mle_1.Text = s
```

Details about variables

This example looks at the properties of each variable in the VariableList array and reports its datatype, cardinality, and whether it is global, shared, or instance. It also checks whether an instance variable overrides an ancestor declaration:

```
string s
integer li
VariableDefinition vard
lineend = "~r~n"

FOR li = 1 to UpperBound(cd_windef.VariableList)
   vard = cd_windef.VariableList[li]
   s = s + vard.Name + ", "
   s = s + vard.TypeInfo.DataTypeOf

   CHOOSE CASE vard.Cardinality.Cardinality
   CASE ScalarType!
      s = s + ", scalar"
   CASE UnboundedArray!, BoundedArray!
      s = s + ", array"
   END CHOOSE

   CHOOSE CASE vard.Kind
   CASE VariableGlobal!
      s = s + ", global"
   CASE VariableShared!
      s = s + ", shared"
   CASE VariableInstance!
      s = s + ", instance"
      IF vard.OverridesAncestorValue = TRUE THEN
         s = s + ", override"
      END IF
   END CHOOSE
      s = s + lineend
NEXT
mle_1.text = s
```

PART 3

User Interface Techniques

This part presents a collection of techniques you can use to implement user interface features in the applications you develop with PowerBuilder. It includes building an MDI application, using drag and drop in a window, and providing online Help for an application.

CHAPTER 5 Building an MDI Application

About this chapter

This chapter describes how to build a Multiple Document Interface (MDI) application in PowerBuilder.

Contents

About MDI

Multiple Document Interface (MDI) is an application style you can use to open multiple windows (called **sheets**) in a single window and move among the sheets. To build an MDI application, you define a window whose type is MDI Frame and open other windows as sheets within the frame.

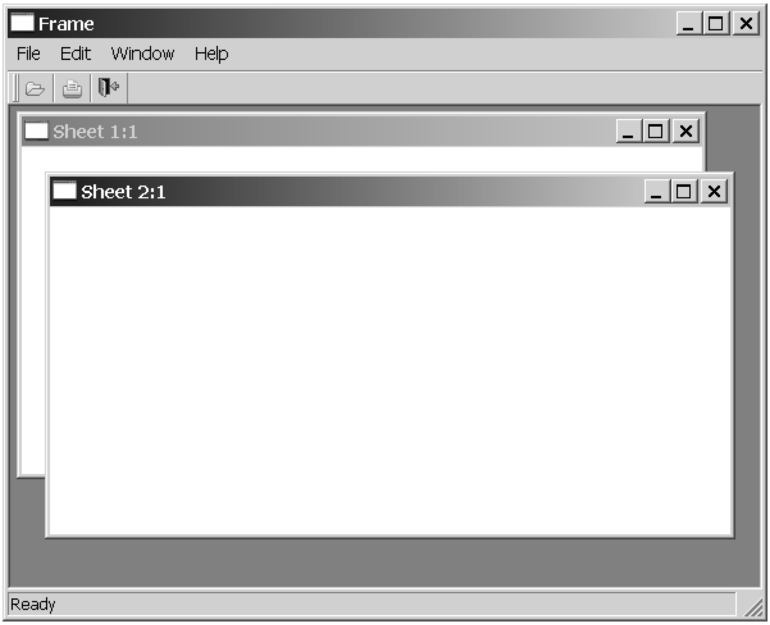
Most large-scale Windows applications are MDI applications. For example, PowerBuilder is an MDI application: the PowerBuilder window is the frame and the painters are the sheets.

If you expect your users to want to open several windows and easily move from window to window, you should make your application an MDI application.

Using the Template Application feature
When you create a new application, you can select the Template Application Start wizard and then choose to create an SDI or MDI application. If you select MDI application, PowerBuilder generates the shell of an MDI application that includes an MDI frame (complete with window functions that do such things as open or close a sheet), a sheet manager object and several sheets, an About dialog box, menus, toolbars, and scripts.

MDI frame windows

An MDI frame window is a window with several parts: a menu bar, a frame, a client area, sheets, and (usually) a status area, which can display **MicroHelp** (a short description of the current menu item or current activity).



The frame

The MDI frame is the outside area of the MDI window that contains the client area. There are two types of MDI frames:

- Standard
- Custom

Standard frames A standard MDI frame window has a menu bar and (usually) a status area for displaying MicroHelp. The client area is empty, except when sheets are open. Sheets can have their own menus, or they can inherit their menus from the MDI frame. Menu bars in MDI applications always display in the frame, never in a sheet. The menu bar typically has an item that lists all open sheets and lets the user tile, cascade, or layer the open sheets.

Custom frames Like a standard frame, a custom frame window usually has a menu bar and a status area. The difference between standard and custom frames is in the client area: in standard frames, the client area contains only open sheets; in custom frames, the client area contains the open sheets as well as other objects, such as buttons and StaticText. For example, you might want to add a set of buttons with some explanatory text in the client area.

Client area

In a standard frame window, PowerBuilder sizes the client area automatically and the open sheets display within the client area. In custom frame windows containing objects in the client area, you must size the client area yourself. If you do not size the client area, the sheets will open, but may not be visible.

The MDI_1 control When you build an MDI frame window, PowerBuilder creates a control named MDI_1, which it uses to identify the client area of the frame window. In standard frames, PowerBuilder manages MDI_1 automatically. In custom frames, you write a script for the frame's Resize event to size MDI_1 appropriately.

Displaying information about MDI_1
You can see the properties and functions for MDI_1 in the Browser. Create a window of type MDI and select the Window tab in the Browser. Select the MDI frame window and select Expand All from the pop-up menu. MDI_1 is listed as a window control, and you can examine its properties, functions, and so forth in the right pane of the Browser.

MDI sheets

Sheets are windows that can be opened in the client area of an MDI frame. You can use any type of window except an MDI frame as a sheet in an MDI application. To open a sheet, use either the OpenSheet or OpenSheetWithParm function.

Toolbars

Often you want to provide a toolbar for users of an MDI application. You can have PowerBuilder automatically create and manage a toolbar that is based on the current menu, or you can create your own custom toolbar (generally as a user object) and size the client area yourself.

For information on providing a toolbar, see the chapter on menus and toolbars in the *User's Guide*. For more information on sizing the client area, see "Sizing the client area" on page 77.

Building an MDI frame window

When you create a new window in PowerBuilder, its default window type is Main. Select mdi! or mdihelp! on the General property page to change the window to an MDI frame window.

Using menus

When you change the window type to MDI, you must associate a menu with the frame. Menus usually provide a way to open sheets in the frame and to close the frame if the user has closed all the sheets.

About menus and sheets
A sheet can have its own menu but is not required to. When a sheet without a menu is opened, it uses the frame's menu.

Using sheets

In an MDI frame window, users can open windows (sheets) to perform activities. For example, in an electronic mail application, an MDI frame might have sheets that users open to create and send messages and read and reply to messages. All sheets can be open at the same time and the user can move among the sheets, performing different activities in each sheet.

About menus and sheets
A sheet can have its own menu but is not required to. When a sheet without a menu is opened, it uses the frame's menu.

Opening sheets

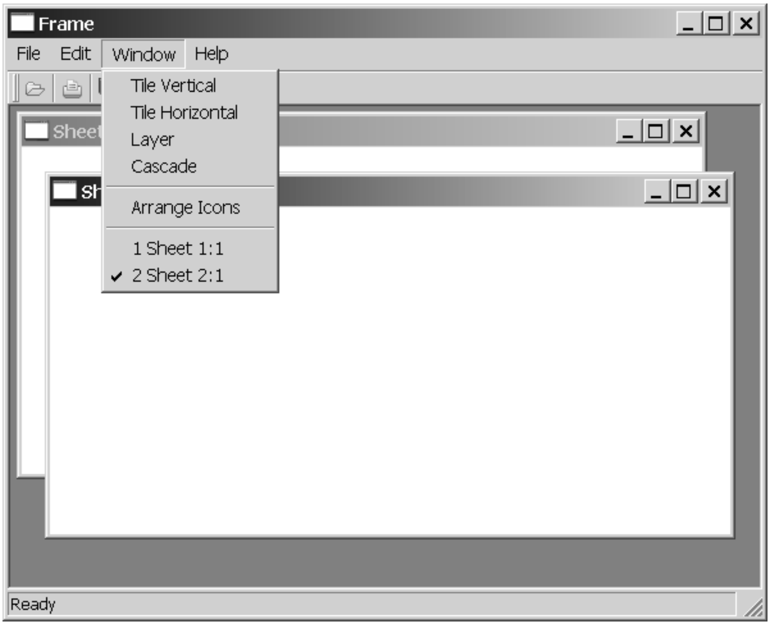
To open a sheet in the client area of an MDI frame, use the OpenSheet function in a script for an event in a menu item, an event in another sheet, or an event in any object in the frame.

For more information about OpenSheet, see the *PowerScript Reference*.

Opening instances of windows
Typically in an MDI application, you allow users to open more than one instance of a particular window type. For example, in an order entry application, users can probably look at several different orders at the same time. Each of these orders displays in a separate window (sheet).

Listing open sheets

When you open a sheet in the client area, you can display the title of the window (sheet) in a list at the end of a drop-down menu. This menu lists two open sheets:



❖ **To list open sheets in a drop-down menu:**

- Specify the number of the menu bar item in which you want the open sheets listed when you call the OpenSheet function. Typically you list the open sheets in the Windows menu. In a menu bar with four items in the order File, Edit, Windows, and Help, you specify the Windows menu with the number 3.

If more than nine sheets are open at one time, only nine sheets are listed in the menu and More Windows displays in the tenth position. To display the rest of the sheets in the list, click More Windows.

Arranging sheets

After you open sheets in an MDI frame, you can change the way they are arranged in the frame with the ArrangeSheets function.

To allow arrangement of sheets
To allow the user to arrange the sheets, create a menu item (typically on a menu bar item named Window) and use the ArrangeSheets function to arrange the sheets when the user selects a menu item.

Maximizing sheets

If sheets opened in an MDI window have a control menu, users can maximize the sheets. When the active sheet is maximized:

- If another sheet becomes the active sheet, that sheet is maximized (the sheet inherits the state of the previous sheet).
- If a new sheet is opened, the current sheet is restored to its previous size and the new sheet is opened in its original size.

Closing sheets

Active sheet To close the active window (sheet), users can press CTRL+F4. You can write a script for a menu item that closes the parent window of the menu (make sure the menu is associated with the sheet, not the frame.) For example:

```
Close(ParentWindow)
```

All sheets To close all sheets and exit the application, users can press ALT+F4. You can write a script to keep track of the open sheets in an array and then use a loop structure to close them.

Providing MicroHelp

MDI provides a MicroHelp facility that you can use to display information to the user in the status area at the bottom of the frame. For example, when the user selects a menu item, the MicroHelp facility displays a description of the selected item in the status area.

You can define MicroHelp for menu items and for controls in custom frame windows.

Providing MicroHelp for menu items

You specify the text for the MicroHelp associated with a menu item on the General property page in the Menu painter. To change the text of the MicroHelp in a script for a menu item, use the SetMicroHelp function.

Providing MicroHelp for controls

You can associate MicroHelp with a control in a custom frame window by using the control's Tag property. For example, say you have added a Print button to the client area. To display MicroHelp for the button, write a script for the button's GetFocus event that sets the Tag property to the desired text and then uses SetMicroHelp to display the text. For example:

```
cb_print.Tag="Prints information about current job"
w_genapp_frame.SetMicroHelp(This.Tag)
```

You can also set a control's Tag property in the control's property sheet.

In the LoseFocus event, you should restore the MicroHelp:

```
w_genapp_frame.SetMicroHelp("Ready")
```

Using toolbars in MDI applications

This section describes some techniques you can use to customize the behavior of your toolbars and save and restore toolbar settings. For information about how to define and use menus and toolbars, see the *User's Guide*.

Customizing toolbar behavior

Disabling toolbar buttons

To disable a toolbar button, you need to disable the menu item with which it is associated. Disabling the menu item disables the toolbar button automatically.

To disable a menu item, you need to set the Enabled property of the menu item:

```
m_test.m_file.m_new.Enabled = FALSE
```

Hiding toolbar buttons

To hide a menu item, you set the Visible property of the item:

```
m_test.m_file.m_open.Visible = FALSE
```

Hiding a menu item does not cause its toolbar button to disappear or be disabled. To hide a toolbar button, you need to set the ToolbarItemVisible property of the menu item:

```
m_test.m_file.m_open.ToolBarItemVisible = FALSE
```

Hiding a menu item does not cause the toolbar buttons for the drop-down or cascading menu items at the next level to disappear or be disabled. You need to hide or disable these buttons individually.

Setting the current item in a drop-down toolbar

When a user clicks on a toolbar button in a drop-down toolbar, PowerBuilder makes the selected button the current item. This makes it easy for the user to perform a particular toolbar action repeatedly. You can also make a particular button the current item programmatically by setting the CurrentItem property of the MenuCascade object. For example, to set the current item to the toolbar button for the New option on the File menu, you could execute this line in a script:

```
m_test.m_file.currentitem = m_test.m_file.m_new
```

In order for this to work, you would need to specify MenuCascade as the object type for the File menu in the Menu painter.

Testing for whether a toolbar is moved

Whenever a toolbar moves in an MDI frame window, PowerBuilder triggers the ToolBarMoved event for the window. In the script for this event, you can test to see which toolbar has moved and perform some processing. When the user moves the FrameBar or SheetBar, the ToolbarMoved event is triggered and the Message.WordParm and Message.LongParm properties are populated with values that indicate which toolbar was moved:

Table 5-1: Values of Message.WordParm and Message.LongParm properties

Property	Value	Meaning
Message.WordParm	0	FrameBar moved
	1	SheetBar moved
Message.LongParm	0	Moved to left
	1	Moved to top
	2	Moved to right
	3	Moved to bottom
	4	Set to floating

Saving and restoring toolbar settings

You can save and restore the current toolbar settings using functions that retrieve information about your toolbar settings, and you can modify these settings.

GetToolbar and GetToolbarPos allow you to retrieve the current toolbar settings. SetToolbar and SetToolbarPos allow you to change the toolbar settings. The syntax you use for the GetToolbarPos and SetToolbarPos functions varies depending on whether the toolbar you are working with is floating or docked.

Floating toolbars

The position of a floating toolbar is determined by its x and y coordinates. The size of a floating toolbar is determined by its width and height.

When you code the GetToolbarPos and SetToolbarPos functions for a floating toolbar, you need to include arguments for the x and y coordinates and the width and height.

Docked toolbars

The position of a docked toolbar is determined by its docking row and its offset from the beginning of the docking row. For toolbars at the top or bottom, the offset for a docked toolbar is measured from the left edge. For toolbars at the left or right, the offset is measured from the top.

By default, the docking row for a toolbar is the same as its bar index. If you align the toolbar with a different border in the window, its docking row may change depending on where you place it.

When you code the GetToolbarPos and SetToolbarPos functions for a docked toolbar, you need to include arguments for the docking row and the offset.

Example

The example below shows how to use a custom class user object to manage toolbar settings. The user object has two functions, one for saving the current settings and the other for restoring the settings later on. Because the logic required to save and restore the settings is handled in the user object (instead of in the window itself), this logic can easily be used with any window.

The sample code shown below supports both fixed and floating toolbars.

Script for the window’s Open event When the window opens, the following script restores the toolbar settings from an initialization file. To restore the settings, it creates a custom class user object called u_toolbar and calls the Restore function:

```
// Create the toolbar NVO
u_toolbar = create u_toolbar
// Restore the toolbar positions
u_toolbar.Restore(this,"toolbar.ini",   this.ClassName
())
```

Script for the window’s Close event When the window closes, the following script saves the toolbar settings by calling the Save function. Once the settings have been saved, it destroys the user object:

```
// Save the toolbar
stateu_toolbar.Save(this, "toolbar.ini", ClassName())
// Destroy the toolbar NVOdestroy u_toolbar
```

Script for the Save function The Save function has three arguments:

- *Win* – provides the window reference
- *File* – provides the name of the file where the settings should be saved
- *Section* – identifies the section where the settings should be saved

The Save function uses the GetToolbar and GetToolbarPos functions to retrieve the current toolbar settings. To write the settings to the initialization file, it uses the SetProfileString function.

The Save function can handle multiple toolbars for a single menu. It uses the bar index to keep track of information for each toolbar:

```
// Store the toolbar settings for the passed window
integer index, row, offset, x, y, w, h
boolean visible
string visstring, alignstring, title
toolbaralignment alignmentFOR index = 1 to 16

// Try to get the attributes for the bar.
IF win.GetToolbar(index, visible, alignment, &
   title)= 1 THEN
   // Convert visible to a string
   CHOOSE CASE visible
   CASE true
      visstring = "true"
   CASE false
      visstring = "false"
   END CHOOSE// Convert alignment to a string

   CHOOSE CASE alignment
   CASE AlignAtLeft!
      alignstring = "left"
   CASE AlignAtTop!
      alignstring = "top"
   CASE AlignAtRight!
      alignstring = "right"
   CASE AlignAtBottom!
      alignstring = "bottom"
   CASE Floating!
      alignstring = "floating"
   END CHOOSE

   // Save the basic attributes
   SetProfileString(file, section +  &
      String(index), "visible", visstring)
   SetProfileString(file, section +  &
      String(index), "alignment", alignstring)
   SetProfileString(file, section +  &
      String(index), "title", title)

   // Save the fixed position
   win.GetToolbarPos(index, row, offset)
```

```
      SetProfileString(file, section +  &
         String(index), "row", String(row))
      SetProfileString(file, section +  &
         String(index), "offset", String(offset))

      // Save the floating position
      win.GetToolbarPos(index, x, y, w, h)
      SetProfileString(file, section +  &
         String(index), "x", String(x))
      SetProfileString(file, section +  &
         String(index), "y", String(y))
      SetProfileString(file, section +  &
         String(index), "w", String(w))

      SetProfileString(file, section +  &
         String(index), "h", String(h))
      END IF
   NEXT
```

Script for the Restore function The Restore function has the same three arguments as the Save function. It uses the ProfileString function to retrieve toolbar settings from the initialization file. Once the settings have been retrieved, it uses the SetToolbar and SetToolbarPos functions to restore the toolbar settings.

Like the Save function, the Restore function can handle multiple toolbars for a single menu. It uses the bar index to keep track of information for each toolbar:

```
// Restore toolbar settings for the passed window

integer index, row, offset, x, y, w, h
boolean visible
string visstring, alignstring, title
toolbaralignment alignment

FOR index = 1 to 16
  // Try to get the attributes for the bar.
  IF win.GetToolbar(index, visible, alignment, &
    title)= 1 THEN
    // Try to get the attributes from the .ini
    // file
    visstring = ProfileString(file, section +  &
      String(index), "visible", "")
    IF visstring > "" THEN
      // Get all of the attributes
      alignstring = ProfileString(file, section +  &
        String(index), "alignment", "left")
```

```
            title = ProfileString(file, section +  &
              String(index), "title", "(Untitled)")
            row = Integer(ProfileString(file, section +  &
              String(index), "row", "1"))
            offset = Integer(ProfileString(file, &
              section + String(index), "offset", "0"))
            x = Integer(ProfileString(file, section +  &
              String(index), "x", "0"))
            y = Integer(ProfileString(file, section +  &
              String(index), "y", "0"))
            w = Integer(ProfileString(file, section +  &
              String(index), "w", "0"))
            h = Integer(ProfileString(file, section +  &
              String(index), "h", "0"))

            // Convert visstring to a boolean
            CHOOSE CASE visstring
            CASE "true"
              visible = true
            CASE "false"
              visible = false
            END CHOOSE

            // Convert alignstring to toolbaralignment
            CHOOSE CASE alignstring
            CASE "left"
              alignment = AlignAtLeft!
            CASE "top"
              alignment = AlignAtTop!
            CASE "right"
              alignment = AlignAtRight!
            CASE "bottom"
              alignment = AlignAtBottom!
            CASE "floating"
              alignment = Floating!
            END CHOOSE

            // Set the new position
            win.SetToolbar(index, visible, alignment, title)
            win.SetToolbarPos(index, row, offset, false)
            win.SetToolbarPos(index, x, y, w, h)
          END IF
        END IF
      NEXT
```

Sizing the client area

PowerBuilder sizes the client area in a standard MDI frame window automatically and displays open sheets unclipped within the client area. It also sizes the client area automatically if you have defined a toolbar based on menu items, as described in the preceding section.

However, in a custom MDI frame window—where the client area contains controls in addition to open sheets—PowerBuilder does not size the client area; you must size it. If you do not size the client area, the sheets open but may not be visible and are clipped if they exceed the size of the client area.

If you plan to use an MDI toolbar with a custom MDI frame, make sure the controls you place in the frame's client area are far enough away from the client area's borders so that the toolbar does not obscure them when displayed.

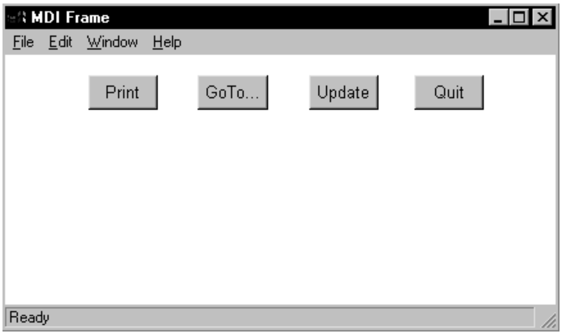
Scroll bars display when a sheet is clipped
If you selected HScrollBar and VScrollBar when defining the window, the scroll bars display when a sheet is clipped. This means not all the information in the sheet is displayed. The user can then scroll to display the information.

When you create a custom MDI frame window, PowerBuilder creates a control named MDI_1 to identify the client area of the frame. If you have enabled AutoScript, MDI_1 displays in the list of objects in the AutoScript pop-up window when you create a script for the frame.

❖ **To size or resize the client area when the frame is opened or resized:**

- Write a script for the frame's Open or Resize event that:
  - Determines the size of the frame
  - Sizes the client area (MDI_1) appropriately

For example, the following script sizes the client area for the frame w_genapp_frame. The frame has a series of buttons running across the frame just below the menu, and MicroHelp at the bottom:



```
int li_width, li_height

//Get the width and height of the frame's workspace.
//
//Note that if the frame displays any MDI toolbars,
//those toolbars take away from the size of the
//workspace as returned by the WorkSpaceWidth and
//WorkSpaceHeight functions. Later, you see how to
//to adjust for this.
//
li_width = w_genapp_frame.WorkSpaceWidth()

li_height = w_genapp_frame.WorkSpaceHeight()

//Next, determine the desired height of the client
//area by doing the following:
//
// 1) Subtract from the WorkSpaceHeight value: the
//    height of your control and the Y coordinate of
//    the control (which is the distance between the
//    top of the frame's workspace -- as if no
//    toolbars were there -- and the top of your
//    control).
//
// 2) Then subtract: the height of the frame's
//    MicroHelp bar (if present)
//
// 3) Then add back: the height of any toolbars that
//    are displayed (to adjust for the fact that the
//    original WorkSpaceHeight value we started with
//    is off by this amount). The total toolbar
//   height is equal to the Y coordinate returned
//   by the WorkspaceY function.

li_height = li_height - (cb_print.y + cb_print.height)

li_height = li_height - MDI_1.MicroHelpHeight

li_height = li_height + WorkspaceY()

//Now, move the client area to begin just below your
//control in the workspace.

mdi_1.Move (WorkspaceX (), cb_print.y +  &
   cb_print.height)

//Finally, resize the client area based on the width
//and height you calculated earlier.

mdi_1.Resize (li_width, li_height)
```

About MicroHelpHeight

MicroHelpHeight is a property of MDI_1 that PowerBuilder sets when you select a window type for your MDI window. If you select MDI Frame, there is no MicroHelp and MicroHelpHeight is 0; if you select MDI Frame with MicroHelp, MicroHelpHeight is the height of the MicroHelp.

About keyboard support in MDI applications

PowerBuilder MDI applications automatically support arrow keys and shortcut keys.

Arrow keys

In an MDI frame, how the pointer moves when the user presses an arrow key depends on where focus is when the key is pressed:

Table 5-2: Arrow key focus changes

Key	If focus is in	Focus moves to
Left	The menu bar	The menu item to the left of the item that has focus
	The first menu bar item	The control menu of the active sheet
	The control menu of the active sheet	The control menu of the frame
	The control menu of the frame	The last menu item
Right	The menu bar	The menu item to the right of the item that has focus
	The last menu bar item	The control menu of the frame
	The control menu of the frame	The control menu of the active sheet
	The control menu of the active sheet	The first item in the menu bar
Down	A drop-down or cascading menu	The menu item below the current item
	The last menu item in the drop-down or cascading menu	The first item in the menu
Up	A drop-down or cascading menu	The menu item above the current item
	The first menu item in a drop-down or cascading menu	The last item in the menu

Shortcut keys

PowerBuilder automatically assigns two shortcut keys to every MDI frame:

Table 5-3: MDI frame shortcut keys

Key	Use to
Ctrl+F4	Close the active sheet and make the previous sheet active. The previous sheet is the sheet that was active immediately before the sheet that was closed.
Ctrl+F6	Make the previous sheet the active sheet.

CHAPTER 6 Managing Window Instances

About this chapter

This chapter describes how to manage several instances of the same window.

Contents

About window instances

When you build an application, you may want to display several windows that are identical in structure but have different data values.

For example, you may have a w_employee window and want to display information for two or more employees at the same time by opening multiple copies (instances) of the w_employee window.

You can do that, but you need to understand how PowerBuilder stores window definitions.

How PowerBuilder stores window definitions

When you save a window, PowerBuilder actually generates two entities in the library:

- **A new datatype** The name of the datatype is the same as the name of the window.

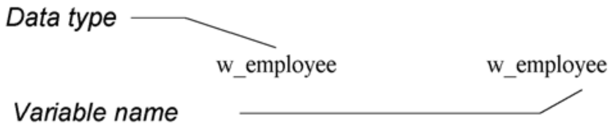
  For example, when you save a window named w_employee, PowerBuilder internally creates a datatype named w_employee.

- **A new global variable of the new datatype** The name of the global variable is the same as the name of the window.

  For example, when you save the w_employee window, you are also implicitly defining a global variable named *w_employee* of type w_employee.

It is as if you had made the following declaration:

Figure 6-1: Variable declaration



By duplicating the name of the datatype and variable, PowerBuilder allows new users to access windows easily through their variables while ignoring the concept of datatype.

What happens when you open a window

To open a window, you use the Open function, such as:

```
Open(w_employee)
```

This actually creates an instance of the datatype w_employee and assigns it a reference to the global variable, also named *w_employee*.

As you have probably noticed, when you open a window that is already open, PowerBuilder simply activates the existing window; it does not open a new window. For example, consider this script for a CommandButton's Clicked event:

```
Open(w_employee)
```

No matter how many times this button is clicked, there is still only one window w_employee. It is pointed to by the global variable *w_employee*.

To open multiple instances of a window, you declare variables of the window's type.

Declaring instances of windows

Because a window is actually a datatype, you can declare variables of that datatype, just as you can declare integers, strings, and so on. You can then refer to those variables in code.

For example:

```
w_employee mywin
```

declares a variable named *mywin* of type w_employee.

Limitation of using variables
When you declare a window instance, you cannot reference it from another window. For example, if there are three windows open, you cannot explicitly refer to the first one from the second or third. There is no global handle for windows opened using reference variables. To maintain references to window instances using a script, see "Using window arrays" on page 84.

Opening an instance

To open a window instance, you refer to the window variable in the Open function:

```
w_employee mywin
Open(mywin)
```

Here the Open function determines that the datatype of the variable *mywin* is w_employee. It then creates an instance of w_employee and assigns a reference to the *mywin* variable.

If you code the above script for the Clicked event for a CommandButton, each time the button is clicked, a new instance of w_employee is created. In other words, a new window is opened each time the button is clicked.

By creating variables whose datatype is the name of a window, you can open multiple instances of a window. This is easy and straightforward. PowerBuilder manages the windows for you—for example, freeing memory when you close the windows.

Closing an instance

A common way to close the instances of a window is to put a CommandButton in the window with this script for the Clicked event:

```
Close(Parent)
```

This script closes the parent of the button (the window in which the button displays). Continuing the example above, if you put a CommandButton in w_employee, the script closes the current instance of w_employee. If you click the CommandButton in the *mywin* instance of w_employee, *mywin* closes.

Using window arrays

To create an array of windows, declare an array of the datatype of the window. For example, the following statement declares an array named myarray, which contains five instances of the window w_employee:

```
w_employee myarray[5]
```

You can also create unbounded arrays of windows if the number of windows to be opened is not known at compile time.

Opening an instance using an array

To open an instance of a window in an array, use the Open function and pass it the array index. Continuing the example above, the following statements open the first and second instances of the window w_employee:

```
Open(myarray[1])          // Opens the first instance
                          // of the window w_employee.
Open(myarray[2])          // Opens the second instance.
```

Moving first instance opened
The statements in this example open the second instance of the window at the same screen location as the first instance. Therefore, you should call the Move function in the script to relocate the first instance before the second Open function call.

Manipulating arrays

Using arrays of windows, you can manipulate particular instances by using the array index. For example, the following statement hides the second window in the array:

```
myarray[2].Hide()
```

You can also reference controls in windows by using the array index, such as:

```
myarray[2].st_count.text = "2"
```

Opening many windows
When you open or close a large number of instances of a window, you may want to use a FOR...NEXT control structure in the main window to open or close the instances. For example:

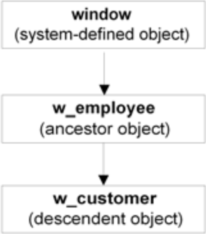
```
w_employee myarray[5]
for i = 1 to 5
   Open(myarray[i])
next
```

Creating mixed arrays

In the previous example, all windows in the array are the same type. You can also create arrays of mixed type. Before you can understand this technique, you need to know one more thing about window inheritance: all windows you define are actually descendants of the built-in datatype window.

Suppose you have a window w_employee that is defined from scratch, and w_customer that inherits from w_employee. The complete inheritance hierarchy is the following:

Figure 6-2: Window inheritance hierarchy



The system-defined object named window is the ancestor of all windows you define in PowerBuilder. The built-in object named window defines properties that are used in all windows (such as X, Y, and Title).

If you declare a variable of type window, you can reference any type of window in the application. This is because all user-defined windows are a kind of window.

The following code creates an array of three windows. The array is named newarray. The array can reference any type of window, because all user-defined windows are derived from the window datatype:

```
window newarray[3]
string win[3]
int iwin[1] = "w_employee"
win[2] = "w_customer"
win[3] = "w_sales"

for i = 1 to 3
   Open(newarray[i], win[i])
next
```

The code uses this form of the Open function:

Open (*windowVariable, windowType*)

where *windowVariable* is a variable of type window (or a descendant of window) and *windowType* is a string that specifies the type of window.

The preceding code opens three windows: an instance of w_employee, an instance of w_customer, and an instance of w_sales.

Using arrays versus reference variables

Table 6-1 shows when you use reference variables and when you use arrays to manipulate window instances.

Table 6-1: Arrays as opposed to reference variables

Item	Advantages	Disadvantages
Arrays	You can refer to particular instances.	Arrays are more difficult to use. For example, if the user closes the second window in an array, then wants to open a new window, your code must determine whether to add a window to the end of the array (thereby using more memory than needed) or find an empty slot in the existing array for the new window.
Reference variables	Easy to use—PowerBuilder manages them automatically.	You cannot manipulate particular instances of windows created using reference variables.

Suppose you use w_employee to provide or modify data for individual employees. You may want to prevent a second instance of w_employee opening for the same employee, or to determine for which employees an instance of w_employee is open. To do this kind of management, you must use an array. If you do not need to manage specific window instances, use reference variables instead to take advantage of their ease of use.

Referencing entities in descendants

When you declare a variable whose datatype is a kind of object, such as a window, you can use the variable to reference any entity defined in the object, but not in one of its descendants. Consider the following code:

```
w_customer mycust

Open(mycust)
// The following statement is legal if
// w_customer window has a st_name control.
mycust.st_name.text = "Joe"
```

mycust is declared as a variable of type w_customer (*mycust* is a w_customer window). If w_customer contains a StaticText control named st_name, then the last statement shown above is legal.

However, consider the following case:

```
window newwin
string winname = "w_customer"
Open(newwin, winname)
// Illegal because objects of type Window
// do not have a StaticText control st_name
newwin.st_name.text = "Joe"
```

Here, *newwin* is defined as a variable of type window. PowerBuilder rejects the above code because the compiler uses what is called **strong type checking**: the PowerBuilder compiler does not allow you to reference any entity for an object that is not explicitly part of the variable's compile-time datatype.

Because objects of type window do not contain a st_name control, the statement is not allowed. You would need to do one of the following:

- Change the declaration of *newwin* to be a w_customer (or an ancestor window that also contains a st_name control), such as:

  ```
  w_customer newwin
  string winname = "w_customer"

  Open(newwin, winname)
  // Legal now
  newwin.st_name.text = "Joe"
  ```

- Define another variable, of type w_customer, and assign it to *newwin*, such as:

  ```
  window newwin
  w_customer custwin
  stringwinname = "w_customer"

  Open(newwin, winname)
  custwin = newwin
  // Legal now
  custwin.st_name.text = "Joe"
  ```

CHAPTER 7

Using Tab Controls in a Window

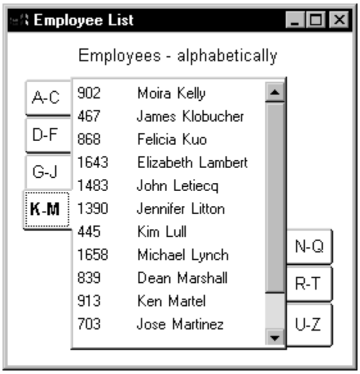
About this chapter

This chapter describes how to use Tab controls in your application.

Contents

About Tab controls

A Tab control is a container for tab pages that display other controls. One page at a time fills the display area of the Tab control. Each page has a tab like an index card divider. The user can click the tab to switch among the pages:



902	Moira Kelly
467	James Klobucher
868	Felicia Kuo
1643	Elizabeth Lambert
1483	John Letiecq
1390	Jennifer Litton
445	Kim Lull
1658	Michael Lynch
839	Dean Marshall
913	Ken Martel
703	Jose Martinez

The Tab control allows you to present many pieces of information in an organized way. You add, resize, and move Tab controls just as you do any control. The *PowerBuilder User’s Guide* describes how to add controls to a window or custom visual user object.

Tab terms

You need to know these definitions:

Tab control A control that you place in a window or user object that contains tab pages. Part of the area in the Tab control is for the tabs associated with the tab pages. Any space that is left is occupied by the tab pages themselves.

Tab page A user object that contains other controls and is one of several pages within a Tab control. All the tab pages in a Tab control occupy the same area of the control and only one is visible at a time. The active tab page covers the other tab pages.

You can define tab pages right in the Tab control or you can define them in the User Object painter and insert them into the Tab control, either in the painter or during execution.

Tab The visual handle for a tab page. The tab displays a label for the tab page. When a tab page is hidden, the user clicks its tab to bring it to the front and make the tab page active.

Defining and managing tab pages

A tab page is a user object.

Two methods

There are different ways to approach tab page definition. You can define:

- **An embedded tab page** In the painter, insert tab pages in the Tab control and add controls to those pages. An embedded tab page is of class UserObject, but is not reusable.
- **An independent user object** In the User Object painter, create a custom visual user object and add the controls that will display on the tab page. You can use the user object as a tab page in a Tab control, either in the painter or by calling OpenTab in a script. A tab page defined as an independent user object is reusable.

You can mix and match the two methods—one Tab control can contain both embedded tab pages and independent user objects.

Creating tab pages

When you create a new Tab control, it has one embedded tab page. You can use that tab page or you can delete it.

❖ **To create a new tab page within the Tab control:**

1 Right-click in the tab area of the Tab control. Do not click a tab page.

2 Select Insert TabPage from the pop-up menu.

3 Add controls to the new page.

❖ **To define a tab page independent of a Tab control:**

1 Select Custom Visual on the Object tab in the New dialog box.

2 In the User Object painter, size the user object to match the size of the display area of the Tab control in which you will use it.

3 Add the controls that will appear on the tab page to the user object and write scripts for their events.

4 On the user object's property sheet, click the TabPage tab and fill in information to be used by the tab page.

❖ **To add a tab page that exists as an independent user object to a Tab control:**

1 Right-click in the tab area of the Tab control. Do not click a tab page.

2 Select Insert User Object from the pop-up menu.

3 Select a user object.

The tab page is inherited from the user object you select. You can set tab page properties and write scripts for the inherited user object just as you do for tab pages defined within the Tab control.

Editing the controls on the tab page user object
You cannot edit the content of the user object within the Tab control. If you want to edit or write scripts for the controls, close the window or user object containing the Tab control and go back to the User Object painter to make changes.

Managing tab pages

You can view, reorder, and delete the tab pages on a Tab control.

❖ **To view a different tab page:**

- Click the page's tab.

The tab page comes to the front and becomes the active tab page. The tabs are rearranged according to the Tab position setting you have chosen.

❖ **To reorder the tabs within a Tab control:**

1 Click the Page Order tab on the Tab control's property sheet.

2 Drag the names of the tab pages to the desired order.

❖ **To delete a tab page from a Tab control:**

1 Click the page’s tab.

2 Right-click the tab page and select Cut or Clear from the pop-up menu.

Selecting tab controls and tab pages
As you click on various areas within a tab control, you will notice the Properties view changing to show the properties of the tab control itself, one of the tab pages, or a control on a tab page. Before you select an item such as Cut from the pop-up menu, make sure that you have selected the right object.

Clicking anywhere in the tab area of a tab control selects the tab control. When you click the tab for a specific page, that tab page becomes active, but the selected object is still the tab control. To select the tab page, click its tab to make it active and then click anywhere on the background of the page except on the tab itself.

Controls on tab pages

The real purpose of a Tab control is to display other controls on its pages. You can think of the tab page as a miniature window. You add controls to it just as you do to a window.

When you are working on a Tab control, you can add controls only to a tab page created within the Tab control.

Adding controls to an independent user object tab page
To add controls to an independent user object tab page, open it in the User Object painter.

❖ **To add a control to an embedded tab page:**

- Choose a control from the toolbar or the Insert menu and click the tab page, just as you do to add a control to a window.

  When you click inside the tab page, the tab page becomes the control’s parent.

❖ **To move a control from one tab page to another:**

- Cut or copy the control and paste it on the destination tab page.

The source and destination tab pages must both be embedded tab pages, not independent user objects.

❖ **To move a control between a tab page and the window containing the Tab control:**

- Cut or copy the control and paste it on the destination window or tab page.

  You cannot drag the control out of the Tab control onto the window.

Moving the control between a tab page and the window changes the control's parent, which affects scripts that refer to the control.

Customizing the Tab control

The Tab control has settings for controlling the position and appearance of the tabs. Each tab can have its own label, picture, and background color.

All tabs share the same font settings, which you set on the Tab control's Font property page.

Pop-up menus and property sheets for Tab controls and tab pages

A Tab control has several elements, each with its own pop-up menu and property sheet. To open the property sheet, double-click or select Properties on the pop-up menu.

Where you click determines what element you access.

Table 7-1: Accessing Tab control elements

To access the pop-up menu or property sheet for a	Do this
Tab control	Right-click or double-click in the tab area of the Tab control.
Tab page	Click the tab to make the tab page active, then right-click or double-click somewhere in the tab page but not on a control on the page.
Control on a tab page	Click the tab to make the tab page active and right-click or double-click the control.

Position and size of tabs

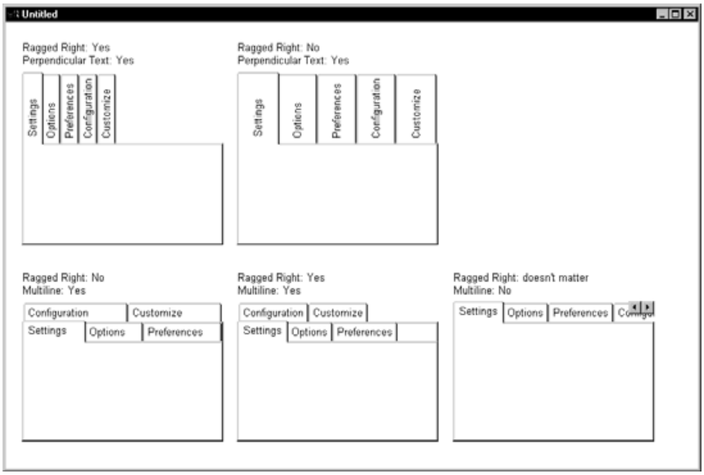
The General tab in the Tab control’s property sheet has several settings for controlling the position and size of the tabs.

Table 7-2: Controlling size and position of tabs

To change	Change the value for
The side(s) of the Tab control on which the tabs appear	Tab Position
The size of the tabs relative to the size of the Tab control	Ragged Right, MultiLine, Fixed Width
The orientation of the text relative to the side of the Tab control (use this setting with caution—only TrueType fonts support perpendicular text)	Perpendicular Text

Fixed Width and Ragged Right
When Fixed Width is checked, the tabs are all the same size. This is different from turning Ragged Right off, which stretches the tabs to fill the edge of the Tab control, like justified text. The effect is the same if all the tab labels are short, but if you have a mix of long and short labels, justified labels can be different sizes unless Fixed Width is on.

This figure illustrates the effect of combining some of these settings. Tab Position is Top:



Ragged Right: Yes
Perpendicular Text: Yes

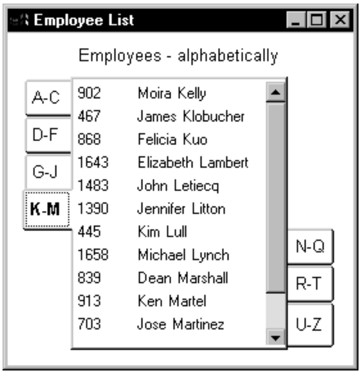
Ragged Right: No
Perpendicular Text: Yes

Ragged Right: No
Multiline: Yes

Ragged Right: Yes
Multiline: Yes

Ragged Right: doesn't matter
Multiline: No

This sample Tab control is set up like an address book. It has tabs that flip between the left and right sides. With the Bold Selected Text setting on and the changing tab positions, it is easy to see which tab is selected:



Employees - alphabetically

902	Moira Kelly
467	James Klobucher
868	Felicia Kuo
1643	Elizabeth Lambert
1483	John Letiecq
1390	Jennifer Litton
445	Kim Lull
1658	Michael Lynch
839	Dean Marshall
913	Ken Martel
703	Jose Martinez

Tab labels

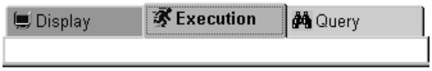
You can change the appearance of the tab using the property sheets of both the Tab control and the Tab page.

Table 7-3: Changing the appearance of a tab

Property sheet	Property page	Setting	Affects
Tab control	General	PictureOnRight, ShowPicture, ShowText	All tabs in the control
Tab page	General	Text, BackColor	The label on the tab and the background color of the tab page
Tab page	TabPage	PictureName, TabTextColor, TabBackColor, PictureMaskColor	The color of the text and picture on the tab and the background color of the tab itself (not the tab page)

If you are working in the User Object painter on an object you will use as a tab page, you can make the same settings on the TabPage page of the user object's property sheet that you can make in the tab page's property sheet.

This example has a picture and text assigned to each tab page. Each tab has a different background color. The Show Picture and Show Text settings are both on:



Changing tab appearance in scripts

All these settings in the painter have equivalent properties that you can set in a script, allowing you to change the appearance of the Tab control dynamically during execution.

Using Tab controls in scripts

This section provides examples of tabs in scripts:

- Referring to tab pages in scripts
- Referring to controls on tab pages
- Opening, closing, and hiding tab pages
- Keeping track of tab pages
- Creating tab pages only when needed
- Events for the parts of the Tab control

Referring to tab pages in scripts

Dot notation allows you to refer to individual tab pages and controls on those tab pages:

- The window or user object containing the Tab control is its parent:

  window.*tabcontrol*

- The Tab control is the parent of the tab pages contained in it:

  window.*tabcontrol*.*tabpageuo*

- The tab page is the parent of the control contained in it:

  window.*tabcontrol*.*tabpageuo*.*controlonpage*

For example, this statement refers to the PowerTips property of the Tab control tab_1 within the window w_display:

```
w_display.tab_1.PowerTips = TRUE
```

This example sets the PowerTipText property of tab page tabpage_1:

```
w_display.tab_1.tabpage_1.PowerTipText = &
   "Font settings"
```

This example enables the CommandButton cb_OK on the tab page tabpage_doit:

```
w_display.tab_1.tabpage_doit.cb_OK.Enabled = TRUE
```

Generic coding

You can use the Parent pronoun and GetParent function to make a script more general.

Parent pronoun In a script for any tab page, you can use the Parent pronoun to refer to the Tab control:

```
Parent.SelectTab(This)
```

GetParent function If you are in an event script for a tab page, you can call the GetParent function to get a reference to the tab page's parent, which is the Tab control, and assign the reference to a variable of type Tab.

In an event script for a user object that is used as a tab page, you can use code like the following to save a reference to the parent Tab control in an instance variable.

This is the declaration of the instance variable. It can hold a reference to any Tab control:

```
tab itab_settings
```

This code saves a reference to the tab page's parent in the instance variable:

```
// Get a reference to the Tab control
// "This" refers to the tab page user object
itab_settings = This.GetParent()
```

In event scripts for controls on the tab page, you can use GetParent twice to refer to the tab page user object and its Tab control:

```
tab tab_mytab
userobject tabpage_generic

tabpage_generic = This.GetParent()
tab_mytab = tabpage_generic.GetParent()

tabpage_generic.PowerTipText = &
   "Important property page"
tab_mytab.PowerTips = TRUE

tab_mytab.SelectTab(tabpage_generic)
```

Generic variables for controls have limitations The type of these variables is the basic PowerBuilder object type—a variable of type Tab has no knowledge of the tab pages in a specific Tab control and a variable of type UserObject has no knowledge of the controls on the tab page.

In this script for a tab page event, a local variable is assigned a reference to the parent Tab control. You cannot refer to specific pages in the Tab control because *tab_settings* does not know about them. You can call Tab control functions and refer to Tab control properties:

```
tab tab_settings
tab_settings = This.GetParent()
tab_settings.SelectTab(This)
```

User object variables If the tab page is an independent user object, you can define a variable whose type is that specific user object. You can now refer to controls defined on the user object, which is the ancestor of the tab page in the control.

In this script for a Tab control's event, the index argument refers to a tab page and is used to get a reference to a user object from the Control property array. The example assumes that all the tab pages are derived from the same user object uo_emprpt_page:

```
uo_emprpt_page tabpage_current
tabpage_current = This.Control[index]
tabpage_current.dw_emp.Retrieve &
   (tabpage_current.st_name.Text)
```

The Tab control's Control property
The Control property array contains references to all the tab pages in the control, including both embedded and independent user objects. New tab pages are added to the array when you insert them in the painter and when you open them in a script.

Referring to controls on tab pages

If you are referring to a control on a tab page in another window, you must fully qualify the control's name up to the window level.

The following example shows a fully qualified reference to a static text control:

```
w_activity_manager.tab_fyi.tabpage_today. &
   st_currlogon_time.Text = ls_current_logon_time
```

This example from the PowerBuilder Code Examples sets the size of a DataWindow control on the tab page to match the size of another DataWindow control in the window. Because all the tab pages were inserted in the painter, the Control property array corresponds with the tab page index. All the pages are based on the same user object u_tab_dir:

```
u_tab_dir luo_Tab
luo_Tab = This.Control[newindex]
luo_Tab.dw_dir.Height = dw_list.Height
luo_Tab.dw_dir.Width = dw_list.Width
```

In scripts and functions for the tab page user object, the user object knows about its own controls. You do not need to qualify references to the controls. This example in a function for the u_tab_dir user object retrieves data for the dw_dir DataWindow control:

```
IF NOT ib_Retrieved THEN
   dw_dir.SetTransObject(SQLCA)
   dw_dir.Retrieve(as_Parm)
   ib_Retrieved = TRUE
END IF

RETURN dw_dir.RowCount()
```

Opening, closing, and hiding tab pages

You can open tab pages in a script. You can close tab pages that you opened, but you cannot close tab pages that were inserted in the painter. You can hide any tab page.

This example opens a tab page of type tabpage_listbox and stores the object reference in an instance variable *i_tabpage*. The value 0 specifies that the tab page becomes the last page in the Tab control. You need to save the reference for closing the tab later.

This is the instance variable declaration for the tab page's object reference:

```
userobject i_tabpage
```

This code opens the tab page:

```
li_rtn = tab_1.OpenTab &
   (i_tabpage, "tabpage_listbox", 0)
```

This statement closes the tab page:

```
tab_1.CloseTab(i_tabpage)
```

Keeping track of tab pages

To refer to the controls on a tab page, you need the user object reference, not just the index of the tab page. You can use the tab page's Control property array to get references to all your tab pages.

Control property for tab pages

The Control property of the Tab control is an array with a reference to each tab page defined in the painter and each tab page added in a script. The index values that are passed to events match the array elements of the Control property.

You can get an object reference for the selected tab using the SelectedTab property:

```
userobject luo_tabpage
luo_tabpage = tab_1.Control[tab_1.SelectedTab]
```

In an event for the Tab control, like SelectionChanged, you can use the index value passed to the event to get a reference from the Control property array:

```
userobject tabpage_generic
tabpage_generic = This.Control[newindex]
```

Adding a new tab page

When you call OpenTab, the control property array grows by one element. The new element is a reference to the newly opened tab page. For example, the following statement adds a new tab in the second position in the Tab control:

```
tab_1.OpenTab(uo_newtab, 2)
```

The second element in the control array for tab_1 now refers to uo_newtab, and the index into the control array for all subsequent tab pages becomes one greater.

Closing a tab page

When you call CloseTab, the size of the array is reduced by one and the reference to the user object or page is destroyed. If the closed tab was not the last element in the array, the index for all subsequent tab pages is reduced by one.

Moving a tab page

The MoveTab function changes the order of the pages in a Tab control and also reorders the elements in the control array to match the new tab order.

Control property array for user objects
The Control property array for controls in a user object works in the same way.

Creating tab pages only when needed

The user might never look at all the tab pages in your Tab control. You can avoid the overhead of creating graphical representations of the controls on all the tab pages by checking Create on Demand on the Tab control's General property page or setting the CreateOnDemand property to TRUE.

The controls on all the tab pages in a Tab control are always instantiated when the Tab control is created. However, when Create on Demand is checked, the Constructor event for controls on tab pages is not triggered and graphical representations of the controls are not created until the user views the tab page.

Constructor events on the selected tab page
Constructor events for controls on the *selected* tab page are always triggered when the Tab control is created.

Tradeoffs for Create on Demand

A window will open more quickly if the creation of graphical representations is delayed for tab pages with many controls. However, scripts cannot refer to a control on a tab page until the control's Constructor event has run and a graphical representation of the control has been created. When Create on Demand is checked, scripts cannot reference controls on tab pages that the user has not viewed.

Whether a tab page has been created

You can check whether a tab page has been created with the PageCreated function. Then, if it has not been created, you can trigger the constructor event for the tab page using the CreatePage function:

```
IF tab_1.tabpage_3.PageCreated() = FALSE THEN
      tab_1.tabpage_3.CreatePage()
END IF
```

You can check whether a control on a tab page has been created by checking whether the control's handle is nonzero. If so, the control has been created.

```
IF Handle(tab_1.tabpage_3.dw_list) > 0 THEN ...
```

Changing CreateOnDemand during execution

If you change the CreateOnDemand property to FALSE in a script, graphical representations of any tab pages that have not been created are created immediately.

It does not do any good to change CreateOnDemand to TRUE during execution, because graphical representations of all the tab pages have already been created.

Creating tab pages dynamically

If CreateOnDemand is FALSE, you can set the label for a dynamically created tab page in its Constructor event, using the argument to OpenTabWithParm that is passed to the Message object. If CreateOnDemand is TRUE, you need to set the label when the tab page is instantiated, because the Constructor event is not triggered until the tab is selected. The following script in a user event that is posted from a window's open event opens five tab pages and sets the label for each tab as it is instantiated:

```
int li_ctr
string is_title
THIS.setredraw(false)

FOR li_ctr = 1 to 5
   is_title = "Tab#" + string(li_ctr)
   tab_test.opentabwithparm(iuo_tabpage[li_ctr], &
      is_title, 0)
iuo_tabpage[li_ctr].text = is_title //set tab label
NEXT

THIS.setredraw(true)
RETURN 1
```

Events for the parts of the Tab control

With so many overlapping pieces in a Tab control, you need to know where to code scripts for events.

Table 7-4: Coding scripts for Tab control events

To respond to actions in the	Write a script for events belonging to
Tab area of the Tab control, including clicks or drag actions on tabs	The Tab control
Tab page (but not the tab)	The tab page (for embedded tab pages) or the user object (for independent tab pages)
Control on a tab page	That control

For example, if the user drags to a tab and you want to do something to the tab page associated with the tab, you need to code the DragDrop event for the Tab control, not the tab page.

Examples

This code in the DragDrop event of the tab_1 control selects the tab page when the user drops something onto its tab. The index of the tab that is the drop target is an argument for the DragDrop event:

```
This.SelectTab( index )
```

The following code in the DragDrop event for the Tab control lets the user drag DataWindow information to a tab and then inserts the dragged information in a list on the tab page associated with the tab.

A user object of type tabpage_listbox that contains a ListBox control, lb_list, has been defined in the User Object painter. The Tab control contains several independent tab pages of type tabpage_listbox.

You can use the index argument for the DragDrop event to get a tab page reference from the Tab control's Control property array. The user object reference lets the script access the controls on the tab page.

The Parent pronoun in this script for the Tab control refers to the window:

```
long ll_row
string ls_name
tabpage_listbox luo_tabpage

IF TypeOf(source) = DataWindow! THEN
   l_row = Parent.dw_2.GetRow()
   ls_name = Parent.dw_2.Object.lname.Primary[ll_row]

    // Get a reference from the Control property array
   luo_tabpage = This.Control[index]

   // Make the tab page the selected tab page
   This.SelectTab(index)

   // Insert the dragged information
   luo_tabpage.lb_list.InsertItem(ls_name, 0)

END IF
```

If the tab page has not been created

If the CreateOnDemand property for the Tab control is TRUE, the Constructor events for a tab page and its controls are not triggered until the tab page is selected. In the previous example, making the tab page the selected tab page triggers the Constructor events. You could also use the CreatePage function to trigger them:

```
IF luo_tabpage.PageCreated() = FALSE THEN &
   luo_tabpage.CreatePage()
```

CHAPTER 8

Using TreeView Controls

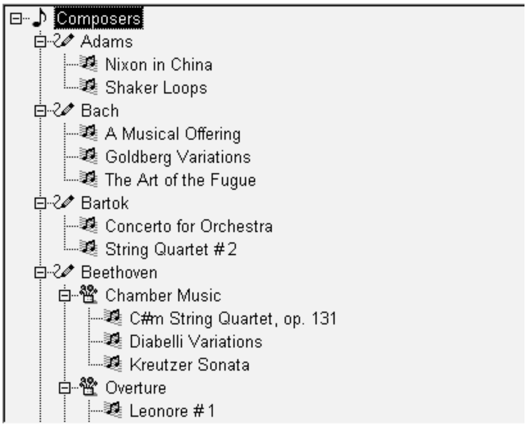
About this chapter

This chapter describes how to use TreeView controls to present hierarchical information in an expandable list.

Contents

About TreeView controls

TreeView controls provide a way to represent hierarchical relationships within a list. The TreeView provides a standard interface for expanding and collapsing branches of a hierarchy:



- Composers
  - Adams
    - Nixon in China
    - Shaker Loops
  - Bach
    - A Musical Offering
    - Goldberg Variations
    - The Art of the Fugue
  - Bartok
    - Concerto for Orchestra
    - String Quartet #2
  - Beethoven
    - Chamber Music
      - C#m String Quartet, op. 131
      - Diabelli Variations
      - Kreutzer Sonata
    - Overture
      - Leonore #1

When to use a TreeView

You use TreeViews in windows and custom visual user objects. Choose a TreeView instead of a ListBox or ListView when your information is more complex than a list of similar items and when levels of information have a one-to-many relationship. Choose a TreeView instead of a DataWindow control when your user will want to expand and collapse the list using the standard TreeView interface.

Hierarchy of items

Although items in a TreeView can be a single, flat list like the report view of a ListView, you tap the power of a TreeView when items have a one-to-many relationship two or more levels deep. For example, your list might have one or several parent categories with child items within each category. Or the list might have several levels of subcategories before getting to the end of a branch in the hierarchy:

```
Root
   Category 1
      Subcategory 1a
         Detail
         Detail
      Subcategory 1b
         Detail
         Detail
   Category 2
      Subcategory 2a
         Detail
```

Number of levels in each branch

You do not have to have the same number of levels in every branch of the hierarchy if your data requires more levels of categorization in some branches. However, programming for the TreeView is simpler if the items at a particular level are the same type of item, rather than subcategories in some branches and detail items in others.

For example, in scripts you might test the level of an item to determine appropriate actions. You can call the SetLevelPictures function to set pictures for all the items at a particular level.

Content sources for a TreeView

For most of the list types in PowerBuilder, you can add items in the painter or in a script, but for a TreeView, you have to write a script. Generally, you will populate the first level (the root level) of the TreeView when its window opens. When the user wants to view a branch, a script for the TreeView's ItemPopulate event can add items at the next levels.

The data for items can be hard-coded in the script, but it is more likely that you will use the user's own input or a database for the TreeView's content. Because of the one-to-many relationship of an item to its child items, you might use several tables in a database to populate the TreeView.

For an example using DataStores, see “Using DataWindow information to populate a TreeView” on page 125.

Pictures for items

Pictures are associated with individual items in a TreeView. You identify pictures you want to use in the control’s picture lists and then associate the index of the picture with an item. Generally, pictures are not unique for each item. Pictures provide a way to categorize or mark items within a level. To help the user understand the data, you might:

- Use a different picture for each level
- Use several pictures within a level to identify different types of items
- Use pictures on some levels only
- Change the picture after the user clicks on an item

Pictures are not required You do not have to use pictures if they do not convey useful information to the user. Item labels and the levels of the hierarchy may provide all the information the user needs.

Appearance of the TreeView

You can control the appearance of the TreeView by setting property values. Properties that affect the overall appearance are shown in Table 8-1.

Table 8-1: TreeView properties

Properties	Effect when set
HasButtons	Puts + and - buttons before items that have children, showing the user whether the item is expanded or collapsed
HasLines and LinesAtRoot	Display lines connecting items within a branch and connecting items at the root level
Checkboxes	Replaces the state image with checked and unchecked check boxes
TrackSelect	Changes the appearance of an item as the mouse moves over it
FullRowSelect	Highlights the entire row of a selected item
SingleExpand	Expands the selected item and collapses the previously selected item automatically
Indent	Sets the amount an item is indented
Font properties	Specifies the font for all the labels
Various picture properties	Controls the pictures and their size
LayoutRTL and RightToLeft	Display elements and characters in the control from right to left

For more information about these properties, see *Objects and Controls*.

User interaction

Basic TreeView functionality allows users to edit labels, delete items, expand and collapse branches, and sort alphabetically, without any scripting on your part. For example, the user can click a second time on a selected item to edit it, or press the Delete key to delete an item. If you do not want to allow these actions, properties let you disable them.

You can customize any of these basic actions by writing scripts. Events associated with the basic actions let you provide validation or prevent an action from completing. You can also implement other features such as adding items, dragging items, and performing customized sorting.

Using custom events

In PowerBuilder 7 and later releases, PowerBuilder uses Microsoft controls for ListView and Treeview controls. The events that fire when the right mouse button is clicked are different from earlier releases.

When you release the right mouse button, the pbm_rbuttonup event does not fire. The stock RightClicked! event for a TreeView control, pbm_tvnrclickedevent, fires when the button is released.

When you click the right mouse button on an unselected TreeView item, focus returns to the previously selected TreeView item when you release the button. To select the new item, insert this code in the pbm_tvnrclickedevent script before any code that acts on the selected item:

```
this.SelectItem(handle)
```

When you right double-click, only the pbm_rbuttondblclk event fires. In previous releases, both the pbm_rbuttondblclk and pbm_tvnrdoubleclick events fired.

Populating TreeViews

You must write a script to add items to a TreeView. You cannot add items in the painter as with other list controls. Although you can populate all the levels of the TreeView at once, TreeView events allow you to populate only branches the user looks at, which saves unnecessary processing.

Typically, you populate the first level of the TreeView when the control is displayed. This code might be in a window’s Open event, a user event triggered from the Open event, or the TreeView’s Constructor event. Then a script for the control’s ItemPopulate event would insert an item’s children when the user chooses to expand it.

The ItemPopulate event is triggered when the user clicks on an item's plus button or double-clicks the item, but only if the item's Children property is TRUE. Therefore, when you insert an item that will have children, you must set its Children property to TRUE so that it can be populated with child items when the user expands it.

You are not restricted to adding items in the ItemPopulate event. For example, you might let the user insert items by dragging from a ListBox or filling in a text box.

Functions for inserting items

There are several functions for adding items to a TreeView control, as shown in Table 8-2.

Table 8-2: Functions for adding items to TreeView control

This function	Adds an item here
InsertItem	After a sibling item for the specified parent. If no siblings exist, you must use one of the other insertion functions.
InsertItemFirst	First child of the parent item.
InsertItemLast	Last child of the parent item.
InsertItemSort	As a child of the parent item in alphabetic order, if possible.

For all the InsertItem functions, the SortType property can also affect the position of the added item.

There are two ways to supply information about the item you add, depending on the item properties that need to be set.

Method 1: specifying the label and picture index only

You can add an item by supplying the picture index and label. All the other properties of the item will have default values. You can set additional properties later as needed, using the item's handle.

Example This example inserts a new item after the currently selected item on the same level as that item. First it gets the handles of the currently selected item and its parent, and then it inserts an item labeled Hindemith after the currently selected item. The item's picture index is 2:

```
long ll_tvi, ll_tvparent

ll_tvi = tv_list.FindItem(CurrentTreeItem!, 0)
ll_tvparent = tv_list.FindItem(ParentTreeItem!, &
   ll_tvi)
```

```
tv_list.InsertItem(ll_tvparent, ll_tvi, &
   "Hindemith", 2)
```

Method 2: setting item properties in a TreeViewItem structure

You can add items by supplying a TreeViewItem structure with properties set to specific values. The only required property is a label. Properties you might set are shown in Table 8-3.

Table 8-3: TreeViewItem properties

Property	Value
Label	The text that is displayed for the item.
PictureIndex	A value from the regular picture list.
SelectedPictureIndex	A value from the regular picture list, specifying a picture that is displayed only when the item is selected. If 0, no picture is displayed for the item when selected.
StatePictureIndex	A value from the State picture list. The picture is displayed to the left of the regular picture.
Children	Must be TRUE if you want double-clicking to trigger the ItemPopulate event. That event script can insert child items.
Data	An optional value of any datatype that you want to associate with the item. You might use the value to control sorting or to make a database query.

Example This example sets all these properties in a TreeViewItem structure before adding the item to the TreeView control. The item is inserted as a child of the current item:

```
treeviewitem tvi
long h_item = 0, h_parent = 0

h_parent = tv_1.FindItem(CurrentTreeItem!, 0)
tvi.Label = "Choral"
tvi.PictureIndex = 1
tvi.SelectedPictureIndex = 2
tvi.Children = true
tvi.StatePictureIndex = 0
h_item = tv_1.InsertItemSort(h_parent, tvi)
```

For more information about inserting items into a TreeView control, see the *PowerScript Reference*.

Inserting items at the root level

The very first item you insert does not have any sibling for specifying a relative position, so you cannot use the InsertItem function—you must use InsertItemFirst or InsertItemLast. For an item inserted at the root level, you specify 0 as its parent.

This sample code is in a user event triggered from the Open event of the window containing the TreeView. It assumes two instance variable arrays:

- A string array called item_label that contains labels for all the items that will be inserted at the root level (here composer names)
- An integer array that has values for the Data property (the century for each composer); the century value is for user-defined sorting:

```
int ct
long h_item = 0
treeviewitem tvi

FOR ct = 1 TO UpperBound(item_label)
      tvi.Label = item_label[ct]
      tvi.Data = item_data[ct]
      tvi.PictureIndex = 1
      tvi.SelectedPictureIndex = 2
      tvi.Children = TRUE
      tvi.StatePictureIndex = 0
      tvi.DropHighlighted = TRUE
      h_item = tv_1.InsertItemSort(0, tvi)
NEXT
```

After inserting all the items, this code scrolls the TreeView back to the top and makes the first item current:

```
// Scroll back to top
h_item = tv_1.FindItem(RootTreeItem!, 0)
tv_1.SetFirstVisible(h_item)
tv_1.SelectItem(h_item)
```

Inserting items below the root level

The first time a user tries to expand an item to see its children, PowerBuilder triggers the ItemPopulate event *if and only if* the value of the item’s Children property is TRUE. In the ItemPopulate event, you can add child items for the item being expanded.

Parent item's Children property
If the ItemPopulate event does not occur when you expect, make sure the Children property for the expanding item is TRUE. It should be set to TRUE for any item that will have children.

Inserting items not restricted to the ItemPopulate event The ItemPopulate event helps you design an efficient program. It will not populate an item that the user never looks at. However, you do not have to wait until the user wants to view an item's children. You can add children in any script, just as you added items at the root level.

For example, you might fully populate a small TreeView when its window opens and use the ExpandAll function to display its items fully expanded.

Has an item been populated? You can check an item's ExpandedOnce property to find out if the user has looked at the item's children. If the user is currently looking at an item's children, the Expanded property is also TRUE.

Example This TreeView lists composers and their music organized into categories. The script for its ItemPopulate event checks whether the item being expanded is at level 1 (a composer) or level 2 (a category). Level 3 items are not expandable.

For a level 1 item, the script adds three standard categories. For a level 2 item, it adds pieces of music to the category being expanded, in this pattern:

```
Mozart
   Orchestral
      Symphony No. 33
      Overture to the Magic Flute
   Chamber
      Quintet in Eb for Horn and Strings
      Eine Kleine Nachtmusik
   Vocal
      Don Giovanni
      Idomeneo
```

This is the script for ItemPopulate:

```
TreeViewItem tvi_current, tvi_child, tvi_root
long hdl_root
Integer ct
string categ[]

// The current item is the parent for the new
itemsThis.GetItem(handle, tvi_current)
```

```
IF tvi_current.Level = 1 THEN
   // Populate level 2 with some standard categories
   categ[1] = "Orchestral"
   categ[2] = "Chamber"
   categ[3] = "Vocal"

      tvi_child.StatePictureIndex = 0
   tvi_child.PictureIndex = 3
   tvi_child.SelectedPictureIndex = 4
   tvi_child.OverlayPictureIndex = 0
   tvi_child.Children = TRUE

   FOR ct = 1 to UpperBound(categ)
      tvi_child.Label = categ[ct]
      This.InsertItemLast(handle, tvi_child)
   NEXT
END IF

// Populate level 3 with music titles
IF tvi_current.Level = 2 THEN

   // Get parent of current item - it's the root of
   // this branch and is part of the key for choosing
   // the children

   hdl_root = This.FindItem(ParentTreeItem!, handle)
   This.GetItem(hdl_root, tvi_root)

   FOR ct = 1 to 4
      // This statement constructs a label -
      // it is more realistic to look up data in a
      // table or database or accept user input
      This.InsertItemLast(handle, &
      tvi_root.Label + " Music " &
      + tvi_current.Label + String(ct), 3)
   NEXT
END IF
```

Managing TreeView items

An item in a TreeView is a TreeViewItem structure. The preceding section described how to set the item's properties in the structure and then insert it into the TreeView.

This code declares a TreeViewItem structure and sets several properties:

```
TreeViewItem tvi_defined

tvi_defined.Label = "Symphony No. 3 Eroica"
tvi_defined.StatePictureIndex = 0
tvi_defined.PictureIndex = 3
tvi_defined.SelectedPictureIndex = 4
tvi_defined.OverlayPictureIndex = 0
tvi_defined.Children = TRUE
```

For information about Picture properties, see "Managing TreeView pictures" on page 122.

When you insert an item, the inserting function returns a handle to that item. The TreeViewItem structure is copied to the TreeView control, and you no longer have access to the item's properties:

```
itemhandle = This.InsertItemLast(parenthandle, &
   tvi_defined)
```

Procedure for items: get, change, and set

If you want to change the properties of an item in the TreeView, you:

1 *Get* the item, which assigns it to a TreeViewItem structure.

2 Make the *changes*, by setting TreeViewItem properties.

3 *Set* the item, which copies it back into the TreeView.

When you work with items that have been inserted in the TreeView, you work with item handles. Most TreeView events pass one or two handles as arguments. The handles identify the items the user is interacting with.

This code for the Clicked event uses the handle of the clicked item to copy it into a TreeViewItem structure whose property values you can change:

```
treeviewitem tvi
This.GetItem(handle, tvi)
tvi.OverlayPictureIndex = 1
This.SetItem(handle, tvi)
```

Important
Remember to call the SetItem function after you change an item's property value. Otherwise, nothing happens in the TreeView.

Items and the hierarchy

You can use item handles with the FindItem function to navigate the TreeView and uncover its structure. The item's properties tell you what its level is, but not which item is its parent. The FindItem function does:

```
long h_parent
h_parent = This.FindItem(ParentTreeItem!, handle)
```

You can use FindItem to find the children of an item or to navigate through visible items regardless of level.

For more information, see the FindItem function in the *PowerScript Reference*.

Enabling TreeView functionality in scripts

By setting TreeView properties, you can enable or disable user actions like deleting or renaming items without writing any scripts. You can also enable these actions by calling functions. You can:

- Delete items
- Rename items
- Move items using drag and drop
- Sort items

Deleting items

To allow the user to delete items, enable the TreeView's DeleteItems property. When the user presses the Delete key, the selected item is deleted and the DeleteItem event is triggered. Any children are deleted too.

If you want more control over deleting, such as allowing deleting of detail items only, you can call the DeleteItem function instead of setting the property. The function also triggers the DeleteItem event.

Example

This script is for a TreeView user event. Its event ID is pbm_keydown and it is triggered by key presses when the TreeView has focus. The script checks whether the Delete key is pressed and whether the selected item is at the detail level. If both are TRUE, it deletes the item.

The value of the TreeView's DeleteItems property is FALSE. Otherwise, the user could delete any item, despite this code:

```
TreeViewItem tvi
long h_item

IF KeyDown(KeyDelete!) = TRUE THEN
   h_item = This.FindItem(CurrentTreeItem!, 0)
   This.GetItem(h_item, tvi)
   IF tvi.Level = 3 THEN
      This.DeleteItem(h_item
)   END IF
END IF
RETURN 0
```

Renaming items

If you enable the TreeView's EditLabels property, the user can edit an item label by clicking twice on the text.

Events

There are two events associated with editing labels.

The BeginLabelEdit event occurs after the second click when the EditLabels property is set or when the EditLabel function is called. You can disallow editing with a return value of 1.

This script for BeginLabelEdit prevents changes to labels of level 2 items:

```
TreeViewItem tvi
This.GetItem(handle, tvi)
IF tvi.Level = 2 THEN
   RETURN 1
ELSE
   RETURN 0
END IF
```

The EndLabelEdit event occurs when the user finishes editing by pressing ENTER, clicking on another item, or clicking in the text entry area of another control. A script you write for the EndLabelEdit event might validate the user's changes—for example, it could invoke a spelling checker.

EditLabel function

For control over label editing, the BeginLabelEdit event can prohibit editing of a label, as shown above. Or you can set the EditLabels property to FALSE and call the EditLabel function when you want to allow a label to be edited.

When you call the EditLabel function, the BeginLabelEdit event occurs when editing begins and the EndLabelEdit event occurs when the user presses enter or the user clicks another item.

This code for a CommandButton puts the current item into editing mode:

```
long h_tvi
h_tvi = tv_1.findItem(CurrentTreeItem!, 0)
tv_1.EditLabel(h_tvi)
```

Moving items using drag and drop

At the window level, PowerBuilder provides functions and properties for dragging controls onto other controls. Within the TreeView, you can also let the user drag items onto other items. Users might drag items to sort them, move them to another branch, or put child items under a parent.

When you implement drag and drop as a way to move items, you decide whether the dragged item becomes a sibling or child of the target, whether the dragged item is moved or copied, and whether its children get moved with it.

There are several properties and events that you need to coordinate to implement drag and drop for items, as shown in Table 8-4.

Table 8-4: Drag-and-drop properties and events

Property or event	Setting or purpose
DragAuto property	TRUE or FALSE If FALSE, you must call the Drag function in the BeginDrag event.
DisableDragDrop property	FALSE
DragIcon property	An appropriate icon *or* None!, which means the user drags an image of the item
BeginDrag event	Script for saving the handle of the dragged item and optionally preventing particular items from being dragged
DragWithin event	Script for highlighting drop targets
DragDrop event	Script for implementing the result of the drag operation

Example

The key to a successful drag-and-drop implementation is in the details. This section illustrates one way of moving items. In the example, the dragged item becomes a sibling of the drop target, inserted after it. All children of the item are moved with it and the original item is deleted.

A function called recursively moves the children, regardless of the number of levels. To prevent an endless loop, an item cannot become a child of itself. This means a drop target that is a child of the dragged item is not allowed.

BeginDrag event The script saves the handle of the dragged item in an instance variable:

```
ll_dragged_tvi_handle = handle
```

If you want to prevent some items from being dragged—such as items at a particular level—that code goes here too:

```
TreeViewItem tvi
This.GetItem(handle, tvi)
IF tvi.Level = 3 THEN This.Drag(Cancel!)
```

DragWithin event The script highlights the item under the cursor so the user can see each potential drop target. If only some items are drop targets, your script should check an item's characteristics before highlighting it. In this example, you could check whether an item is a parent of the dragged item and highlight it only if it is not:

```
TreeViewItem tvi
This.GetItem(handle, tvi)
tvi.DropHighlighted = TRUE
This.SetItem(handle, tvi)
```

DragDrop event This script does all the work. It checks whether the item can be inserted at the selected location and inserts the dragged item in its new position—a sibling after the drop target. Then it calls a function that moves the children of the dragged item too:

```
TreeViewItem tvi_src, tvi_child
long h_parent, h_gparent, h_moved, h_child
integer rtn

// Get TreeViewItem for dragged item
This.GetItem(ll_dragged_tvi_handle, tvi_src)
// Don't allow moving an item into its own branch,
// that is, a child of itself
h_gparent = This.FindItem(ParentTreeItem!, handle)

DO WHILE h_gparent <> -1
   IF h_gparent = ll_dragged_tvi_handle THEN
      MessageBox("No Drag", &
      "Can't make an item a child of itself.")
      RETURN 0
   END IF
```

```
h_gparent=This.FindItem(ParentTreeItem!, h_gparent)
LOOP

// Get item parent for inserting
h_parent = This.FindItem(ParentTreeItem!, handle)

// Use 0 if no parent because target is at level 1
IF h_parent = -1 THEN h_parent = 0

// Insert item after drop target
h_moved = This.InsertItem(h_parent, handle, tvi_src)
IF h_moved = -1 THEN
   MessageBox("No Dragging", "Could not move item.")
   RETURN 0
ELSE
   // Args: old parent, new parent
   rtn = uf_movechildren(ll_dragged_tvi_handle, &
      h_moved)

   / If all children are successfully moved,
   // delete original item
   IF rtn = 0 THEN
      This.DeleteItem(ll_dragged_tvi_handle)
   END IF

END IF
```

The DragDrop event script shown above calls the function uf_movechildren. The function calls itself recursively so that all the levels of children below the dragged item are moved:

```
// Function: uf_movechildren
// Arguments:
// oldparent - Handle of item whose children are
// being moved. Initially, the dragged item in its
// original position
//
// newparent - Handle of item to whom children are
// being moved. Initially, the dragged item in its
// new position.

long h_child, h_movedchild
TreeViewItem tvi

// Return -1 if any Insert action fails

// Are there any children?
```

```
h_child = tv_2.FindItem(ChildTreeItem!, oldparent)
IF h_child <> -1 THEN
   tv_2.GetItem(h_child, tvi)
   h_movedchild = tv_2.InsertItemLast(newparent, tvi)
   IF h_movedchild = -1 THEN RETURN -1

   // Move the children of the child that was found
   uf_movechildren(h_child, h_movedchild)

   // Check for more children at the original level
   h_child = tv_2.FindItem(NextTreeItem!, h_child)
   DO WHILE h_child <> -1
      tv_2.GetItem(h_child, tvi)
      h_movedchild= tv_2.InsertItemLast(newparent,tvi)
      IF h_movedchild = -1 THEN RETURN -1
      uf_movechildren(h_child, h_movedchild)

      // Any more children at original level?
      h_child = tv_2.FindItem(NextTreeItem!, h_child)
   LOOP
END IF
RETURN 0 // Success, all children moved
```

Sorting items

A TreeView can sort items automatically, or you can control sorting manually. Manual sorting can be alphabetic by label text, or you can implement a user-defined sort to define your own criteria. The SortType property controls the way items are sorted. Its values are of the enumerated datatype grSortType.

Automatic alphabetic sorting To enable sorting by the text label, set the SortType property to Ascending! or Descending!. Inserted items are sorted automatically.

Manual alphabetic sorting For more control over sorting, you can set SortType to Unsorted! and sort by calling the functions in Table 8-5.

Table 8-5: TreeView sorting functions

Use this function	To do this
InsertItemSort	Insert an item at the correct alphabetic position, if possible
Sort	Sort the immediate children of an item
SortAll	Sort the whole branch below an item

If users will drag items to organize the list, you should disable sorting.

Sorting by other criteria To sort items by criteria other than their labels, implement a user-defined sort by setting the SortType property to UserDefinedSort! and writing a script for the Sort event. The script specifies how to sort items.

PowerBuilder triggers the Sort event for each pair of items it tries to reorder. The Sort script returns a value reporting which item is greater than the other. The script can have different rules for sorting based on the type of item. For example, level 2 items can be sorted differently from level 3. The TreeView is sorted whenever you insert an item.

Example of Sort event

This sample script for the Sort event sorts the first level by the value of the Data property and other levels alphabetically by their labels. The first level displays composers chronologically, and the Data property contains an integer identifying a composer's century:

```
//Return values
//-1   Handle1 is less than handle2
// 0   Handle1 is equal to handle2
// 1   Handle1 is greater than handle2

TreeViewItem tvi1, tvi2

This.GetItem(handle1, tvi1)
This.GetItem(handle2, tvi2)

IF tvi1.Level = 1 THEN
   // Compare century values stored in Data property
   IF tvi1.data > tvi2.Data THEN
      RETURN 1
   ELSEIF tvi1.data = tvi2.Data THEN
      RETURN 0
   ELSE
      RETURN -1
   END IF
   ELSE
   // Sort other levels in alpha order
   IF tvi1.Label > tvi2.Label THEN
      RETURN 1
   ELSEIF tvi1.Label = tvi2.Label THEN
      RETURN 0
   ELSE
      RETURN -1
   END IF
END IF
```

Managing TreeView pictures

PowerBuilder stores TreeView images in three image lists:

- Picture list (called the *regular picture list* here)
- State picture list
- Overlay picture list

You add pictures to these lists and associate them with items in the TreeView.

Pictures for items

There are several ways to use pictures in a TreeView. You associate a picture in one of the picture lists with an item by setting one of the item's picture properties, described in Table 8-6.

Table 8-6: TreeView picture properties

Property	Purpose
PictureIndex	The primary picture associated with the item is displayed just to the left of the item's label.
StatePictureIndex	A state picture is displayed to the left of the regular picture. The item moves to the right to make room for the state picture. If the Checkboxes property is TRUE, the state picture is replaced by a pair of check boxes. Because a state picture takes up room, items without state pictures will not align with items that have pictures. So that all items have a state picture and stay aligned, you could use a blank state picture for items that do not have a state to be displayed. A use for state pictures might be to display a check mark beside items the user has chosen.
OverlayPictureIndex	An overlay picture is displayed on top of an item's regular picture. You set up the overlay picture list in a script by designating a picture in the regular picture list for the overlay picture list. An overlay picture is the same size as a regular picture, but it often uses a small portion of the image space so that it only partially covers the regular picture. A typical use of overlay pictures is the arrow marking shortcut items in the Windows Explorer.

Property	Purpose
SelectedPictureIndex	A picture from the regular picture list that is displayed instead of the regular picture when the item is the current item. When the user selects another item, the first item gets its regular picture and the new item displays its selected picture. If you do not want a different picture when an item is current, set SelectedPictureIndex to the same value as PictureIndex.

How to set pictures You can change the pictures for all items at a particular level with the SetLevelPictures function, or you can set the picture properties for an individual item.

If you do not want pictures Your TreeView does not have to use pictures for items. If an item's picture indexes are 0, no pictures are displayed. However, the TreeView always leaves room for the regular picture. You can set the PictureWidth property to 0 to eliminate that space:

```
tv_2.DeletePictures()
tv_2.PictureWidth = 0
```

Setting up picture lists

You can add pictures to the regular and state picture lists in the painter or during execution. During execution, you can assign pictures in the regular picture list to the overlay list.

Mask color

The mask color is a color used in the picture that becomes transparent when the picture is displayed. Usually you should pick the picture's background color so that the picture blends with the color of the TreeView.

Before you add a picture, in the painter or in a script, you can set the mask color to a color appropriate for that picture. This statement sets the mask color to white, which is right for a picture with a white background:

```
tv_1.PictureMaskColor = RGB(255, 255, 255)
```

Each picture can have its own mask color. A picture uses the color that is in effect when the picture is inserted. To change a picture's mask color, you have to delete the picture and add it again.

Image size

In the painter you can change the image size at any time by setting the Height and Width properties on each picture tab. All the pictures in the list are scaled to the specified size.

During execution, you can change the image size for a picture list only when that list is empty. The DeletePictures and DeleteStatePictures functions let you clear the lists.

Example

This sample code illustrates how to change properties and add pictures to the regular picture list during execution. Use similar code for state pictures:

```
tv_list.DeletePictures()
tv_list.PictureHeight = 32
tv_list.PictureWidth = 32

tv_list.PictureMaskColor = RGB(255,255,255)
tv_list.AddPicture("c:\apps_pb\kelly.bmp")
tv_list.PictureMaskColor = RGB(255,0,0)
tv_list.AddPicture("Custom078!")
tv_list.PictureMaskColor = RGB(128,128,128)
tv_list.AddPicture("Custom044!")
```

Deleting pictures and how it affects existing items

Deleting pictures from the picture lists can have an unintended effect on item pictures being displayed. When you delete pictures, the remaining pictures in the list are shifted to remove gaps in the list. The remaining pictures get a different index value. This means items that use these indexes get new images.

Deleting pictures from the regular picture list also affects the overlay list, since the overlay list is not a separate list but points to the regular pictures.

To avoid unintentional changes to item pictures, it is best to avoid deleting pictures after you have begun using picture indexes for items.

Using overlay pictures

The pictures in the overlay list come from the regular picture list. First you must add pictures to the regular list, either in the painter or during execution. Then during execution you specify pictures for the overlay picture list. After that you can assign an overlay picture to items, individually or with the SetLevelPictures function.

This code adds a picture to the regular picture list and then assigns it to the overlay list:

```
integer idx
idx = tv_1.AddPicture("Custom085!")
IF tv_1.SetOverlayPicture(1, idx) <> 1 THEN
   sle_get.Text = "Setting overlay picture failed"
END IF
```

This code for the Clicked event turns the overlay picture on or off each time the user clicks an item:

```
treeviewitem tvi
This.GetItem(handle, tvi)
IF tvi.OverlayPictureIndex = 0 THEN
   tvi.OverlayPictureIndex = 1
ELSE
   tvi.OverlayPictureIndex = 0
END IF
This.SetItem(handle, tvi)
```

Using DataWindow information to populate a TreeView

A useful implementation of the TreeView control is to populate it with information that you retrieve from a DataWindow. To do this your application must:

- Declare and instantiate a DataStore and assign a DataWindow object
- Retrieve information as needed
- Use the retrieved information to populate the TreeView
- Destroy the DataStore instance when you have finished

Because a TreeView can display different types of information at different levels, you will probably define additional DataWindows, one for each level. Those DataWindows usually refer to different but related tables. When an item is expanded, the item becomes a retrieval argument for getting child items.

Populating the first level

This example populates a TreeView with a list of composers. The second level of the TreeView displays music by each composer. In the database there are two tables: composer names and music titles (with composer name as a foreign key).

This example declares two DataStore instance variables for the window containing the TreeView control:

```
datastore ids_data, ids_info
```

This example uses the TreeView control’s Constructor event to:

- Instantiate the DataStore
- Associate it with a DataWindow and retrieve information

- Use the retrieved data to populate the root level of the TreeView:

```
//Constructor event for tv_1
treeviewitem tvi1, tvi2
long ll_lev1, ll_lev2, ll_rowcount, ll_row

//Create instance variable datastore
ids_data = CREATE datastore
ids_data.DataObject = "d_composers"
ids_data.SetTransObject(SQLCA)
ll_rowcount = ids_data.Retrieve()

//Create the first level of the TreeView
tvi1.PictureIndex = 1
tvi1.Children = TRUE
//Populate the TreeView with
//data retrieved from the datastore
FOR ll_row = 1 to ll_rowcount
      tvi1.Label = ids_data.GetItemString(ll_row, &
      'name')
      This.InsertItemLast(0, tvi1)
NEXT
```

Populating the second level

When the user expands a root level item, the ItemPopulate event occurs. This script for the event:

- Instantiates a second DataStore

  Its DataWindow uses the composer name as a retrieval argument for the music titles table.

- Inserts music titles as child items for the selected composer

  The handle argument of ItemPopulate will be the parent of the new items:

```
//ItemPopulate event for tv_1
TreeViewItem tvi1, tvi2
long ll_row, ll_rowcount

//Create instance variable datastore
ids_info = CREATE datastore
ids_info.DataObject = "d_music"
ids_info.SetTransObject(SQLCA)

//Use the label of the item being populated
// as the retrieval argument
This.GetItem(handle, tvi1)
ll_rowcount = ids_info.Retrieve(tvi1.Label)
```

```
//Use information retrieved from the database
//to populate the expanded item
FOR ll_row = 1 to ll_rowcount
      This.InsertItemLast(handle, &
      ids_info.GetItemString(ll_row, &
      music_title'), 2)
LOOP
```

Destroying DataStore instances

When the window containing the TreeView control closes, this example destroys the DataStore instances:

```
//Close event for w_treeview
DESTROY ids_data
DESTROY ids_info
```

CHAPTER 9 Using Lists in a Window

About this chapter

This chapter describes how to use lists to present information in an application.

Contents

About presenting lists

You can choose a variety of ways to present lists in your application:

- ListBoxes and PictureListBoxes display available choices that can be used for invoking an action or viewing and displaying data.
- DropDownListBoxes and DropDownPictureListBoxes also display available choices to the user. However, you can make them editable to the user. DropDownListBoxes are text-only lists; DropDownPictureListBoxes display a picture associated with each item.
- ListView controls present lists in a combination of graphics and text. You can allow the user to add, delete, edit, and rearrange ListView items, or you can use them to invoke an action.

TreeView controls
TreeView controls also combine graphics and text in lists. The difference is that TreeView controls show the hierarchical relationship among the TreeView items. As with ListView controls, you can allow the user to add, delete, edit, and rearrange TreeView items. You can also use them to invoke actions.

For more information on TreeViews, see Chapter 8, “Using TreeView Controls.”

Using lists

You can use lists to present information to the user in simple lists with scroll bars. You can present this information with text and pictures (in a PictureListBox) or with text alone (using a ListBox).

Depending on how you design your application, the user can select one or more list items to perform an action based on the list selection.

You add ListBox and PictureListBox controls to windows in the same way you add other controls: select ListBox or PictureListBox from the Insert>Control menu and click the window.

Adding items to list controls

In the painter Use the Item property page for the control to add new items.

❖ **To add items to a ListBox or PictureListBox:**

1 Select the Items tab in the Properties view for the control.

2 Enter the name of the items for the ListBox. For a PictureListBox, also enter a picture index number to associate the item with a picture.

For instructions on adding pictures to a PictureListBox, see “Adding pictures to PictureListBox controls” on page 131.

In a script Use the AddItem and InsertItem functions to dynamically add items to a ListBox or PictureListBox at runtime. AddItem adds items to the end of the list. However, if the list is sorted, the item will then be moved to its position in the sort order. Use InsertItem if you want to specify where in the list the item will be inserted.

Table 9-1: Using the InsertItem and AddItem functions

Function	You supply
InsertItem	Item name Position in which the item will be inserted Picture index (for a PictureListBox)
AddItem	Item name Picture index (for a PictureListBox)

For example, this script adds items to a ListBox:

```
This.AddItem ("Vaporware")
This.InsertItem ("Software",2)
This.InsertItem ("Hardware",2)
This.InsertItem ("Paperware",2)
```

This script adds items and images to a PictureListBox:

```
This.AddItem ("Monitor",2)
This.AddItem ("Modem", 3)
This.AddItem ("Printer",4)
This.InsertItem ("Scanner",5,1)
```

Using the Sort property
You can set the control's sort property to TRUE or check the Sorted check box on the General property page to ensure that the items in the list are always arranged in ascending alphabetical order.

Adding pictures to PictureListBox controls

In the painter Use the Pictures and Items property pages for the control to add pictures.

❖ **To add pictures to a PictureListBox:**

1 Select the Pictures tab in the Properties view for the control.

2 Select an image from the stock image list, or use the Browse button to select a bitmap, cursor, or icon image.

3 Select a color from the PictureMaskColor drop-down menu for the image.

The color selected for the picture mask will appear transparent in the PictureListBox.

4 Select a picture height and width.

This will control the size of the images in the PictureListBox.

Dynamically changing image size
You can use a script to change the image size at runtime by setting the PictureHeight and PictureWidth properties before you add any pictures when you create a PictureListBox.

For more information about PictureHeight and PictureWidth, see the *PowerScript Reference*.

5 Repeat the procedure for the number of images you plan to use in your PictureListBox.

6 Select the Items tab and change the Picture Index for each item to the appropriate number.

In a script Use the AddPicture function to dynamically add pictures to a PictureListBox at runtime. For example, the script below sets the size of the picture, adds a *BMP* file to the PictureListBox, and adds an item to the control:

```
This.PictureHeight = 75
This.PictureWidth = 75
This.AddPicture ("c:\ArtGal\bmps\butterfly.bmp")
This.AddItem("Aine Minogue",8)
```

Deleting pictures from picture list controls

Use the DeletePicture and DeletePictures functions to delete pictures from either a PictureListBox or a DropDownPictureListBox.

When you use the DeletePicture function, you must supply the picture index of the picture you want to delete.

For example:

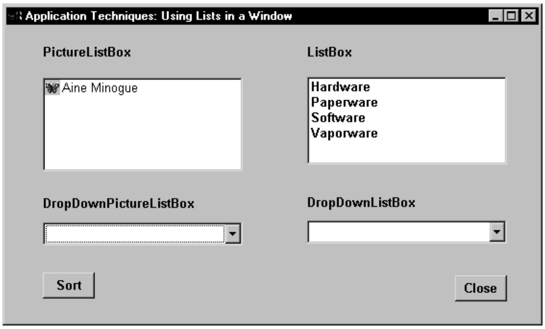
```
This.DeletePicture (1)
```

deletes the first Picture from the control, and

```
This.DeletePictures ()
```

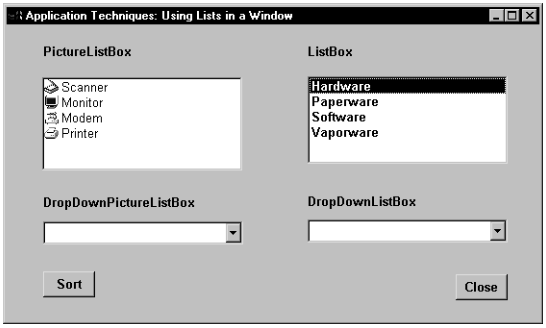
deletes all the pictures in a control.

Example The following window contains a ListBox control and a PictureListBox. The ListBox control contains four items, and the PictureListBox has one:



When the user double-clicks an item in the ListBox, a script executes to:

- Delete all the items in the PictureListBox
- Add new items to the PictureListBox that are related to the ListBox item that was double-clicked



This is the script used in the ListBox DoubleClicked event:

```
int li_count
//Find out the number of items
//in the PictureListBox
li_count = plb_1.totalItems()

// Find out which item was double-clicked
// Then:
// * Delete all the items in the PictureListBox
// * Add the items associated with the
//    double-clicked item
```

```
CHOOSE CASE index
   CASE 1
      DO WHILE plb_1.totalitems() > 0
      plb_1.DeleteItem(plb_1.totalitems())
      LOOP
      plb_1.AddItem("Monitor",2)
      plb_1.AddItem("Modem",3)
      plb_1.AddItem("Printer",4)
      plb_1.InsertItem("Scanner",5,1)

   CASE 2
      DO WHILE plb_1.totalitems() > 0
      plb_1.DeleteItem(plb_1.totalitems())
      LOOP
      plb_1.InsertItem("GreenBar",6,1)
      plb_1.InsertItem("LetterHead",7,1)
      plb_1.InsertItem("Copy",8,1)
      plb_1.InsertItem("50 lb.",9,1)

   CASE 3
      DO WHILE plb_1.totalitems() > 0
      plb_1.DeleteItem(plb_1.totalitems())
      LOOP
      plb_1.InsertItem("SpreadIt!",10,1)
      plb_1.InsertItem("WriteOn!",11,1)
      plb_1.InsertItem("WebMaker!",12,1)
      plb_1.InsertItem("Chessaholic",13,1)

   CASE 4
      DO WHILE plb_1.totalitems() > 0
      plb_1.DeleteItem(plb_1.totalitems())
      LOOP
      plb_1.InsertItem("AlnaWarehouse",14,1)
      plb_1.InsertItem("AlnaInfo",15,1)
      plb_1.InsertItem("Info9000",16,1)
      plb_1.InsertItem("AlnaThink",17,1)

END CHOOSE
```

Using drop-down lists

Drop-down lists are another way to present simple lists of information to the user. You can present your lists with just text (in a DropDownListBox) or with text and pictures (in a DropDownPictureListBox). You add DropDownListBox and DropDownPictureListBox controls to windows in the same way you add other controls: select DropDownListBox or DropDownPictureListBox from the Insert>Control menu and click the window.

Adding items to drop-down list controls

In the painter Use the Items property page for the control to add items.

❖ **To add items to a DropDownListBox or DropDownPictureListBox:**

1 Select the Items tab in the Properties view for the control.

2 Enter the name of the items for the ListBox. For a PictureListBox, also enter a picture index number to associate the item with a picture.

For how to add pictures to a DropDownPictureListBox, see “Adding pictures to DropDownPicture ListBox controls” on page 136.

In a script Use the AddItem and InsertItem functions to dynamically add items to a DropDownListBox or DropDownPictureListBox at runtime.

AddItem adds items to the end of the list. However, if the list is sorted, the item will then be moved to its position in the sort order. Use InsertItem if you want to specify where in the list the item will be inserted.

Table 9-2: Using the InsertItem and AddItem functions

Function	You supply
InsertItem	Item name Picture index (for a DropDownPictureListBox) Position in which the item will be inserted
AddItem	Item name Picture index (for a DropDownPictureListBox)

This example inserts three items into a DropDownPictureListBox in the first, second, and third positions:

```
This.InsertItem ("Atropos", 2, 1)
This.InsertItem ("Clotho", 2, 2)
This.InsertItem ("Lachesis", 2, 3)
```

This example adds two items to a DropDownPictureListBox:

```
this.AddItem ("Plasma", 2)
this.AddItem ("Platelet", 2)
```

Using the Sort property
You can set the control's sort property to TRUE to ensure that the items in the list are always arranged in ascending sort order.

Adding pictures to DropDownPicture ListBox controls

In the painter Use the Pictures and Items property pages for the control to add pictures.

❖ **To add pictures to a DropDownPictureListBox:**

1 Select the Pictures tab in the Properties view for the control.

2 Select an image from the stock image list, or use the Browse button to select a bitmap, cursor, or icon image.

3 Select a color from the PictureMaskColor drop-down menu for the image.

The color selected for the picture mask will appear transparent in the DropDownPictureListBox.

4 Select a picture height and width for your image.

This will control the size of the image in the DropDownPictureListBox.

Dynamically changing image size
The image size can be changed at runtime by setting the PictureHeight and PictureWidth properties before you add any pictures when you create a DropDownPictureListBox. For more information about PictureHeight and PictureWidth, see the *PowerScript Reference*.

5 Repeat the procedure for the number of images you plan to use in your DropDownPictureListBox.

6 Select the Items tab and change the Picture Index for each item to the appropriate number.

In a script Use the AddPicture function to dynamically add pictures to a PictureListBox at runtime. For instance, this example adds two *BMP* files to the PictureListBox:

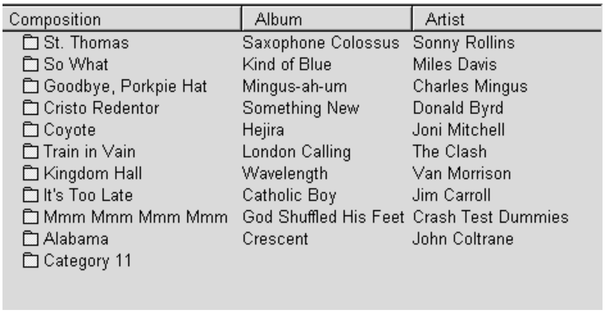
```
This.AddPicture ("c:\images\justify.bmp")
This.AddPicture ("c:\images\center.bmp")
```

Deleting pictures from DropDownPicture ListBox controls

For instructions on deleting pictures from DropDownPictureListBox controls, see "Deleting pictures from picture list controls" on page 132.

Using ListView controls

A **ListView control** allows you to display items and icons in a variety of arrangements. You can display large icon or small icon freeform lists, or you can display a vertical static list. You can also display additional information about each list item by associating additional columns with each list item:



Composition	Album	Artist
St. Thomas	Saxophone Colossus	Sonny Rollins
So What	Kind of Blue	Miles Davis
Goodbye, Porkpie Hat	Mingus-ah-um	Charles Mingus
Cristo Redentor	Something New	Donald Byrd
Coyote	Hejira	Joni Mitchell
Train in Vain	London Calling	The Clash
Kingdom Hall	Wavelength	Van Morrison
It's Too Late	Catholic Boy	Jim Carroll
Mmm Mmm Mmm Mmm	God Shuffled His Feet	Crash Test Dummies
Alabama	Crescent	John Coltrane
Category 11		

ListView controls consist of **ListView items**, which are stored in an array. Each ListView item consists of a:

- **Label** The name of the ListView item
- **Index** The position of the ListView item in the control
- **Picture index** The number that associates the ListView item with an image

  Depending on the style of the presentation, an item could be associated with a large picture index and a small picture index.
- **Overlay picture index** The number that associates the ListView item with an overlay picture
- **State picture index** The number that associates the ListView item with a state picture

For more information about ListView items, picture indexes, and presentation style, see the *PowerBuilder User's Guide*.

Creating ListView controls

You add ListView controls to windows in the same way you add other controls: select ListView from the Insert>Control menu and click the window.

Adding ListView items

In the painter Use the Items property page for the control to add items.

❖ To add items to a ListView:

1. Select the Items tab in the Properties view for the control.
2. Enter a name and a picture index number for each of the items you want to add to the ListView.

Clearing all entries on the Items tab page
Setting the picture index for the first item to zero clears all the settings on the tab page.

For more information about adding pictures to a ListView control, see “Adding pictures to ListView controls” on page 139.

In a script Use the AddItem and InsertItem functions to add items to a ListView dynamically at runtime. There are two levels of information you supply when you add items to a ListView using AddItem or InsertItem.

You can add an item by supplying the picture index and label, as this example shows:

```
lv_1.AddItem ("Item 1", 1)
```

or you can insert an item by supplying the item’s position in the ListView, label, and picture index:

```
lv_1.InsertItem (1,"Item 2", 2)
```

You can add items by supplying the ListView item itself. This example in the ListView’s DragDrop event inserts the dragged object into the ListView:

```
listviewitem lvi
This.GetItem(index, lvi)
lvi.label = "Test"
lvi.pictureindex = 1
This.AddItem (lvi)
```

You can insert an item by supplying the ListView position and ListView item:

```
listviewitem l_lvi
//Obtain the information for the
//second listviewitem
lv_list.GetItem(2, l_lvi)
//Change the item label to Entropy
//Insert the second item into the fifth position
lv_list.InsertItem (5, l_lvi)
lv_list.DeleteItem(2)
```

Adding pictures to ListView controls

PowerBuilder stores ListView images in four image lists:

- Small picture index
- Large picture index
- State picture index
- Overlay picture index

You can associate a ListView item with these images when you create a ListView in the painter or use the AddItem and InsertItem at runtime.

However, before you can associate pictures with ListView items, they must be added to the ListView control.

In the painter Use the Pictures and Items property pages for the control to add pictures.

❖ To add pictures to a ListView control:

1 Select the Large Picture, Small Picture, or State tab in the Properties view for the control.

Overlay images
You can add overlay images only to a ListView control in a script.

2 Select an image from the stock image list, or use the Browse button to select a bitmap, cursor, or icon image.

3 Select a color from the PictureMaskColor drop-down menu for the image.

The color selected for the picture mask appears transparent in the ListView.

4 Select a picture height and width for your image.

This controls the size of the image in the ListView.

Dynamically changing image size
The image size can be changed at runtime by setting the PictureHeight and PictureWidth properties before you add any pictures when you create a ListView. For more information about PictureHeight and PictureWidth, see the *PowerScript Reference*.

5 Repeat the procedure for the:

- Number of image types (large, small, and state) you plan to use in your ListView
- Number of images for each type you plan to use in your ListView

In a script Use the functions in Table 9-3 to add pictures to a ListView image.

Table 9-3: Functions that add pictures to a ListView image

Function	Adds a picture to this list
AddLargePicture	Large image
AddSmallPicture	Small image
AddStatePicture	State image

Adding large and small pictures This example sets the height and width for large and small pictures and adds three images to the large picture image list and the small picture image list:

```
//Set large picture height and width
lv_1.LargePictureHeight=32
lv_1.LargePictureWidth=32

//Add large pictures
lv_1.AddLargePicture("c:\ArtGal\bmps\celtic.bmp")
lv_1.AddLargePicture("c:\ArtGal\bmps\list.ico")
lv_1.AddLargePicture("Custom044!")

//Set small picture height and width
lv_1.SmallPictureHeight=16
lv_1.SmallPictureWidth=16

//Add small pictures
lv_1.AddSmallPicture("c:\ArtGal\bmps\celtic.bmp")
lv_1.AddSmallPicture("c:\ArtGal\bmps\list.ico")
lv_1.AddSmallPicture("Custom044!")

//Add items to the ListView
lv_1.AddItem("Item 1", 1)
lv_1.AddItem("Item 2", 1)
lv_1.AddItem("Item 3", 1)
```

Adding overlay pictures Use the SetOverLayPicture function to use a large picture or small picture as an overlay for an item. This example adds a large picture to a ListView, and then uses it for an overlay picture for a ListView item:

```
listviewitem lvi_1
int li_index

//Add a large picture to a ListView
li_index = lv_list.AddLargePicture &
   ("c:\ArtGal\bmps\dil2.ico")

//Set the overlay picture to the
//large picture just added
lv_list.SetOverlayPicture (3, li_index)

//Use the overlay picture with a ListViewItem
lv_list.GetItem(lv_list.SelectedIndex (), lvi_1)
lvi_1.OverlayPictureIndex = 3
lv_list.SetItem(lv_list.SelectedIndex (), lvi_1)
```

Adding state pictures This example uses an item's state picture index property to set the state picture for the selected ListView item:

```
listviewitem lvi_1

lv_list.GetItem(lv_list.SelectedIndex (), lvi_1)
lvi_1.StatePictureIndex = 2
lv_list.SetItem(lv_list.SelectedIndex (), lvi_1)
```

Deleting ListView items and pictures

You can delete items from a ListView one at a time with the DeleteItem function, or you can use the DeleteItems function to purge all the items in a ListView. Similarly, you can delete pictures one at a time with the DeleteLargePicture, DeleteSmallPicture, and DeleteStatePicture functions, or purge all pictures of a specific type by using the DeleteLargePictures, DeleteSmallPictures, and DeleteStatePictures functions.

This example deletes one item and all the small pictures from a ListView:

```
int li_index
li_index = This.SelectedIndex()
This.DeleteItem (li_index)
This.DeleteSmallPictures ()
```

Hot tracking and one- or two-click activation

Hot tracking changes the appearance of items in the Listview control as the mouse moves over them and, if the mouse pauses, selects the item under the cursor automatically. You can enable hot tracking by setting the TrackSelect property to TRUE.

Setting either OneClickActivate or TwoClickActivate to TRUE also enables hot tracking. When OneClickActivate is TRUE, you can specify that either selected or unselected items are underlined by setting the UnderlineHot or UnderlineCold properties. All these properties can be set on the control's general properties page or in a script.

The settings for OneClickActivate and TwoClickActivate shown in Table 9-4 affect when the ItemActivate event is fired.

Table 9-4: OneClickActivate and TwoClickActivate settings

OneClickActivate	TwoClickActivate	ItemActivate is fired when you
TRUE	TRUE or FALSE	Click any item
FALSE	TRUE	Click a selected item
FALSE	FALSE	Double-click any item

Using custom events

In PowerBuilder 7 and later releases, PowerBuilder uses Microsoft controls for ListView and Treeview controls, and the events that fire when the right mouse button is clicked are different than in earlier releases. These are the events that fire when the right mouse button is clicked in a ListView control:

Table 9-5: ListView control events fired by right mouse button

Location	Action	Events fired
On an item in the ListView	Press right mouse button	pbm_rbuttondown
	Release right mouse button	pbm_lvnrclicked (stock RightClicked! event) pbm_contextmenu
On white space in the ListView	Press right mouse button	pbm_rbuttondown pbm_lvnrclicked (stock RightClicked! event) pbm_contextmenu
	Release right mouse button	pbm_rbuttonup pbm_contextmenu

Using report view

ListView report view requires more information than the large icon, small icon, and list view. To enable report view in a ListView control, you must write a script that establishes columns with the AddColumn and SetColumn functions, and then populate the columns using the SetItem function.

Populating columns

Use AddColumn to create columns in a ListView. When you use the AddColumn function, you specify the:

- **Column label** The name that will display in the column header
- **Column alignment** Whether the text will be left-aligned, right-aligned, or centered
- **Column size** The width of the column in PowerBuilder units

This example creates three columns in a ListView:

```
This.AddColumn("Name", Left!, 1000)
This.AddColumn("Size", Left!, 400)
This.AddColumn("Date", Left!, 300)
```

Setting columns

Use SetColumn to set the column number, name, alignment, and size:

```
This.SetColumn (1, "Composition", Left!, 860)
This.SetColumn (2, "Album", Left!, 610)
This.SetColumn (3, "Artist", Left!, 710")
```

Setting column items

Use SetItem to populate the columns of a ListView:

```
This.SetItem (1, 1, "St.Thomas")
This.SetItem (1, 2, "Saxophone Colossus")
This.SetItem (1, 3, "Sonny Rollins")
This.SetItem (2, 1, "So What")
This.SetItem (2, 2, "Kind of Blue")
This.SetItem (2, 3, "Miles Davis")
This.SetItem (3, 1, "Good-bye, Porkpie Hat")
This.SetItem (3, 2, "Mingus-ah-um")
This.SetItem (3, 3, "Charles Mingus")
```

For more information about adding columns to a ListView control, see the *PowerScript Reference*.

CHAPTER 10

Using Drag and Drop in a Window

About this chapter

This chapter describes how to make applications graphical by dragging and dropping controls.

Contents

About drag and drop

Drag and drop allows users to initiate activities by dragging a control and dropping it on another control. It provides a simple way to make applications graphical and easy to use. For example, in a manufacturing application you might allow the user to pick parts from a bin for an assembly by dragging a picture of the part and dropping it in the picture of the finished assembly.

Drag and drop involves at least two controls: the control that is being dragged (the **drag control**) and the control to which it is being dragged (the **target**). In PowerBuilder, all controls except drawing objects (lines, ovals, rectangles, and rounded rectangles) can be dragged.

Automatic drag mode

When a control is being dragged, it is in drag mode. You can define a control so that PowerBuilder puts it automatically in drag mode whenever a Clicked event occurs in the control, or you can write a script to put a control into drag mode when an event occurs in the window or the application.

Drag icons

When you define the style for a draggable object in the Window painter, you can specify a drag icon for the control. The drag icon displays when the control is dragged over a valid drop area (an area in which the control can be dropped). If you do not specify a drag icon, a rectangle the size of the control displays.

Drag events

Window objects and all controls except drawing objects have events that occur when they are the drag target. When a dragged control is within the target or dropped on the target, these events can trigger scripts. The scripts determine the activity that is performed when the drag control enters, is within, leaves, or is dropped on the target.

Drag-and-drop properties, events, and functions

Drag-and-drop properties

Each PowerBuilder control has two drag-and-drop properties:

- DragAuto
- DragIcon

The DragAuto property DragAuto is a boolean property.

Table 10-1: DragAuto property values

Value	Meaning
TRUE	When the object is clicked, the control is placed automatically in drag mode
FALSE	When the object is clicked, the control is not placed automatically in drag mode; you have to put the object in drag mode manually by using the Drag function in a script

❖ **To specify automatic drag mode for a control in the Window painter:**

1 Select the Other property page in the Properties view for the control.

2 Check the Drag Auto check box.

The DragIcon property Use the DragIcon property to specify the icon you want displayed when the control is in drag mode. The DragIcon property is a stock icon or a string identifying the file that contains the icon (the *ICO* file). The default icon is a box the size of the control.

When the user drags a control, the icon displays when the control is over an area in which the user can drop it (a valid drop area). When the control is over an area that is not a valid drop area (such as a window scroll bar), the No-Drop icon displays.

❖ **To specify a drag icon:**

1 Select the Other property page in the Properties view for the control.

2 Choose the icon you want to use from the list of stock icons or use the Browse button to select an *ICO* file and click OK.

Creating icons
To create icons, use a drawing application that can save files in the Microsoft Windows *ICO* format.

Drag-and-drop events

There are six drag-and-drop events.

Table 10-2: Drag-and-drop events

Event	Occurs
BeginDrag	When the user presses the left mouse button in a ListView or TreeView control and begins dragging
BeginRightDrag	When the user presses the right mouse button in a ListView or TreeView control and begins dragging
DragDrop	When the hot spot of a drag icon (usually its center) is over a target (a PowerBuilder control or window to which you drag a control) and the mouse button is released
DragEnter	When the hot spot of a drag icon enters the boundaries of a target
DragLeave	When the hot spot of a drag icon leaves the boundaries of a target
DragWithin	When the hot spot of a drag icon moves within the boundaries of a target

Drag-and-drop functions

Each PowerBuilder control has two functions you can use to write scripts for drag-and-drop events.

Table 10-3: Drag-and-drop event functions

Function	Action
Drag	Starts or ends the dragging of a control
DraggedObject	Returns the control being dragged

For more information about these events and functions, see the *PowerScript Reference*.

Identifying the dragged control

To identify the type of control that was dropped, use the source argument of the DragDrop event.

This script for the DragDrop event in a picture declares two variables, then determines the type of object that was dropped:

```
CommandButton lcb_button
StaticText lst_info

IF source.TypeOf() = CommandButton! THEN
      lcb_button = source
      lcb_button.Text = "You dropped a Button!"
ELSEIF source.TypeOf() = StaticText! THEN
      lst_info = source
      lst_info.Text = "You dropped the text!"
END IF
```

Using CHOOSE CASE

If your window has a large number of controls that can be dropped, use a CHOOSE CASE statement.

CHAPTER 11

Providing Online Help for an Application

About this chapter

This chapter describes how to provide online Help for other PowerBuilder developers and for end users on Windows.

Contents

Creating Help files

About Help authoring tools

There are many authoring tools and related products available for creating online Help files on Windows. All of the authoring tools for *RTF*-based Help files use the Microsoft Help compiler to compile a set of source files into a finished Help file.

What to include

The source files for any Help system typically include:

- *Topic files* (*RTF*), which contain the text of your Help system as well as footnote codes and commands that serve to identify the topics and provide navigation and other features
- *Graphics files*, typically bitmaps (*BMP*), which are associated with specific topics
- *A project file* (*HPJ*), which does not become part of the finished Help system but contains instructions for the compiler, some of which can affect the appearance and functionality of the Help windows you specify

- *A contents file* (*CNT*), which is not compiled into the finished Help file but is deployed along with the Help file as part of the overall Help system

  The contents file provides the entries that populate the Contents tab of the Help Topics dialog box and other information that can affect the overall behavior of the Help system.

How to proceed

If you are using one of the full-featured Help authoring tools, you can follow its instructions for creating the necessary source files and compiling them.

If you are using the Microsoft Help Workshop that comes with the Microsoft SDK, you can use the tool to create your project file and contents file and to compile the finished Help file. However, you will need to author your Help topics directly in Microsoft Word. The online Help for the Microsoft Help Workshop provides much information on Help authoring, including instructions on how to use the various footnote codes recognized by the Help compiler, the WinHelp macros, and the WinHelp API.

Sample project file

For your convenience, the text of a sample project file is provided in one of the topics of the *PBUSR110.HLP* file that is installed with PowerBuilder. To access it:

1. Open PowerBuilder Help and click the User button.
2. Navigate to the Sample Help Project File topic from the Contents tab in the Help Topics dialog box, under User-defined Help.
3. Copy the Help topic to the Windows clipboard.
4. Open a text editor such as Notepad and paste the clipboard text into a blank document.
5. Save the document in text format as *PBUSR110.HPJ*.

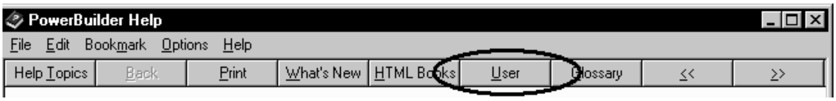
You need to edit your project file to reflect the details of your Help development environment, such as source file names and directory path names. You can do this in a text editor, or open the project file in Help Workshop or another Help authoring tool and edit it there.

Providing online Help for developers

Two ways to provide Help

There are two ways to integrate online Help for your user-defined functions, user events, and user objects into the PowerBuilder development environment:

- **The User button** In the PowerBuilder main Help window, the User button is hard-coded to launch a file named *PBUSR110.HLP*:



- **Context-sensitive Help** For user-defined functions, context-sensitive help can be displayed when you select the function name in the Script view (or place the cursor in the function name) and press Shift + F1.

How context-sensitive Help for user-defined functions works

When you select the name of a function or place the cursor in the function name in the Script view and press Shift + F1:

1. PowerBuilder looks for the standard prefix (the default is uf_) in the function name.
2. If the standard prefix is found, PowerBuilder looks for the Help topic in the Help file containing your user-defined function Help topics (instead of looking in *PBUSR110.HLP*, its own main Help file). The default file name for Help on user-defined functions is *PBUSR110.HLP*.

   PowerBuilder determines the name of the Help file to look in by reading the *UserHelpFile* variable in *PB.INI*. For information on changing the value of this variable, see “Advanced procedures” on page 152.
3. If PowerBuilder finds the variable, it looks in the specified Help file for the name of the selected function. If there is no *UserHelpFile* variable in *PB.INI*, PowerBuilder looks for the keyword in the *PBUSR110.HLP* file in the PowerBuilder *Help* directory.

Simplest approach

If you work within the PowerBuilder defaults, you must:

- Compile all of your online Help for your user-defined functions, user events, and user objects into a single file named *PBUSR110.HLP*

  You can optionally provide a contents file, which must be named *PBUSR110.CNT*.
- Prefix the name of each user-defined function you create with uf_ (for example, uf_calculate)

Basic procedures

Here are details on how to build online Help into the PowerBuilder environment.

❖ **To launch online Help for PowerBuilder developers from the User button:**

1. Create your online Help file using Microsoft Word and the Microsoft Help Workshop or other Help authoring tool.
2. Rename the *PBUSR110.HLP* and *PBUSR110.CNT* files that were installed with PowerBuilder. Be sure to rename the original *PBUSR110.CNT* file even if you do not provide your own contents file.
3. Save the compiled Help file and optional contents file in your PowerBuilder *Help* directory. Make sure your Help file is named *PBUSR110.HLP* and your contents file is named *PBUSR110.CNT*.

   Your Help file will display when you click the User button.

❖ **To create context-sensitive Help for user-defined functions:**

1. When you create a user-defined function, give the name of the function a standard prefix. The default prefix is uf_ (for example, uf_calculate).
2. For each user-defined function Help topic, assign a search keyword (a K footnote entry) identical to the function name.

   For example, in the Help topic for the user-defined function uf_CutBait, create a keyword footnote uf_CutBait. PowerBuilder uses the keyword to locate the correct topic to display in the Help window.
3. Compile the Help file and save it in the PowerBuilder *Help* directory.

Advanced procedures

You can specify a different file name for context-sensitive Help by changing the value of the *UserHelpFile* variable in your *PB.INI* file. To use the variable, your Help file must be in the PowerBuilder Help directory.

❖ **To specify a different file name for context-sensitive Help:**

1. Open your *PB.INI* file in a text editor.
2. In the [PB] section, add a *UserHelpFile* variable, specifying the name of the Help file that contains your context-sensitive topics. The format of the variable is:

   UserHelpFile = *helpfile.hlp*

   Specify only the file name. A full path name designation will not be recognized.

You can prefix your user-defined functions with a standard prefix other than the default uf_ prefix. You define the prefix you want to use by entering the *UserHelpPrefix* variable in your *PB.INI* file.

❖ **To use a different prefix for user-defined functions:**

1 Open your *PB.INI* file in a text editor.

2 In the [PB] section, add a *UserHelpPrefix* variable, specifying the value of your prefix. Use this format:

UserHelpPrefix = *yourprefix_*

The prefix you provide must end with the underscore character.

Providing online Help for users

Two ways to call Help from an application

PowerBuilder provides two principal ways of calling an online Help file from a PowerBuilder application:

- Use the ShowHelp and ShowPopupHelp PowerScript functions in your application scripts to call Help topics.
- Declare the WinHelp API as an external function and use the WinHelp function in your application scripts to call Help topics.

Using ShowHelp

ShowHelp is simpler to implement than the WinHelp API. You can use the ShowHelp PowerScript function to search for Help topics by Help context ID, by keyword, and by accessing the Help file Contents topic (the topic defined in the project file as the Help Contents topic). ShowHelp can also be used with compiled HTML (*.chm*) files.

ShowPopupHelp displays pop-up help for a control. Typically, you use ShowPopupHelp in the Help event of a response window with the Context Help property enabled. Events relating to movement of the cursor over a control or to the dragging of a control or object are also logical places for a ShowPopupHelp call.

For more information on the ShowHelp and ShowPopupHelp functions as well as the Help event, see the *PowerScript Reference*.

Using the WinHelp API

Declaring and using the WinHelp API allows access to the full range of WinHelp functions, many of which are not available in ShowHelp. For example, using the WinHelp function, you can easily specify a window type or window name in which to present a Help topic.

❖ **To declare the WinHelp API as an external function:**

1. Select Declare>Global External Functions from the menu bar of any painter that accesses the Script view.
2. Enter the function declaration in the text box and click OK.

   This example declares the WinHelp API:

   ```
   FUNCTION boolean WinHelpW(long hWndMain, &
         string lpszHelp, long uCommand, &
         long dwData) &
   LIBRARY "USER32.DLL"
   ```

For more information about the WinHelp API, see the online Help for the Microsoft Help Workshop or the documentation for your Help authoring tool. For more information about declaring and using global external functions, see the *PowerScript Reference* and “Using external functions” on page 421.

PART 4

Data Access Techniques

This part presents a collection of techniques you can use to implement data access features in the applications you develop with PowerBuilder. It includes using Transaction objects, XML processing, graphs, rich text, and piping of data between data sources. The use of DataWindow objects and DataStores for data access is described in the *DataWindow Programmer's Guide*.

CHAPTER 12 Using Transaction Objects

About this chapter

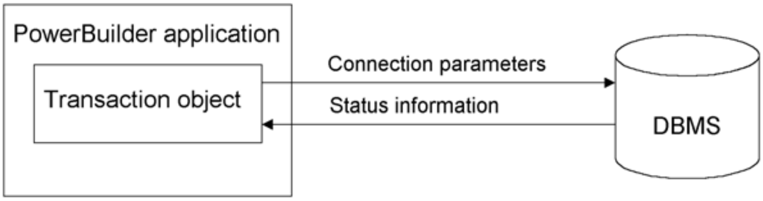
This chapter describes Transaction objects and how to use them in PowerBuilder applications.

Contents

About Transaction objects

In a PowerBuilder database connection, a **Transaction object** is a special nonvisual object that functions as the communications area between a PowerBuilder application and the database. The Transaction object specifies the parameters that PowerBuilder uses to connect to a database. You must establish the Transaction object before you can access the database from your application, as shown in Figure 12-1:

Figure 12-1: Transaction object to access database



Communicating with the database

In order for a PowerBuilder application to display and manipulate data, the application must communicate with the database in which the data resides.

❖ **To communicate with the database from your PowerBuilder application:**

1. Assign the appropriate values to the Transaction object.
2. Connect to the database.
3. Assign the Transaction object to the DataWindow control.
4. Perform the database processing.
5. Disconnect from the database.

For information about setting the Transaction object for a DataWindow control and using the DataWindow to retrieve and update data, see the *DataWindow Programmer's Guide*.

Default Transaction object

When you start executing an application, PowerBuilder creates a global default Transaction object named SQLCA (SQL Communications Area). You can use this default Transaction object in your application or define additional Transaction objects if your application has multiple database connections.

Transaction object properties

Each Transaction object has 15 properties, of which:

- Ten are used to connect to the database.
- Five are used to receive status information from the database about the success or failure of each database operation. (These error-checking properties all begin with SQL.)

Description of Transaction object properties

Table 12-1 describes each Transaction object property. For each of the ten connection properties, it also lists the equivalent field in the Database Profile Setup dialog box that you complete to create a database profile in the PowerBuilder development environment.

Transaction object properties for your PowerBuilder database interface
For the Transaction object properties that apply to your PowerBuilder database interface, see "Transaction object properties and supported PowerBuilder database interfaces" on page 160.

For information about the values you should supply for each connection property, see the section for your PowerBuilder database interface in *Connecting to Your Database*.

Table 12-1: Transaction object properties

Property	Datatype	Description	In a database profile
DBMS	String	The DBMS identifier for your connection. For a complete list of the identifiers for the supported database interfaces, see the online Help.	DBMS
Database	String	The name of the database to which you are connecting.	Database Name
UserID	String	The name or ID of the user who connects to the database.	User ID
DBPass	String	The password used to connect to the database.	Password
Lock	String	For those DBMSs that support the use of lock values and isolation levels, the isolation level to use when you connect to the database. For information about the lock values you can set for your DBMS, see the description of the Lock DBParm parameter in the online Help.	Isolation Level
LogID	String	The name or ID of the user who logs in to the database server.	Login ID
LogPass	String	The password used to log in to the database server.	Login Password
ServerName	String	The name of the server on which the database resides.	Server Name
AutoCommit	Boolean	For those DBMSs that support it, specifies whether PowerBuilder issues SQL statements outside or inside the scope of a transaction. Values you can set are: • **True** PowerBuilder issues SQL statements *outside* the scope of a transaction; that is, the statements are not part of a logical unit of work (LUW). If the SQL statement succeeds, the DBMS updates the database immediately as if a COMMIT statement had been issued. • **False** (Default) PowerBuilder issues SQL statements *inside* the scope of a transaction. PowerBuilder issues a BEGIN TRANSACTION statement at the start of the connection. In addition, PowerBuilder issues another BEGIN TRANSACTION statement after each COMMIT or ROLLBACK statement is issued. For more information, see the AutoCommit description in the online Help.	AutoCommit Mode
DBParm	String	Contains DBMS-specific connection parameters that support particular DBMS features. For a description of each DBParm parameter that PowerBuilder supports, see the chapter on setting additional connection parameters in *Connecting to Your Database*.	DBPARM
SQLReturnData	String	Contains DBMS-specific information. For example, after you connect to an Informix database and execute an embedded SQL INSERT statement, SQLReturnData contains the serial number of the inserted row.	—

Property	Datatype	Description	In a database profile
SQLCode	Long	The success or failure code of the most recent SQL operation. For details, see “Error handling after a SQL statement” on page 170.	—
SQLNRows	Long	The number of rows affected by the most recent SQL operation. The database vendor supplies this number, so the meaning may be different for each DBMS.	—
SQLDBCode	Long	The database vendor’s error code. For details, see “Error handling after a SQL statement” on page 170.	—
SQLErrText	String	The text of the database vendor’s error message corresponding to the error code. For details, see “Error handling after a SQL statement” on page 170.	—

Transaction object properties and supported PowerBuilder database interfaces

The Transaction object properties required to connect to the database are different for each PowerBuilder database interface. Except for SQLReturnData, the properties that return status information about the success or failure of a SQL statement apply to all PowerBuilder database interfaces.

Table 12-2 lists each supported PowerBuilder database interface and the Transaction object properties you can use with that interface.

Table 12-2: PowerBuilder database interfaces

Database interface	Transaction object properties	
Informix	DBMS	AutoCommit
	UserID	SQLReturnData
	DBPass	SQLCode
	Database	SQLNRows
	ServerName	SQLDBCode
	DBParm	SQLErrText
	Lock	
JDBC	DBMS	AutoCommit
	LogID	SQLCode
	LogPass	SQLNRows
	DBParm	SQLDBCode
	Lock	SQLErrText

Database interface	Transaction object properties	
Microsoft SQL Server	DBMS	AutoCommit
	Database	SQLCode
	ServerName	SQLNRows
	LogID	SQLDBCode
	LogPass	SQLErrText
	DBParm	
	Lock	
ODBC	DBMS	AutoCommit
	#UserID	SQLReturnData
	+LogID	SQLCode
	+LogPass	SQLNRows
	DBParm	SQLDBCode
	Lock	SQLErrText
OLE DB	DBMS	AutoCommit
	LogID	SQLCode
	LogPass	SQLNRows
	DBParm	SQLDBCode
		SQLErrText
Oracle	DBMS	SQLReturnData
	ServerName	SQLCode
	LogID	SQLNRows
	LogPass	SQLDBCode
	DBParm	SQLErrText
Sybase DirectConnect	DBMS	AutoCommit
	Database	SQLCode
	ServerName	SQLNRows
	LogID	SQLDBCode
	LogPass	SQLErrText
	DBParm	
	Lock	
Sybase Adaptive Server Enterprise	DBMS	AutoCommit
	Database	SQLCode
	ServerName	SQLNRows
	LogID	SQLDBCode
	LogPass	SQLErrText
	DBParm	
	Lock	

# UserID is optional for ODBC. (Be careful specifying the UserID property; it overrides the connection’s UserName property returned by the ODBC SQLGetInfo call.)

+ PowerBuilder uses the LogID and LogPass properties only if your ODBC driver does *not* support the SQL driver CONNECT call.

Working with Transaction objects

PowerBuilder uses a basic concept of database transaction processing called **logical unit of work** (**LUW**). LUW is synonymous with transaction. A **transaction** is a set of one or more SQL statements that forms an LUW. Within a transaction, all SQL statements must succeed or fail as one logical entity.

There are four PowerScript transaction management statements:

- COMMIT
- CONNECT
- DISCONNECT
- ROLLBACK

Transaction basics

CONNECT and DISCONNECT

A successful CONNECT starts a transaction, and a DISCONNECT terminates the transaction. All SQL statements that execute between the CONNECT and the DISCONNECT occur within the transaction.

Before you issue a CONNECT statement, the Transaction object must exist and you must assign values to all Transaction object properties required to connect to your DBMS.

COMMIT and ROLLBACK

When a COMMIT executes, all changes to the database since the start of the current transaction (or since the last COMMIT or ROLLBACK) are made permanent, and a new transaction is started. When a ROLLBACK executes, all changes since the start of the current transaction are undone and a new transaction is started.

When a transactional component is deployed to EAServer , COM+, or another application server, you can use the TransactionServer context object to control transactions. See “Transaction server deployment” on page 163.

AutoCommit setting

You can issue a COMMIT or ROLLBACK only if the AutoCommit property of the Transaction object is set to False (the default) and you have not already started a transaction using embedded SQL.

For more about AutoCommit, see “Description of Transaction object properties” on page 158.

Automatic COMMIT when disconnected
When a transaction is disconnected, PowerBuilder issues a COMMIT statement.

Transaction pooling

To optimize database processing, you can code your PowerBuilder application to take advantage of transaction pooling.

For information, see “Pooling database transactions” on page 171.

Transaction server deployment

Components that you develop in PowerBuilder can participate in transactions in EAServer, COM+, or other application servers. You can mark components to indicate that they will provide transaction support. When a component provides transaction support, the transaction server ensures that the component’s database operations execute as part of a transaction and that the database changes performed by the participating components are all committed or rolled back. By defining components to use transactions, you can ensure that all work performed by components that participate in a transaction occurs as intended.

PowerBuilder provides a transaction service context object called TransactionServer that gives you access to the transaction state primitives that influence whether the transaction server commits or aborts the current transaction. COM+ clients can also use the OleTxnObject object to control transactions. If you use the TransactionServer context object and set the UseContextObject DBParm parameter to Yes, COMMIT and ROLLBACK statements called in the Transaction object will result in a database error.

By default, the TransactionServer context object is not used. Instead you can use COMMIT and ROLLBACK statements to manage transactions. In this case, COMMIT is interpreted as a SetComplete function and ROLLBACK is interpreted as a SetAbort function.

For information, see “Providing support for transactions” on page 465 (for EAServer) and “Providing support for transactions” on page 568 (for COM+).

The default Transaction object

SQLCA

Since most applications communicate with only one database, PowerBuilder provides a global default Transaction object called SQLCA (SQL Communications Area).

PowerBuilder creates the Transaction object before the application’s Open event script executes. You can use PowerScript dot notation to reference the Transaction object in any script in your application.

You can create additional Transaction objects as you need them (such as when you are using multiple database connections at the same time). But in most cases, SQLCA is the only Transaction object you need.

Example

This simple example uses the default Transaction object SQLCA to connect to and disconnect from an ODBC data source named Sample:

```
// Set the default Transaction object properties.
SQLCA.DBMS="ODBC"
SQLCA.DBParm="ConnectString='DSN=Sample'"
// Connect to the database.
CONNECT USING SQLCA;
IF SQLCA.SQLCode < 0 THEN &
   MessageBox("Connect Error", SQLCA.SQLErrText,&
   Exclamation!)
...
// Disconnect from the database.
DISCONNECT USING SQLCA;
IF SQLCA.SQLCode < 0 THEN &
   MessageBox("Disconnect Error", SQLCA.SQLErrText,&
      Exclamation!)
```

Semicolons are SQL statement terminators
When you use embedded SQL in a PowerBuilder script, all SQL statements must be terminated with a semicolon (;). You do *not* use a continuation character for multiline SQL statements.

Assigning values to the Transaction object

Before you can use a default (SQLCA) or nondefault (user-defined) Transaction object, you must assign values to the Transaction object connection properties. To assign the values, use PowerScript dot notation.

Example

The following PowerScript statements assign values to the properties of SQLCA required to connect to a Sybase Adaptive Server Enterprise database through the PowerBuilder Adaptive Server Enterprise database interface:

```
sqlca.DBMS="SYC"
sqlca.database="testdb"
sqlca.LogId="CKent"
sqlca.LogPass="superman"
sqlca.ServerName="Dill"
```

Reading values from an external file

Using external files

Often you want to set the Transaction object values from an external file. For example, you might want to retrieve values from your PowerBuilder initialization file when you are developing the application or from an application-specific initialization file when you distribute the application.

ProfileString function

You can use the PowerScript ProfileString function to retrieve values from a text file that is structured into sections containing variable assignments, like a Windows *INI* file. The PowerBuilder initialization file is such a file, consisting of several sections including *PB*, *Application*, and *Database*:

```
[PB]
variables and their values
...
[Application]
variables and their values
...
[Database]
variables and their values
...
```

The ProfileString function has this syntax:

ProfileString (*file, section, key, default*)

Example

This script reads values from an initialization file to set the Transaction object to connect to a database. Conditional code sets the variable *startupfile* to the appropriate value for each platform:

```
sqlca.DBMS = ProfileString(startupfile, "database",&
   "dbms", "")
sqlca.database = ProfileString(startupfile,&
   "database", "database", "")
sqlca.userid = ProfileString(startupfile, "database",&
   "userid", "")
sqlca.dbpass = ProfileString(startupfile, "database",&
   "dbpass", "")
sqlca.logid = ProfileString(startupfile, "database",&
   "logid", "")
sqlca.logpass = ProfileString(startupfile, "database",&
   "LogPassWord","")
sqlca.servername = ProfileString(startupfile,&
   "database", "servername","")
sqlca.dbparm = ProfileString(startupfile, "database",&
   "dbparm", "")
```

Connecting to the database

Once you establish the connection parameters by assigning values to the Transaction object properties, you can connect to the database using the SQL CONNECT statement:

```
// Transaction object values have been set.
CONNECT;
```

Because CONNECT is a SQL statement—not a PowerScript statement—you need to terminate it with a semicolon.

If you are using a Transaction object other than SQLCA, you *must* include the USING TransactionObject clause in the SQL syntax:

CONNECT USING *TransactionObject*;

For example:

```
CONNECT USING ASETrans;
```

Using the Preview tab to connect in a PowerBuilder application

The Preview tab page in the Database Profile Setup dialog box makes it easy to generate accurate PowerScript connection syntax in the development environment for use in your PowerBuilder application script.

As you complete the Database Profile Setup dialog box, the correct PowerScript connection syntax for each selected option is generated on the Preview tab. PowerBuilder assigns the corresponding DBParm parameter or SQLCA property name to each option and inserts quotation marks, commas, semicolons, and other characters where needed. You can copy the syntax you want from the Preview tab directly into your script.

❖ **To use the Preview tab to connect in a PowerBuilder application:**

1. In the Database Profile Setup dialog box for your connection, supply values for basic options (on the Connection tab) and additional DBParm parameters and SQLCA properties (on the other tabbed pages) as required by your database interface.

   For information about connection parameters for your interface and the values you should supply, click Help.

2. Click Apply to save your settings without closing the Database Profile Setup dialog box.

3 Click the Preview tab.

The correct PowerScript connection syntax for each selected option displays in the Database Connection Syntax box on the Preview tab.

4 Select one or more lines of text in the Database Connection Syntax box and click Copy.

PowerBuilder copies the selected text to the clipboard. You can then paste this syntax into your script, modifying the default Transaction object name (SQLCA) if necessary.

5 Click OK.

Disconnecting from the database

When your database processing is completed, you disconnect from the database using the SQL DISCONNECT statement:

```
DISCONNECT;
```

If you are using a Transaction object other than SQLCA, you *must* include the USING TransactionObject clause in the SQL syntax:

DISCONNECT USING *TransactionObject*;

For example:

```
DISCONNECT USING ASETrans;
```

Automatic COMMIT when disconnected
When a transaction is disconnected, PowerBuilder issues a COMMIT statement by default.

Defining Transaction objects for multiple database connections

Use one Transaction object per connection

To perform operations in multiple databases at the same time, you need to use multiple Transaction objects, one for each database connection. You must declare and create the additional Transaction objects before referencing them, and you must destroy these Transaction objects when they are no longer needed.

Caution
PowerBuilder creates and destroys SQLCA automatically. Do not attempt to create or destroy it.

Creating the nondefault Transaction object

To create a Transaction object other than SQLCA, you first declare a variable of type transaction:

transaction *TransactionObjectName*

You then instantiate the object:

TransactionObjectName = CREATE transaction

For example, to create a Transaction object named DBTrans, code:

```
transaction DBTrans
DBTrans = CREATE transaction
// You can now assign property values to DBTrans.
DBTrans.DBMS = "ODBC"
...
```

Assigning property values

When you assign values to properties of a Transaction object that you declare and create in a PowerBuilder script, you *must* assign the values *one property at a time*, like this:

```
// This code produces correct results.
transaction ASETrans
ASETrans = CREATE TRANSACTION
ASETrans.DBMS = "Sybase"
ASETrans.Database = "Personnel"
```

You *cannot* assign values by setting the nondefault Transaction object equal to SQLCA, like this:

```
// This code produces incorrect results.
transaction  ASETrans
ASETrans = CREATE TRANSACTION
ASETrans = SQLCA // ERROR!
```

Specifying the Transaction object in SQL statements

When a database statement requires a Transaction object, PowerBuilder assumes the Transaction object is SQLCA unless you specify otherwise. These CONNECT statements are equivalent:

```
CONNECT;
CONNECT USING SQLCA;
```

However, when you use a Transaction object *other* than SQLCA, you *must* specify the Transaction object in the SQL statements in Table 12-3 with the USING TransactionObject clause.

Table 12-3: SQL statements that require USING TransactionObject

COMMIT	INSERT
CONNECT	PREPARE (dynamic SQL)
DELETE	ROLLBACK
DECLARE Cursor	SELECT
DECLARE Procedure	SELECTBLOB
DISCONNECT	UPDATEBLOB
EXECUTE (dynamic SQL)	UPDATE

❖ To specify a user-defined Transaction object in SQL statements:

- Add the following clause to the end of any of the SQL statements in the preceding list:

  USING *TransactionObject*

  For example, this statement uses a Transaction object named ASETrans to connect to the database:

  ```
  CONNECT USING ASETrans;
  ```

Always code the Transaction object
Although specifying the USING TransactionObject clause in SQL statements is optional when you use SQLCA and required when you define your own Transaction object, it is good practice to code it for any Transaction object, including SQLCA. This avoids confusion and ensures that you supply USING TransactionObject when it is required.

Example

The following statements use the default Transaction object (SQLCA) to communicate with a SQL Anywhere database and a nondefault Transaction object named ASETrans to communicate with an Adaptive Server Enterprise database:

```
// Set the default Transaction object properties.
SQLCA.DBMS = "ODBC"
SQLCA.DBParm = "ConnectString='DSN=Sample'"
// Connect to the SQL Anywhere database.
CONNECT USING SQLCA;
// Declare the ASE Transaction object.
transaction ASETrans
// Create the ASE Transaction object.
ASETrans = CREATE TRANSACTION
// Set the ASE Transaction object properties.
ASETrans.DBMS = "Sybase"
ASETrans.Database = "Personnel"
```

```
ASETrans.LogID = "JPL"
ASETrans.LogPass = "JPLPASS"
ASETrans.ServerName = "SERVER2"

// Connect to the ASE database.
CONNECT USING ASETrans;

// Insert a row into the SQL Anywhere database
INSERT INTO CUSTOMER
VALUES ( 'CUST789', 'BOSTON' )
USING SQLCA;
// Insert a row into the ASE database.
INSERT INTO EMPLOYEE
VALUES ( "Peter Smith", "New York" )
USING ASETrans;

// Disconnect from the SQL Anywhere database
DISCONNECT USING SQLCA;
// Disconnect from the ASE database.
DISCONNECT USING ASETrans;
// Destroy the ASE Transaction object.
DESTROY ASETrans
```

Using error checking
An actual script would include error checking after the CONNECT, INSERT, and DISCONNECT statements.

For details, see "Error handling after a SQL statement" next.

Error handling after a SQL statement

When to check for errors

You should always test the success or failure code (the SQLCode property of the Transaction object) after issuing one of the following statements in a script:

- Transaction management statement (such as CONNECT, COMMIT, and DISCONNECT)
- Embedded or dynamic SQL

Not in DataWindows
Do *not* do this type of error checking following a retrieval or update made in a DataWindow.

For information about handling errors in DataWindows, see the *DataWindow Programmer's Guide*.

SQLCode return values

Table 12-4 shows the SQLCode return values.

Table 12-4: SQLCode return values

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

Using SQLErrText and SQLDBCode

The string SQLErrText in the Transaction object contains the database vendor-supplied error message. The long named SQLDBCode in the Transaction object contains the database vendor-supplied status code. You can reference these variables in your script.

Example To display a message box containing the DBMS error number and message if the connection fails, code the following:

```
CONNECT USING SQLCA;
IF SQLCA.SQLCode = -1 THEN
   MessageBox("SQL error " + String(SQLCA.SQLDBCode),&
   SQLCA.SQLErrText )
END IF
```

Pooling database transactions

Transaction pooling

To optimize database processing, an application can pool database transactions. **Transaction pooling** maximizes database throughput while controlling the number of database connections that can be open at one time. When you establish a transaction pool, an application can reuse connections made to the same data source.

How it works

When an application connects to a database *without using transaction pooling*, PowerBuilder physically terminates each database transaction for which a DISCONNECT statement is issued.

When transaction pooling is in effect, PowerBuilder logically terminates the database connections and commits any database changes, but does not physically remove them. Instead, the database connections are kept open in the transaction pool so that they can be reused for other database operations.

When to use it

Transaction pooling can enhance the performance of an application that services a high volume of short transactions to the same data source.

How to use it

To establish a transaction pool, you use the SetTransPool function. You can code SetTransPool anywhere in your application, as long as it is executed before the application connects to the database. A logical place to execute SetTransPool is in the application Open event.

Example

This statement specifies that up to 16 database connections will be supported through this application, and that 12 connections will be kept open once successfully connected. When the maximum number of connections has been reached, each subsequent connection request will wait for up to 10 seconds for a connection in the pool to become available. After 10 seconds, the application will return an error:

```
myapp.SetTransPool (12,16,10)
```

For more information

For more information about the SetTransPool function, see the *PowerScript Reference*.

Using Transaction objects to call stored procedures

SQLCA is a built-in global variable of type transaction that is used in all PowerBuilder applications. In your application, you can define a specialized version of SQLCA that performs certain processing or calculations on your data.

If your database supports stored procedures, you might already have defined remote stored procedures to perform these operations. You can use the remote procedure call (RPC) technique to define a customized version of the Transaction object that calls these database stored procedures in your application.

Result sets
You *cannot* use the RPC technique to access result sets returned by stored procedures. If the stored procedure returns one or more result sets, PowerBuilder ignores the values and returns the output parameters and return value. If your stored procedure returns a result set, you can use the embedded SQL DECLARE Procedure statement to call it.

For information about the DECLARE Procedure statement, see the chapter on SQL statements in the *PowerScript Reference*.

Overview of the RPC procedure

To call database stored procedures from within your PowerBuilder application, you can use the remote procedure call technique and PowerScript dot notation (*object.function*) to define a customized version of the Transaction object that calls the stored procedures.

❖ **To call database stored procedures in your application:**

1. From the Objects tab in the New dialog box, define a standard class user object inherited from the built-in Transaction object.
2. In the Script view in the User Object painter, use the RPCFUNC keyword to declare the stored procedure as an external function or subroutine for the user object.
3. Save the user object.
4. In the Application painter, specify the user object you defined as the default global variable type for SQLCA.
5. Code your PowerBuilder application to use the user object.

For instructions on using the User Object and Application painters and the Script view in PowerBuilder, see the *PowerBuilder User's Guide*.

Understanding the example

u_trans_database user object The following sections give step-by-step instructions for using a Transaction object to call stored procedures in your application. The example shows how to define and use a standard class user object named u_trans_database.

The u_trans_database user object is a descendant of (inherited from) the built-in Transaction object SQLCA. A **descendant** is an object that inherits functionality (properties, variables, functions, and event scripts) from an ancestor object. A descendent object is also called a **subclass**.

GIVE_RAISE stored procedure The u_trans_database user object calls an Oracle database stored procedure named GIVE_RAISE that calculates a five percent raise on the current salary. Here is the Oracle syntax to create the GIVE_RAISE stored procedure:

SQL terminator character
The syntax shown here for creating an Oracle stored procedure assumes that the SQL statement terminator character is ` (backquote).

```
// Create GIVE_RAISE function for Oracle
// SQL terminator character is ` (backquote).
CREATE OR REPLACE FUNCTION give_raise
(salary IN OUT NUMBER)
return NUMBER
IS rv NUMBER;
BEGIN
   salary := salary * 1.05;
   rv := salary;
   return rv;
END; `
// Save changes.
COMMIT WORK`
// Check for errors.
SELECT * FROM all_errors`
```

Step 1: define the standard class user object

❖ **To define the standard class user object:**

1. Start PowerBuilder.
2. Connect to a database that supports stored procedures.

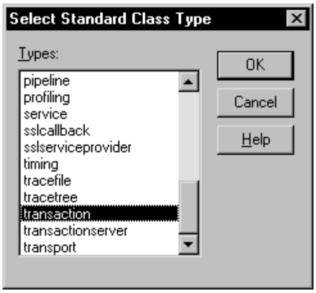
   The rest of this procedure assumes you are connected to an Oracle database that contains remote stored procedures on the database server.

   For instructions on connecting to an Oracle database in PowerBuilder and using Oracle stored procedures, see *Connecting to Your Database*.
3. Click the New button in the PowerBar, or select File>New from the menu bar.

   The New dialog box displays.

4 On the Object tab, select the Standard Class icon and click OK to define a new standard class user object.

The Select Standard Class Type dialog box displays:



5 Select *transaction* as the built-in system type that you want your user object to inherit from, and click OK.

The User Object painter workspace displays so that you can assign properties (instance variables) and functions to your user object:

Step 2: declare the stored procedure as an external function

FUNCTION or SUBROUTINE declaration

You can declare a non-result-set database stored procedure as an external function or external subroutine in a PowerBuilder application. If the stored procedure has a return value, declare it as a function (using the FUNCTION keyword). If the stored procedure returns nothing or returns VOID, declare it as a subroutine (using the SUBROUTINE keyword).

RPCFUNC and ALIAS FOR keywords

You *must* use the RPCFUNC keyword in the function or subroutine declaration to indicate that this is a remote procedure call (RPC) for a database stored procedure rather than for an external function in a dynamic library. Optionally, you can use the ALIAS FOR "*spname*" expression to supply the name of the stored procedure as it appears in the database if this name differs from the one you want to use in your script.

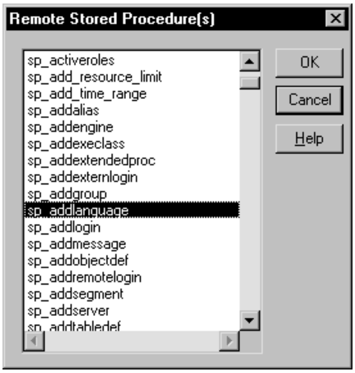
For complete information about the syntax for declaring stored procedures as remote procedure calls, see the chapter on calling functions and events in the *PowerScript Reference*.

❖ **To declare stored procedures as external functions for the user object:**

1 In the Script view in the User Object painter, select [Declare] from the first list and Local External Functions from the second list.

2 Place your cursor in the Declare Local External Functions view. From the pop-up menu or the Edit menu, select Paste Special>SQL>Remote Stored Procedures.

PowerBuilder loads the stored procedures from your database and displays the Remote Stored Procedures dialog box. It lists the names of stored procedures in the current database.



3 Select the names of one or more stored procedures that you want to declare as functions for the user object, and click OK.

PowerBuilder retrieves the stored procedure declarations from the database and pastes each declaration into the view.

For example, here is the declaration that displays on one line when you select sp_addlanguage:

```
function long sp_addlanguage()
RPCFUNC ALIAS FOR "dbo.sp_addlanguage"
```

4 Edit the stored procedure declaration as needed for your application.

Use either of the following syntax formats to declare the database remote procedure call (RPC) as an external function or external subroutine (for details about the syntax, see the *PowerScript Reference*):

FUNCTION *rtndatatype functionname* ({ { **REF** } *datatype1 arg1*, ...,
{ **REF** } *datatypen argn* }) **RPCFUNC** { **ALIAS FOR** "*spname*" }

SUBROUTINE *functionname* ({ { **REF** } *datatype1 arg1* , ...,
{ **REF** } *datatypen argn* }) **RPCFUNC** { **ALIAS FOR** "*spname*" }

Here is the edited RPC function declaration for sp_addlanguage:

```
FUNCTION long sp_addlanguage()
   RPCFUNC ALIAS FOR "addlanguage_proc"
```

Step 3: save the user object

❖ **To save the user object:**

1. In the User Object painter, click the Save button, or select File>Save from the menu bar.

   The Save User Object dialog box displays.

2. Specify the name of the user object, comments that describe its purpose, and the library in which to save the user object.

3. Click OK to save the user object.

   PowerBuilder saves the user object with the name you specified in the selected library.

Step 4: specify the default global variable type for SQLCA

In the Application painter, you must specify the user object you defined as the default global variable type for SQLCA. When you execute your application, this tells PowerBuilder to use your standard class user object instead of the built-in system Transaction object.

Using your own Transaction object instead of SQLCA
This procedure assumes that your application uses the default Transaction object SQLCA, but you can also declare and create an instance of your own Transaction object and then write code that calls the user object as a property of your Transaction object. For instructions, see the chapter on working with user objects in the *PowerBuilder User’s Guide*.

❖ **To specify the default global variable type for SQLCA:**

1. Click the Open button in the PowerBar, or select File>Open from the menu bar.

   The Open dialog box displays.

2. Select Applications from the Object Type drop-down list. Choose the application where you want to use your new user object and click OK.

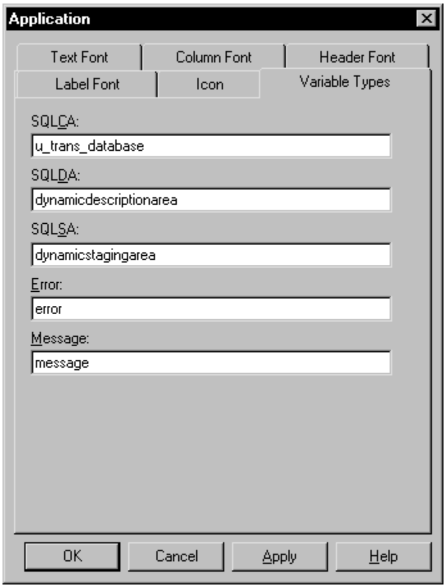
   The Application painter workspace displays.

3 Select the General tab in the Properties view. Click the Additional Properties button.

The Additional Properties dialog box displays.

4 Click the Variable Types tab to display the Variable Types property page.

5 In the SQLCA box, specify the name of the standard class user object you defined in Steps 1 through 3:



SQLCA: u_trans_database

SQLDA: dynamicdescriptionarea

SQLSA: dynamicstagingarea

Error: error

Message: message

6 Click OK or Apply.

When you execute your application, PowerBuilder will use the specified standard class user object instead of the built-in system object type it inherits from.

Step 5: code your application to use the user object

What you have done so far In the previous steps, you defined the GIVE_RAISE remote stored procedure as an external function for the u_trans_database standard class user object. You then specified u_trans_database as the default global variable type for SQLCA. These steps give your PowerBuilder application access to the properties and functions encapsulated in the user object.

What you do now You now need to write code that uses the user object to perform the necessary processing.

In your application script, you can use PowerScript dot notation to call the stored procedure functions you defined for the user object, just as you do when using SQLCA for all other PowerBuilder objects. The dot notation syntax is:

object.function (*arguments*)

For example, you can call the GIVE_RAISE stored procedure with code similar to the following:

```
SQLCA.give_raise(salary)
```

❖ **To code your application to use the user object:**

1 Open the object or control for which you want to write a script.

2 Select the event for which you want to write the script.

For instructions on using the Script view, see the *PowerBuilder User's Guide*.

3 Write code that uses the user object to do the necessary processing for your application.

Here is a simple code example that connects to an Oracle database, calls the GIVE_RAISE stored procedure to calculate the raise, displays a message box with the new salary, and disconnects from the database:

```
// Set Transaction object connection properties.
SQLCA.DBMS="OR7"
SQLCA.LogID="scott"
SQLCA.LogPass="xxyyzz"
SQLCA.ServerName="@t:oracle:testdb"
SQLCA.DBParm="sqlcache=24,pbdbms=1"

// Connect to the Oracle database.
CONNECT USING SQLCA ;

// Check for errors.
IF SQLCA.sqlcode <> 0 THEN
      MessageBox ("Connect Error",SQLCA.SQLErrText)
      return
END IF

// Set 20,000 as the current salary.
DOUBLE val = 20000
DOUBLE rv
```

```
// Call the GIVE_RAISE stored procedure to
// calculate the raise.
// Use dot notation to call the stored procedure
rv = SQLCA.give_raise(val)

// Display a message box with the new salary.
MessageBox("The new salary is",string(rv))

// Disconnect from the Oracle database.
DISCONNECT USING SQLCA;
```

4 Compile the script to save your changes.

Using error checking
An actual script would include error checking after the CONNECT statement, DISCONNECT statement, and call to the GIVE_RAISE procedure. For details, see "Error handling after a SQL statement" on page 170.

Supported DBMS features when calling stored procedures

When you define and use a custom Transaction object to call remote stored procedures in your application, the features supported depend on the DBMS to which your application connects.

The following sections describe the supported features for some of the DBMSs that you can access in PowerBuilder. Read the section for your DBMS to determine what you can and cannot do when using the RPC technique in a PowerBuilder application.

Result sets
You *cannot* use the remote procedure call technique to access result sets returned by stored procedures. If the stored procedure returns one or more result sets, PowerBuilder ignores the values and returns the output parameters and return value.

If your stored procedure returns a result set, you can use the embedded SQL DECLARE Procedure statement to call it. For information about the DECLARE Procedure statement, see the chapter on SQL statements in the *PowerScript Reference*.

Informix

If your application connects to an Informix database, you can use simple nonarray datatypes. You *cannot* use binary large objects (blobs).

ODBC

If your application connects to an ODBC data source, you can use the following ODBC features if the back-end driver supports them. (For information, see the documentation for your ODBC driver.)

- IN, OUT, and IN OUT parameters, as shown in Table 12-5.

Table 12-5: ODBC IN, OUT, and IN OUT parameters

Parameter	What happens
IN	An IN variable is passed by value and indicates a value being passed to the procedure.
OUT	An OUT variable is passed by reference and indicates that the procedure can modify the PowerScript variable that was passed. Use the PowerScript REF keyword for this parameter type.
IN OUT	An IN OUT variable is passed by reference and indicates that the procedure can reference the passed value and can modify the PowerScript variable. Use the PowerScript REF keyword for this parameter type.

- Blobs as parameters. You can use blobs that are up to 32,512 bytes long.
- Integer return codes.

Oracle

If your application connects to an Oracle database, you can use the following Oracle PL/SQL features:

- IN, OUT, and IN OUT parameters, as shown in Table 12-6.

Table 12-6: Oracle IN, OUT, and IN OUT parameters

Parameter	What happens
IN	An IN variable is passed by value and indicates a value being passed to the procedure.
OUT	An OUT variable is passed by reference and indicates that the procedure can modify the PowerScript variable that was passed. Use the PowerScript REF keyword for this parameter type.
IN OUT	An IN OUT variable is passed by reference and indicates that the procedure can reference the passed value and can modify the PowerScript variable. Use the PowerScript REF keyword for this parameter type.

- Blobs as parameters. You can use blobs that are up to 32,512 bytes long.
- PL/SQL tables as parameters. You can use PowerScript arrays.
- Function return codes.

Microsoft SQL Server or Sybase Adaptive Server Enterprise

If your application connects to a Microsoft SQL Server or Sybase Adaptive Server Enterprise database, you can use the following Transact-SQL features:

- IN, OUT, and IN OUT parameters, as shown in Table 12-7.

Table 12-7: Adaptive Server Enterprise and Microsoft SQL Server IN, OUT, and IN OUT parameters

Parameter	What happens
IN	An IN variable is passed by value and indicates a value being passed to the procedure.
OUT	An OUT variable is passed by reference and indicates that the procedure can modify the PowerScript variable that was passed. Use the PowerScript REF keyword for this parameter type.
IN OUT	An IN OUT variable is passed by reference and indicates that the procedure can reference the passed value and can modify the PowerScript variable. Use the PowerScript REF keyword for this parameter type.

- Blobs as parameters. You can use blobs that are up to 32,512 bytes long.
- Integer return codes.

SQL Anywhere

If your application connects to a SQL Anywhere database, you can use the following SQL Anywhere features:

- IN, OUT, and IN OUT parameters, as shown in Table 12-8.

Table 12-8: SQL Anywhere IN, OUT, and IN OUT parameters

Parameter	What happens
IN	An IN variable is passed by value and indicates a value being passed to the procedure.
OUT	An OUT variable is passed by reference and indicates that the procedure can modify the PowerScript variable that was passed. Use the PowerScript REF keyword for this parameter type.
IN OUT	An IN OUT variable is passed by reference and indicates that the procedure can reference the passed value and can modify the PowerScript variable. Use the PowerScript REF keyword for this parameter type.

- Blobs as parameters. You can use blobs that are up to 32,512 bytes long.

CHAPTER 13 Using MobiLink Synchronization

About this chapter

This chapter supplements the introduction to MobiLink synchronization presented in the database management chapter of the *User's Guide*. It provides additional background on the synchronization process and the use of objects generated by the MobiLink synchronization wizard. It also discusses how to create synchronization objects without using the wizard.

Contents

About MobiLink synchronization

MobiLink is a session-based synchronization system that allows two-way synchronization between a main database, called the consolidated database, and many remote databases.

This section introduces some MobiLink terms and concepts.

Where to find additional information

Detailed information about MobiLink synchronization is provided in the *MobiLink Getting Started*, the *MobiLink - Client Administration*, and the *Mobilink - Server Administration* books. These books are available online on the SQL Anywhere Product Manuals Web site at http://www.ianywhere.com/developer/product_manuals/sqlanywhere/.

If you are already familiar with MobiLink, go to “Working with PowerBuilder synchronization objects” on page 189 to learn about PowerBuilder integration with MobiLink.

Data movement and synchronization

Data movement occurs when shared data is distributed over multiple databases on multiple nodes and changes to data in one database are applied to the corresponding data in other databases. Data can be moved using replication or synchronization.

Data replication moves all transactions from one database to another, whereas data synchronization moves only the net result of transactions. Both techniques get their information by scanning transaction log files, but synchronization uses only updated log file segments instead of the full log file, making data movement much faster and more efficient.

With synchronization, data is available locally and can be modified without a connection to a server. MobiLink synchronization uses a loose consistency model, which means that all changes are synchronized with each site over time in a consistent manner, but different sites might have different copies of data at any instant. Only successful transactions are synchronized.

Consolidated and remote databases

The consolidated database, which can be any ODBC-compliant database, such as SQL Anywhere, Sybase Adaptive Server Enterprise, Oracle, IBM DB2 UDB, or Microsoft SQL Server, holds the master copy of all the data.

The remote database contains a subset of the consolidated data. Although MobiLink can synchronize SQL Anywhere and UltraLite databases, for PowerBuilder 11.0 applications, remote databases must be SQL Anywhere databases.

The MobiLink synchronization server

The MobiLink synchronization server, mlsrv10, manages the synchronization process and provides the interface between remote databases and the consolidated database server. All communication between the MobiLink synchronization server and the consolidated database occurs through an ODBC connection.The consolidated database and synchronization server often reside on the same machine, but that is not a requirement.

The MobiLink server must be running before a synchronization process is launched. You can start the MobiLink synchronization server from the Utilities folder in the Objects view in the Database painter.

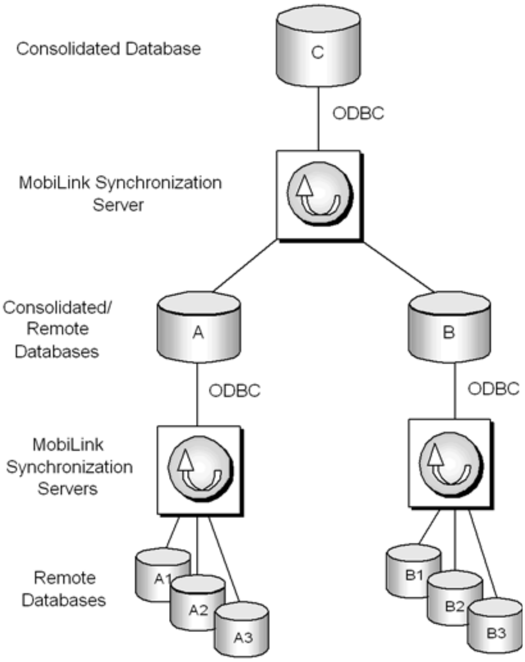
For information about starting the server from the command line, see “Running the MobiLink server” in the online *MobiLink - Server Administration* book.

MobiLink hierarchy

MobiLink typically uses a hierarchical configuration. The nodes in the hierarchy can reside on servers, desktop computers, and handheld or embedded devices. A simple hierarchy might consist of a consolidated database on a server and multiple remote databases on mobile devices. A more complex hierarchy might contain multiple levels in which some sites act as both remote and consolidated databases. For PowerBuilder applications, any consolidated database that also acts as a remote database must be a SQL Anywhere database.

For example, suppose remote sites A1, A2, and A3 synchronize with a consolidated database A on a local server, and remote sites B1, B2, and B3 synchronize with a consolidated database B on another local server. A and B in turn act as remote sites and synchronize with a consolidated database C on a master server. C can be any ODBC-compliant database, but A and B must both be SQL Anywhere databases.

Figure 13-1: MobiLink hierarchy


Consolidated Database
C
ODBC
MobiLink Synchronization Server
Consolidated/ Remote Databases
A
B
ODBC
ODBC
MobiLink Synchronization Servers
Remote Databases
A1
A2
A3
B1
B2
B3

Synchronization scripts

MobiLink synchronization is an event-driven process. When a MobiLink client initiates a synchronization, a number of synchronization events occur inside the MobiLink server. When an event occurs, MobiLink looks for a script to match the synchronization event. If you want the MobiLink server to take an action, you must provide a script for the event.

You can write synchronization scripts for connection-level events and for events for each table in the remote database. You save these scripts in the consolidated database.

You can write scripts using SQL, Java, or .NET. For more information about event scripts and writing them in the MobiLink Synchronization plug-in in Sybase Central, see “Preparing consolidated databases” on page 200.

The MobiLink synchronization client

SQL Anywhere clients at remote sites initiate synchronization by running a command-line utility called dbmlsync. This utility synchronizes one or more subscriptions in a remote database with the MobiLink synchronization server. Subscriptions are described in “Publications, articles, users, and subscriptions” next. For more information about the dbmlsync utility and its options, see “dbmlsync utility” in the index of the SQL Anywhere online books.

In PowerBuilder, synchronization objects that you create with the ASA MobiLink Synchronization wizard manage the dbmlsync process. For more information, see “Working with PowerBuilder synchronization objects” on page 189.

Publications, articles, users, and subscriptions

A publication is a database object on the remote database that identifies tables and columns to be synchronized. Each publication can contain one or more articles. An article is a database object that represents a whole table, or a subset of the columns and rows in a table.

A user is a database object in the remote database describing a unique synchronization client. There is one MobiLink user name for each remote database in the MobiLink system. The ml_user MobiLink system table, located in the consolidated database, holds a list of MobiLink user names. These names are used for authentication.

A subscription associates a user with one or more publications. It specifies the synchronization protocol (such as TCP/IP, HTTP, or HTTPS), address (such as *myserver.acmetools.com*), and additional optional connection and extended options.

Users, publications, and subscriptions are created in the remote database. You can create them in Sybase Central with the SQL Anywhere plug-in (not the MobiLink Synchronization plug-in). For information about creating users, publications, and subscriptions, see “Creating remote databases” on page 207.

The synchronization process

Dbmlsync connects to the remote database using TCP/IP, HTTP, or HTTPS, and prepares a stream of data (the upload stream) to be uploaded to the consolidated database. Dbmlsync uses information contained in the transaction log of the remote database to build the upload stream. The upload stream contains the MobiLink user name and password, the version of synchronization scripts to use, the last synchronization timestamp, the schema of tables and columns in the publication, and the net result of all inserts, updates, and deletes since the last synchronization.

After building the upload stream, dbmlsync uses information stored in the specified publication and subscription to connect to the MobiLink synchronization server and to exchange data.

When the MobiLink synchronization server receives data, it updates the consolidated database, then builds a download stream that contains all relevant changes and sends it back to the remote site. At the end of each successful synchronization, the consolidated and remote databases are consistent. Either a whole transaction is synchronized, or none of it is synchronized. This ensures transactional integrity at each database.

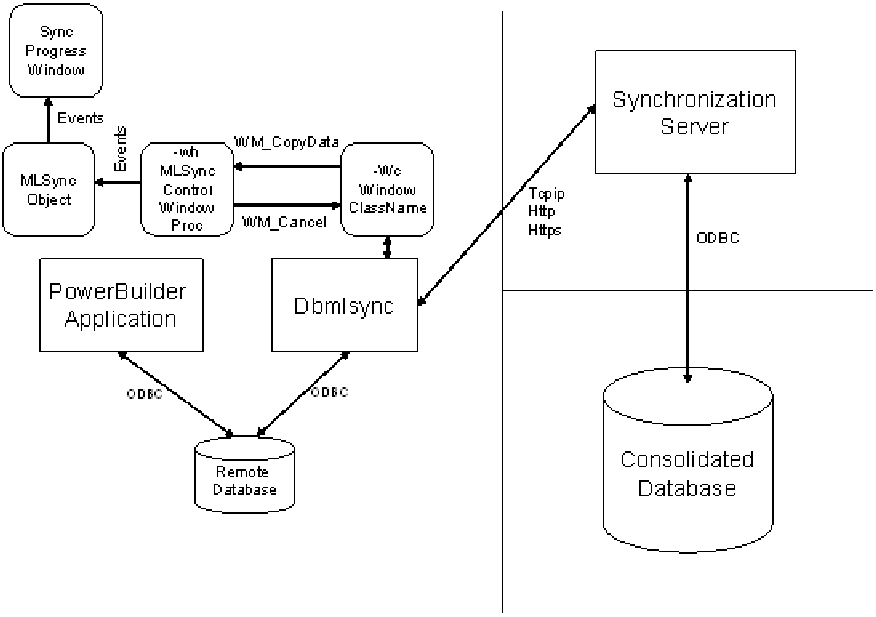
How the synchronization works

How MLSync events are implemented

The MLSync object in a PowerBuilder application and the dbmlsync process communicate with each other by sending messages between two windows, as shown in Figure 13-2. The window that the MLSync object creates uses an internal function, MlSyncControlWindowProc, to process these messages.

The Synchronize function adds a “-wh window_handle” argument to the end of the command line string that launches dbmlsync. This lets dbmlsync send WM_COPYDATA messages to this window handle. MlSyncControlWindowProc then triggers the appropriate event in the MLSync object.

Figure 13-2: How the synchronization process works



How progress window events are triggered

The MobiLink Synchronization Wizard generates an instance of an MLSync object that contains PowerScript code in each of its events. When appropriate, this code triggers an event of the same name in the progress window that is either generated by the wizard or customized for your applications.

How the CancelSync function is implemented

On the dbmlsync command string, there is a “-wc window_class” argument that specifies the class name of a communications window that dbmlsync registers and creates. If the PowerBuilder application needs to cancel the synchronization process during any of its event processing logic, it calls CancelSync. This function finds the window handle associated with the -wc window class and sends a WM_CLOSE message.

Working with PowerBuilder synchronization objects

When you run the ASA MobiLink Synchronization wizard from the Database page in the New dialog box, the wizard generates objects that let you initiate and control MobiLink synchronization requests from a PowerBuilder application. These objects let you obtain feedback during the synchronization process, code PowerScript events at specific points during synchronization, and cancel the process programmatically.

For more information about the MobiLink synchronization wizard, see “Managing the Database” in the *User’s Guide*.

Preparing to use the wizard

Before you use the wizard in a production application, you need to complete the following tasks:

- Set up a consolidated database and write synchronization scripts as described in “Preparing consolidated databases” on page 200
- Create a remote database on the desktop and set up one or more publications, users, and subscriptions as described in “Creating remote databases” on page 207
- Register the database with the ODBC manager on all remote machines, or create a file DSN for the remote database, as described in *Connecting to Your Database* in the PowerBuilder online Help and in “Using a file DSN instead of a registry DSN” on page 199
- Make sure all remote machines have the required supporting files, as described in “Runtime requirements for synchronization on remote machines” on page 197
- (Optional) Create a database connection profile for the remote database, as described in *Connecting to Your Database* in the PowerBuilder online Help. This allows the wizard to retrieve a list of publications in the remote database for which MobiLink subscriptions have been entered

What gets generated

The wizard generates two sets of objects.

Objects that initiate and monitor synchronization

The first set of objects lets the end user initiate and monitor synchronization:

- nvo_*appname*_mlsync – a custom class user object that controls the MobiLink client (*appname* is the name of your application)
- gf_*appname*_sync – a global function that instantiates the user object and calls a function to launch a synchronization request
- w_*appname*_syncprogress – an optional status window that reports the progress of the synchronization process

In the wizard, you can choose whether the application uses the status window. The generated status window includes an OK button that lets the user view the status before dismissing the window, and a Cancel button that lets the user cancel synchronization before it completes. You can also customize the window to fit your application's needs.

Objects that modify synchronization options

The second set of objects is generated only if you select Prompt User for Password and Runtime Changes in the wizard. It lets the end user change synchronization options before initiating synchronization:

- w_*appname*_sync_options – an options window that lets the end user modify the MobiLink user name and password, the host name and port of the MobiLink server, and other options for dbmlsync, and choose how to display status
- gf_*appname*_configure_sync – a global function that opens the options window and, if the user clicked OK, calls gf_*appname*_sync to initiate synchronization

Most applications that use the options window provide two menu items or command buttons to launch synchronization: one to open the options window so that users can set up or modify dbmlsync options before requesting a synchronization, and one to request a synchronization with the preset options.

Creating an instance of MLSync

You do not have to use the MobiLink Synchronization Wizard to create a nonvisual object that launches Dbmlsync.exe. You can include an MLSync system object in your applications:

- Programmatically with PowerScript
- By selecting it from the New dialog box

Adding an MLSync object programatically

The code fragment below creates an instance of an MLSync object and programmatically populates all of the necessary properties—as well as some optional properties—using an instance of the system SyncParm structure. Then it calls the Synchronize function to start the database synchronization.

```
SyncParm  Parms
MLSync    mySync
Long      rc

mySync = CREATE MLSync
mySync.MLServerVersion = 9// required property
mySync.Publication = 'salesapi'// required property
mySync.UseLogFile = TRUE// optional
mySync.LogFileName = "C:\temp\sync.log"// optional
mySync.Datasource = 'salesdb_remote'// required
Parms.MLUser = '50'// required
Parms.MLPass = 'xyz123'// required

//The following values are required if they are not
//set by the DSN
Parms.DBUser = 'dba'
Parms.DBPass = 'sql'

//  Apply the property values to the sync object
mySync.SetParm(Parms)
//  Launch the synchronization process
rc = mySync.Synchronize()
destroy mySync
```

Adding an MLSync object from the New dialog box

You can add an MLSync object to a target PBL using the New dialog box: from the PowerBuilder menu, choose File>New, go to the PB Object tab, select Standard Class, then MLSync. This opens a new MLSync object in the User Object painter, where you can initialize all or some of the properties. When you are finished, you can save it as a new object in your target PBL.

Since all of the properties are already initialized, including userids and passwords, it is ready for immediate use. To launch a synchronization requires very little coding, as this example for an MLsync object that you save as "nvo_my_mlsync" illustrates:

```
nvo_my_mlsync mySync
Long   rc
mySync = CREATE nvo_my_mlsync
mySync.Synchronize()
destroy mySync
```

You would typically add the above code to the Clicked event for a menu item or a command button on one of the application windows.

For more information

For more information on system objects related to synchronization, and their functions, events, and properties, see MLSynchronization, MLSync, and SyncParm in the online Help.

Auxiliary objects for MobiLink synchronization

If you create an instance of MLSync by PowerScript code or from the New dialog box, you should also consider using auxiliary objects that are generated automatically by the wizard that you can customize in the PowerBuilder Window painter.

Using an existing synchronization progress window

After you instantiate an MLSync object and call SetParm to enable an end user to set authentication properties at runtime, you can call a Response! type window to document the progress of a database synchronization. You open the progress window with an OpenWithParm call, using the window name and the MLSync object name as arguments. By default, the wizard generates a progress window named w_*appname*_syncprogress and adds the OpenWithParm call for you.

In the Properties view for an MLSync object, you can select a customized progress window to document the progress of a synchronization call. If you customize a wizard-generated progress window—typically to hide some of the fields on its tab pages, or even to hide one or two of the tab pages—you can select the customized progress window for all of your MobiLink applications.

Changing the connection arguments at runtime

To allow a user to override authentication parameters at runtime, you can call a customized options window or the synchronization options window generated by the wizard. The options window can, in turn, call an instance of the SyncParm object that can be initialized with authentication values from a highly secure persistent store, such as a remote database table. You can choose to make some or all of the authentication values writeable, allowing the end user to override them at runtime.

Maintaining property settings in the MLSync object

Normally when you call SetParm(SyncParm) from an MLSync object, you automatically override any authentication values (AuthenticationParms, DBUser, DBPass, EncryptionKey, MLUser, and MLPass) that you set for properties of the MLSync object—even when the value of a particular SyncParm property is an empty string. However, if you call SetNull to set a particular property of the SyncParm object to NULL before you call SetParm, the property value in the MLSync object will be used instead.

The default synchronization options window, w_*appname*_sync_options, returns a SyncParm structure to its caller through the PowerObjectParm property of the Message object. This allows the caller to save the highly sensitive authentication property values in a secure location. It also sets the SyncParm ReturnCode property with an integer value that indicates whether to proceed with the actual synchronization.

Default tab pages of the options window

The default synchronization options window has four tab pages: Subscriptions, SQL Anywhere, MobiLink Server, and Settings.

Subscriptions page When you used the MobiLink wizard, you selected one or more publications from the list of available publications. The selected publications display on the Subscriptions page, but cannot be edited at runtime.

Each remote user can supply a MobiLink synchronization user name on this page. The name must be associated in a subscription with the publications displayed on the page. If the application is always used by the same MobiLink user, this information never needs to be supplied again. The name is saved in the registry and used by default every time synchronization is launched from the application on this device.

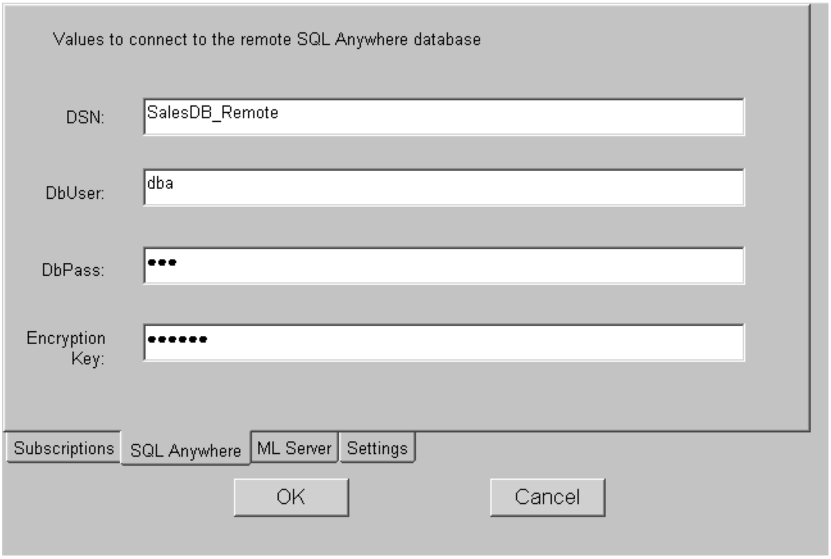
The MobiLink password and authentication parameters are never saved to the user’s registry. They can either be entered each time by the user or provided from a secure database.

SQL Anywhere page Remote users can supply a DSN file name on this page to pass all the arguments needed to connect to a remote database.

If a DSN file is not used, or if the DSN file does not include a user name and password, each remote user can supply a remote database user name. The name is saved in the registry and used by default every time synchronization is launched from the application on this device.

Figure 13-3 displays the options window SQL Anywhere tab page with DSN, DBUser, DBPass, and Encryption Key fields. The database password and encryption key are never saved in the registry.

Figure 13-3: Synchronization options window



Values to connect to the remote SQL Anywhere database

DSN: SalesDB_Remote

DbUser: dba

DbPass: •••

Encryption Key: ••••••

MobiLink Server page When you create a subscription, you specify a protocol, host, port, and other connection options. For ease of testing, the default protocol is TCP/IP and the default host is localhost. The default port is 2439 for TCP/IP, 80 for HTTP, and 443 for HTTPS.

You might need to change these defaults when you are testing, and your users might need to change them when your application is in use if the server is moved to another host or the port changes. If you did not enter values for the host and port at design time, and the user does not make any changes on this page, dbmlsync uses the values in the subscription.

For more information about subscriptions, see “Adding subscriptions” on page 212.

Settings page The Settings page displays logging options, and any other dbmlsync options you specified at design time and lets the user change any of these options at runtime. It also gives the user a choice of displaying or not displaying a synchronization progress window.

Extended options
Extended options are added to the dbmlsync command line with the -e switch. You do not need to type the -e switch in the text box.

Using the synchronization objects in your application

Before you use the generated objects, you should examine them in the PowerBuilder painters to understand how they interact. Many of the function and event scripts contain comments that describe their purpose.

All the source code is provided so that you have total control over how your application manages synchronization. You can use the objects as they are, modify them, or use them as templates for your own objects.

Properties of the user object

The nvo_*appname*_mlsync user object contains properties that represent specific dbmlsync arguments, including the publication name, the MobiLink server host name and port, and the user name and password for a connection to the remote database.

When you run the wizard, the values that you specify for these properties are set as default values in the script for the constructor event of the user object. They are also set in the Windows registry on the development computer in *HKEY_CURRENT_USER\Software\Sybase\PowerBuilder\11.0\appname \MobiLink*, where *appname* is the name of your application.

At runtime, the constructor event script gets the values of the properties from the registry on the remote machine. If they cannot be obtained from the registry, or if you override the registry settings, the default value supplied in the script is used instead and is written to the registry.

You can change the default values in the event script, and you can let the user change the registry values at runtime by providing a menu item that opens the w_*appname*_sync_options window.

Launching dbmlsync

To enable the user to launch a synchronization process, code a button or menu event script to call the gf_*appname*_sync global function. This function creates an instance of the nvo_*appname*_mlsync user object, and the user object's constructor event script sets the *appname\MobiLink* key in the registry of the remote machine.

If you specified in the wizard that the progress window should display, the global function opens the progress window, whose ue_postopen event calls the nvo_*appname*_mlsync user object's synchronize function; otherwise, the global function calls the synchronize function. The synchronize function launches dbmlsync as an external process.

Supplying a MobiLink user name and password

The global function takes a structure for its only argument. You can pass a system SyncParm structure that you instantiate. The structure includes six variables with string datatypes (one each for MobiLink and remote database user names and passwords, as well as variables for the authentication parameters and the encryption key) and another variable that takes a long datatype for a return code.

If you assign valid values to the structure that you pass as an argument, the global function passes these values to the user object to enable MobiLink server and remote database connections.

The options window (described in "Default tab pages of the options window" on page 193) provides a mechanism to store certain of these values in the registry the first time a user starts a synchronization. (Sensitive password and encryption information is never saved to the registry.) Subsequent synchronizations can be started without the user having to reenter the information, however, the options window can still be used to override and reset the registry values.

Retrieving data after synchronization

After synchronizing, you would typically test for synchronization errors, then retrieve data from the newly synchronized database. For example:

```
if gf_myapp_sync(s_opt) <> 0 then
   MessageBox("Error", "MobiLink error")
else
   dw_1.Retrieve()
end if
```

Capturing dbmlsync messages

The PowerBuilder VM traps messages from the dbmlsync process and triggers events in the user object as the synchronization process runs.

These events are triggered before synchronization begins as the upload stream is prepared:

ue_begin_logscan (long *rescan_log*)
ue_progress_info (long *progress_index*, long *progress_max*)
ue_end_logscan ( )

These events correspond to events on the synchronization server, as described in "Connection events" on page 201:

ue_begin_sync (string *user_name*, string *pub_names*)
ue_connect_MobiLink ( )
ue_begin_upload ( )
ue_end_upload ( )
ue_begin_download ( )
ue_end_download (long *upsert_rows*, long *delete_rows*)

```
ue_disconnect_MobiLink( )
ue_end_sync ( long status_code )
```

These events are triggered after ue_end_upload and before ue_begin_download:

```
ue_wait_for_upload_ack ( )
ue_upload_ack ( long upload_status )
```

These events are triggered when various messages are sent by the server:

```
ue_error_msg ( string error_msg )
ue_warning_msg ( string warning_msg )
ue_file_msg ( string file_msg )
ue_display_msg ( string display_msg )
```

The default event scripts created by the wizard trigger corresponding events in the optional progress window, if it exists. The window events write the progress to the multiline edit control in the progress window. Some window events also update a static text control that displays the phase of the synchronization operation that is currently running (log scan, upload, or download) and control a horizontal progress bar showing what percentage of the operation has completed.

You can also add code to the user object or window events that will execute at the point in the synchronization process when the corresponding MobiLink events are triggered. The dbmlsync process sends the event messages to the controlling PowerBuilder application and waits until PowerBuilder event processing is completed before continuing.

Cancelling synchronization

The Cancel button on the progress window calls the cancelsync user object function to cancel the synchronization process. If your application does not use the progress window, you can call this function in an event script elsewhere in your application.

Runtime requirements for synchronization on remote machines

Support files required on remote machine

If you do not install PowerBuilder or SQL Anywhere on remote machines, you must copy the files listed in Table 13-1 to use MobiLink synchronization with a PowerBuilder application. These files must be copied to the system path on the remote machine or the directory where you copy your PowerBuilder applications.

Table 13-1: Required runtime files on system path of remote machine

Required files	Description
PBVM110.DLL, *PBDWE110.DLL*, *PBSHR110.DLL*, *PBODB110.DLL*, *PBODB110.INI*, *LIBJCC.DLL*, *LIBJUTILS.DLL*	PowerBuilder files that you can copy from the *Shared\PowerBuilder* directory of the development machine.
ATL71.DLL, *GDIPLUS.DLL*, *MSVCP71.DLL*, *MSVCR71.DLL*	Microsoft files that ship with PowerBuilder. For restrictions on distributing these files with client applications, see the PowerBuilder *Release Bulletin*.
DBENG10.EXE, *DBMLSYNC.EXE*, *DBSERV10.DLL*, *DBTOOL10.DLL*, *DBODBC10.DLL*, *DBMLSOCK.DLL*, *DBLIB10.DLL*, *DBGEN10.DLL*, *DBCON10.DLL*, *DBCTRS.DLL*	SQL Anywhere and MobiLink files that you can copy from the *Sybase\SQL Anywhere 10\win32* directory of the development machine. You should copy these files to a "win32" subdirectory of the location where you copy the PowerBuilder application and supporting runtime files.

Registry requirements for a remote machine

If you install SQL Anywhere on all remote machines that you use with MobiLink synchronization, the required registry entries are assigned automatically. If you copy SQL Anywhere and MobiLink files to a remote machine, you must create the *HKEY_CURRENT_USER\SOFTWARE\Sybase\SQL Anywhere\10.0* registry key and add a "Location" string value that points to the parent directory of the *win32* subdirectory where you copied SQL Anywhere and MobiLink files. (The code in the uf_runsync function of the nvo_*appname*_sync user object appends "*\win32\dbmlsync.exe*" to the path that you assign to this registry value.)

Objects generated by the MobiLink Synchronization wizard also require registry entries to define the ODBC data source for a remote SQL Anywhere connection. Table 13-2 lists the required registry entries. You can create a REG file that installs these registry entries.

Table 13-2: Required registry entries on remote machine

Registry key	Name of string value and data to assign it
HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\SQL Anywhere\10.0	Driver = full path to *DBODBC10.DLL* Setup = full path to *DBODBC10.DLL*
HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\ODBC Drivers	SQL Anywhere 10.0 = "Installed"

Registry key	Name of string value and data to assign it
HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\ODBC Data Sources	*dataSourceName* = “SQL Anywhere 10.0”
HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\dataSourceName	Driver = full path to *DBODBC10.DLL* Userid = user name for remote database Password = password for remote database DatabaseName = *remoteDatabaseName* DatabaseFile = full path to remote database ServerName = *remoteDatabaseName* Start = “dbeng10 -c 8M” CommLinks = “shmem”

Using a file DSN instead of a registry DSN

You can use a file DSN or a registry DSN for your remote database connections. To avoid having to specify a fully qualified path, you can copy file DSNs to a path specified by the ODBC registry key (typically *c:\program files\common files\ODBC\data sources*).

The following is an example of the contents of a valid file DSN:

```
[ODBC]
DRIVER=SQL Anywhere 10.0
UID=dba
Compress=NO
AutoStop=YES
Start=dbeng10 -c 8M -zl -ti 0
EngineName=SalesDB_Remote
DBN=SalesDB_Remote
DatabaseFile=C:\work\salesdb\salesdb_remote.db
DatabaseName=SalesDB_remote
```

The Datasource property of the MLSync object distinguishes a file DSN from a registry DSN using these rules:

- If the Datasource name ends with a *.dsn* file extension, it is a file DSN
- If the Datasource name begins with “***drive***:\” prefix where *drive* is any alphabetic character, then it is a file DSN

File DSN location before EBFs are applied
If you have not applied the latest EBFs to SQL Anywhere 10.0.0 or Adaptive Server Anywhere 9, dbmlsync looks in the current directory for file DSNs when a full path is not specified—not in the path specified by the ODBC registry key. The registry key is used by SQL Anywhere 10.0.1 and later to locate file DSNs when their paths are not fully qualified.

Preparing consolidated databases

Whether you are designing a new database or preparing an existing one to be used as a MobiLink consolidated database, you must install the MobiLink system tables in that database. SQL Anywhere provides setup scripts for Sybase Adaptive Server Enterprise, Oracle, Microsoft SQL Server, and IBM DB2. A setup script is not required for SQL Anywhere databases.

MobiLink system tables store information for MobiLink users, tables, scripts, and script versions in the consolidated database. You will probably not directly access these tables, but you alter them when you perform actions such as adding synchronization scripts.

ODBC connections and drivers

To carry out synchronization, the MobiLink synchronization server needs an ODBC connection to the consolidated database. You must have an ODBC driver for your server and you must create an ODBC data source for the database on the machine on which your MobiLink synchronization server is running. For a list of supported drivers, see Recommended ODBC Drivers for MobiLink at http://www.sybase.com/detail?id=1011880.

Writing synchronization scripts

There are two types of events that occur during synchronization and for which you need to write synchronization scripts:

- Connection events that perform global tasks required during every synchronization
- Table events that are associated with a specific table and perform tasks related to modifying data in that table

Connection events

At the connection level, the sequence of major events is as follows:

```
begin_connection
   begin_synchronization
      begin_upload
      end_upload
      prepare_for_download
      begin_download
      end_download
   end_synchronization
end_connection
```

When a synchronization request occurs, the begin_connection event is fired. When all synchronization requests for the current script version have been completed, the end_connection event is fired. Typically you place initialization and cleanup code in the scripts for these events, such as variable declaration and database cleanup.

Apart from begin_connection and end_connection, all of these events take the MobiLink user name stored in the ml_user table in the consolidated database as a parameter. You can use parameters in your scripts by placing question marks where the parameter value should be substituted.

To make scripts in SQL Anywhere databases easier to read, you might declare a variable in the begin_connection script, then set it to the value of ml_username in the begin_synchronization script.

For example, in begin_connection:

```
CREATE VARIABLE @sync_user VARCHAR(128);
```

In begin_synchronization:

```
SET @sync_user = ?
```

The begin_synchronization and end_synchronization events are fired before and after changes are applied to the remote and consolidated databases.

The begin_upload event marks the beginning of the upload transaction. Applicable inserts and updates to the consolidated database are performed for all remote tables, then rows are deleted as applicable for all remote tables. After end_upload, upload changes are committed.

If you do not want to delete rows from the consolidated database, do not write scripts for the upload_delete event, or use the STOP SYNCHRONIZATION DELETE statement in your PowerScript code. For more information, see “Deleting rows from the remote database only” on page 214.

The begin_download event marks the beginning of the download transaction. Applicable deletes are performed for all remote tables, and then rows are added as applicable for all remote tables in the download_cursor. After end_download, download changes are committed. These events have the date of the last download as a parameter.

Other connection-level events can also occur, such as handle_error, report_error, and synchronization_statistics. For a complete list of events and examples of their use, see the chapter on synchronization events in the *MobiLink Administration Guide*.

Table events

Many of the connection events that occur between the begin_synchronization and end_synchronization events, such as begin_download and end_upload, also have table equivalents. These and other overall table events might be used for tasks such as creating an intermediate table to hold changes or printing information to a log file.

You can also script table events that apply to each row in the table. For row-level events, the order of the columns in your scripts must match the order in which they appear in the CREATE TABLE statement in the remote database, and the column names in the scripts must refer to the column names in the consolidated database.

Generating default scripts

Although there are several row-level events, most tables need scripts for three upload events (for INSERT, UPDATE, and DELETE) and one download event. To speed up the task of creating these four scripts for every table, you can generate scripts for them automatically by running the “create a synchronization model” task from the MobiLink plug-in in Sybase Central.

For information on the MobiLink plug-in, see the online *MobiLink Getting Started* book.

The MobiLink plug-in allows you to add more functionality to default scripts than default scripts generated in earlier versions of MobiLink. However, if you are using ASA 8 or ASA 9 instead of SQL Anywhere 10, you can still generate default synchronization scripts by starting the MobiLink synchronization server with the -za switch and setting the SendColumnNames extended option for dbmlsync.

The following procedure describes how to generate ASA 8 or 9 synchronizations scripts from the PowerBuilder UI.

❖ To generate ASA 8 or 9 synchronization scripts automatically from PowerBuilder:

1 Expand the ODBC Utilities folder in the Database painter and double-click the MobiLink Synchronization Server item.

  The MobiLink Synchronize Server Options dialog box displays.

2 Select Adaptive Server Anywhere 8 or 9 from the MobiLink Version drop-down list.

  You enable the Automatic Script Generation check box.

3 Select the Automatic Script Generation check box in the MobiLink Synchronize Server Options dialog box and click OK to start the server.

  You can open this dialog box from the Utilities folder in the Database painter or the Database Profiles dialog box.

4 In your application, enter `SendColumnNames=ON` in the Extended text box on the Settings page of the w_*appname*_sync_options window.

  You must have at least one publication, user, and subscription defined in the remote database. If you have more than one publication or user, you must use the -n and/or -u switches to specify which subscription you want to work with.

  If there are existing scripts in the consolidated database, MobiLink does nothing. If there are no existing scripts, MobiLink generates them for all tables specified in the publication. The scripts control the upload and download of data to and from your client and consolidated databases.

  If the column names on the remote and consolidated database differ, the generated scripts must be modified to match the names on the consolidated database.

You can also generate ASA 8 or 9 synchronization scripts from a command prompt. Start the server using the -za switch, then run dbmlsync and set the SendColumnNames extended option to on. For example:

```
dbmlsrv9 -c "dsn=masterdb" -za
dbmlsync -c "dsn=remotedb" -e SendColumnNames=ON
```

Generated scripts

Table 13-3 shows sample default scripts generated by the MobiLink plug-in in Sybase Central. The scripts are generated for a table named emp with the columns emp_id, emp_name, and dept_id. The primary key is emp_id. The generated download scripts use a timestamp based download.

Table 13-3: Sample default synchronization scripts from MobiLink plug-in

Script name	Script

upload_insert	INSERT INTO "GROUP1"."emp" ("emp_id", "emp_name", "dept_id")
	VALUES ({ml r."emp_id"}, {ml r."emp_name"}, {ml r."dept_id"})

upload_update	UPDATE "GROUP1"."emp" SET "emp_name" = {ml r."emp_name"}, "dept_id" = {ml r."dept_id"}
	WHERE "emp_id" = {ml r."emp_id"}

upload_delete	DELETE FROM "GROUP1"."emp"
	WHERE "emp_id" = {ml r."emp_id"}

download_cursor	SELECT "GROUP1"."emp"."emp_id", "GROUP1"."emp"."emp_name", "GROUP1"."emp"."dept_id"
	FROM "GROUP1"."emp"
	WHERE "GROUP1"."emp"."last_modified" >= {ml s.last_table_download}

download_delete_cursor	SELECT "emp_del"."emp_id
	FROM "emp_del"
	WHERE "emp_del"."last_modified" >= {ml s.last_table_download}

The scripts that you generate with the MobiLink plug-in constitute a synchronization model. After you create a synchronization model, you must use the “Deploy the synchronization model” task of the plug-in to deploy the scripts to consolidated and remote databases or to SQL files.

Table 13-4 shows the scripts that are generated for the same table using the -za command switch for the ASA 9 MobiLink synchronization server. The scripts generated for downloading data perform “snapshot” synchronization. A complete image of the table is downloaded to the remote database. Typically you need to edit these scripts to limit the data transferred.

For more information, see “Limiting data downloads” on page 213.

Table 13-4: Sample default scripts generated by dbmlsrv9 -za

Script name	Script

upload_insert	`INSERT INTO emp (emp_id, emp_name, dept_id)`
	`VALUES (?,?,?)`

upload_update	`UPDATE emp SET emp_name = ?, dept_id = ?`
	`WHERE emp_id=?`

upload_delete	`DELETE FROM emp`
	`WHERE emp_id=?`

download_cursor	`SELECT emp_id, emp_name, dept_id FROM emp`

Before modifying any scripts, you should test the synchronization process to make sure that the generated scripts behave as expected. Performing a test after each modification will help you narrow down errors.

Working with scripts and users in Sybase Central

You can view and modify existing scripts and write new ones in the MobiLink Synchronization plug-in in Sybase Central. These procedures describe how to connect to the plug-in and write scripts, and how to add a user to the consolidated database.

❖ **To connect to a consolidated database in Sybase Central:**

1 Start Sybase Central and select Connections>Connect with MobiLink 10 from the menu bar.

2 On the Identification page in the Connect to Consolidated Database dialog box, select or browse to a data source name or file, and click OK.

When you expand the node for a consolidated database in the MobiLink Synchronization plug-in, you see folders with the following labels: Tables, Connection Scripts, Synchronized Tables, Users, Versions, and Notifications. All the procedures in this section begin by opening one of these folders.

Script versions

Scripts are organized into groups called script versions. By specifying a particular version, MobiLink clients can select which set of synchronization scripts is used to process the upload stream and prepare the download stream. If you want to define different versions for scripts, you must add a script version to the consolidated database before you add scripts for it.

If you create two different versions, make sure that you have scripts for all required events in both versions.

❖ **To add a script version:**

1. Open the Versions folder, then select File>New>Version from the Sybase Central menu bar.
2. In the Create Script Version wizard, provide a name for the version and optionally a description, then click Finish.

   Sybase Central creates the new version and gives it a unique integer identifier.

Adding synchronized tables and scripts

Scripts added for connection events are executed for every synchronization. Scripts added for table events are executed when a specific table has been modified. You must specify that a table is synchronized before you can add scripts for it.

❖ **To add a table for synchronization:**

1. Open the Synchronized Tables folder and select File>New>Synchronized Table.
2. Specify a remote table name you want to synchronize or select a table in the consolidated database that has the same name as a table in the remote database.
3. Click Finish.

❖ **To add a script to a synchronized table:**

1. Double-click a table name in the Synchronized Tables folder, then select File>New>Table Script.
2. In the Create Table Script wizard, select the version for which you want to add a script, select the event you want to cause the script to execute, and click Next.
3. Choose to create a new script definition and the language (SQL, Java, or .NET) in which you want to write the definition, or select an existing script version that you want to share for the new script.
4. Click Finish.
5. Type your script in the editor that displays, then save and close the file.

   For example, if you want to remove rows that have been shipped from the Order table in a remote database, you can place the following SELECT statement in the download_delete_cursor event, where order_id is the primary key column. The first parameter to this event is the last_download timestamp. It is used here to supply the value for a last_modified column:

   ```
   SELECT order_id
   ```

```
  FROM Order
WHERE status = 'Shipped'
  AND last_modified >= ?
```

For more information about using the download_delete_cursor event, see the section on “Writing download_delete_cursor scripts” in the online *MobiLink - Server Administration* book.

❖ **To add a connection-level script:**

1. Open the Connection Scripts folder and select File>New>Connection Script from the menu bar.
2. Follow steps 2 to 5 in the previous procedure.

Adding users

You can add users directly to the ml_user table in the consolidated database, then provide the user names and optional passwords to your users. To add a user, open the Users folder, select File>New>User, and complete the Create User wizard.

You also have to add at least one user name to each remote database, as described in “Creating MobiLink users” on page 210.

Creating remote databases

Any SQL Anywhere database can be converted for use as a remote database in a MobiLink installation. You can also create a new SQL Anywhere remote database that uses all or part of the schema of the consolidated SQL Anywhere database.

You create the database on your desktop using the Sybase Central SQL Anywhere plug-in, the Create ASA Database utility in the Database painter, or another tool. If your database uses an English character set, use the 1252 Latin1 collation sequence.

To use a database as a remote database for MobiLink synchronization, you need to create at least one publication and MobiLink user, then add a subscription to the publication for the user. See “Creating and modifying publications” on page 208, “Creating MobiLink users” on page 210, and “Adding subscriptions” on page 212.

Remote database schemas

Tables in a remote database need not be identical to those in the consolidated database, but you can often simplify your design by using a table structure in the remote database that is a subset of the one in the consolidated database. Using this method ensures that every table in the remote database exists in the consolidated database. Corresponding tables have the same structure and foreign key relationships as those in the consolidated database.

Tables in the consolidated database frequently contain extra columns that are not synchronized. Extra columns can even aid synchronization. For example, a timestamp column can identify new or updated rows in the consolidated database. In other cases, extra columns or tables in the consolidated database might hold information that is not required at remote sites.

Creating and modifying publications

You create publications using Sybase Central or the SQL CREATE PUBLICATION statement. In Sybase Central, all publications and articles appear in the Publications folder. This section describes how to create publications in Sybase Central. For information about creating and modifying publications using SQL, see the online *MobiLink - Client Administration* book.

Connecting to the database in Sybase Central

You use the SQL Anywhere plug-in in Sybase Central, not the MobiLink Synchronization plug-in, to work with MobiLink clients and remote databases. For information on starting Sybase Central from the PowerBuilder design time environment, see the *User's Guide*.

You must have DBA authority to create or modify publications, MobiLink users, and subscriptions.

❖ **To connect to the database in Sybase Central:**

1 Start Sybase Central, select Connections>Connect with SQL Anywhere 10 from the Sybase Central menu bar.

2 On the Identification page in the Connect dialog box, enter DBA as the user name and SQL as the password, select or browse to the data source name or file and click OK.

Publishing all the rows and columns in a table

The simplest publication you can create is a single article that consists of all rows and columns of one or more tables. The tables must already exist.

❖ **To publish one or more entire tables in Sybase Central:**

1 Connect to Sybase Central as described in “Connecting to the database in Sybase Central” on page 208.

2 Open the Publications folder and select File>New>Publication from the Sybase Central menu.

3 Type a name for the new publication and click Next.

4 On the Specify Tables page, select a table from the list of available tables and click Add.

The table appears in the list of selected tables on the right.

5 Optionally, add more tables. The order of the tables is not important.

6 Click Finish.

Publishing only some columns in a table

You can create a publication that contains all the rows but only some of the columns of a table.

❖ **To publish only some columns in a table in Sybase Central:**

1 Follow the first four steps of the procedure in "Publishing all the rows and columns in a table" on page 208.

2 Click Next. On the Specify Columns page, double-click the table's icon to expand the list of available columns, select each column you want to publish, and click Add.

The selected columns appear on the right.

3 Click Finish.

Publishing only some rows in a table

You can create a publication that contains some or all of the columns in a table, but only some of the rows. You do so by writing a search condition that matches only the rows you want to publish.

In MobiLink, you can use the WHERE clause to exclude the same set of rows from all subscriptions to a publication. All subscribers to the publication upload any changes to the rows that satisfy the search condition.

❖ **To create a publication using a WHERE clause in Sybase Central:**

1 Follow the first four steps of the procedure in "Publishing all the rows and columns in a table" on page 208, and optionally the first two steps of the procedure in "Publishing only some columns in a table" on page 209.

2 Click Next. On the Specify Where Clauses page, select the table and type the search condition in the lower box.

Optionally, you can use the Insert dialog box to help you format the search condition.

3 Click Finish.

Adding articles

You can add articles to existing publications.

❖ **To add articles in Sybase Central:**

1. Connect to Sybase Central as described in “Connecting to the database in Sybase Central” on page 208.
2. Open the Publications folder and double-click the name of the publication to which you want to add an article.
3. Select File>New>Article from the Sybase Central menu.
4. In the Create Article wizard, select a table and click Next.
5. If you want only some columns to be synchronized, select the Selected Columns radio button and select the columns.
6. If you want to add a WHERE clause, click Next and enter the clause.
7. Click Finish.

Modifying and removing publications and articles

You can modify or drop existing publications in Sybase Central by navigating to the location of the publication and selecting Properties or Delete from its pop-up menu. You can modify and remove articles in the same way.

Publications can be modified only by the DBA or the publication's owner. You must have DBA authority to drop a publication. If you drop a publication, all subscriptions to that publication are automatically deleted as well.

Avoid altering publications in a running MobiLink setup
Altering publications in a running MobiLink setup is likely to cause replication errors and can lead to loss of data unless carried out with care.

Creating MobiLink users

MobiLink users are not the same as database users. Each type of user resides in a different namespace. MobiLink user IDs can match the names of database users, but there is no requirement that they match.

❖ **To add a MobiLink user to a remote database in Sybase Central:**

1. Connect to Sybase Central as described in “Connecting to the database in Sybase Central” on page 208.
2. Open the MobiLink Users folder and select File>New>User from the Sybase Central menu.

3 Enter a name for the MobiLink user.

The name is supplied to the MobiLink synchronization server during synchronization. In production databases, each user name is usually added to the consolidated database, then provided to the individual user.

4 Click Finish.

❖ **To configure MobiLink user properties in Sybase Central:**

1 Connect to Sybase Central as described in “Connecting to the database in Sybase Central” on page 208.

2 Open the MobiLink Users folder, right-click the MobiLink user, and select Properties from the pop-up menu.

3 Change the properties as needed.

❖ **To drop a MobiLink user in Sybase Central:**

1 Connect to Sybase Central as described in “Connecting to the database in Sybase Central” on page 208.

2 Open the MobiLink Users folder, right-click the MobiLink user, and select Delete from the pop-up menu.

Dropping MobiLink users
You must drop all subscriptions for a MobiLink user before you drop the user from a remote database.

Adding MobiLink users to the consolidated database

The consolidated database contains a table called ml_user that is used to authenticate the names of MobiLink users when a synchronization is requested. When you add a user to a remote database, you need to be sure that the user is also added to the ml_user table.

You can add users automatically by selecting the Automatic Addition of Users check box in the MobiLink Synchronization Server Options dialog box and then starting the server. You open this dialog box from the Utilities folder in the Database painter or Database Profiles dialog box. You can also start the server from a command prompt, passing it the -zu+ switch.

Any users defined in the remote database are added to the ml_user table in the consolidated database, as long as the script for the authenticate_user connection event is undefined. Typically the -zu+ switch is not used in a production environment. Names are usually added to the ml_user table in the consolidated database, then added to each of the remote databases. Each user is given a unique name and optional password.

Adding subscriptions

A synchronization subscription links a particular MobiLink user with a publication. You must have at least one publication and one user to create a subscription.

A subscription can also carry other information needed for synchronization. For example, you can specify the address of the MobiLink server and other connection options. Values for a specific subscription override those set for individual MobiLink users.

Overriding options in the wizard
You can override the MobiLink server name and port set for the subscription and user with settings in the ASA MobiLink Synchronization wizard in PowerBuilder.

Synchronization subscriptions are required in MobiLink SQL Anywhere remote databases. Server logic is implemented through synchronization scripts, stored in the MobiLink system tables in the consolidated database.

A single SQL Anywhere database can synchronize with more than one MobiLink synchronization server. To allow synchronization with multiple servers, create different subscriptions for each server.

❖ **To add a subscription for a MobiLink user in Sybase Central:**

1. Connect to Sybase Central as described in “Connecting to the database in Sybase Central” on page 208.
2. Open the Publications folder, select the publication for which you want to enter a subscription, select the Synchronization Subscriptions tab in the right pane of Sybase Central, then select File>New>Synchronization Subscription from the menu bar.

   Instead of creating a new subscription in the Publications folder, you can create one in the MobiLink Users folder by double-clicking the user for whom you want to create a subscription, and then selecting File>New>Synchronization Subscription from the menu bar.
3. In the Create Synchronization Subscription wizard, select the user for whom you want to enter a subscription and click Finish.

   If you started the wizard from the MobiLink Users folder, the wizard prompts you to select the publication to which you want to subscribe. In this case, select the publication and click Finish.

❖ **To modify a subscription in Sybase Central:**

1. Connect to Sybase Central as described in "Connecting to the database in Sybase Central" on page 208.
2. Open the MobiLink Users folder and double-click the name of the MobiLink user who owns the subscription you want to modify.
3. On the Synchronization Subscriptions tab, right-click the subscription you want to modify and select Properties from the pop-up menu.
4. Change the properties as needed on the Connection and Extended Options pages of the Synchronization Subscription Properties dialog box.

❖ **To delete a synchronization subscription in Sybase Central:**

1. Connect to Sybase Central as described in "Connecting to the database in Sybase Central" on page 208.
2. Open the MobiLink Users folder and double-click the name of the MobiLink user who owns the subscription you want to delete.
3. On the Synchronization Subscriptions tab, right-click the subscription you want to delete and click Delete.
4. Click Yes in the Confirm Delete dialog box.

Synchronization techniques

This section highlights some issues that you need to consider when designing an application that uses MobiLink synchronization.

Limiting data downloads

One of the major goals of synchronization is to increase the speed and efficiency of data movement by restricting the amount of data moved. To limit the data transferred by the download_cursor script, you can partition data based on its timestamp, the MobiLink user name, or both.

Timestamp partitioning One way to limit downloads to data changed since the last download is to add a last_modified column to each table in the consolidated database (or, if the table itself cannot be changed, to a shadow table that holds the primary key and that is joined to the original table in the download_cursor script). The last_modified column need only be added to the consolidated database.

In SQL Anywhere, you can use built-in DEFAULT TIMESTAMP datatypes for this column. In other DBMSs, you need to provide an update trigger to set the timestamp of the last_modified column.

The timestamp is generated on the consolidated database and downloaded unmodified to the remote database during synchronization; the time zone of the remote database does not affect it.

User-based partitioning The download_cursor script has two parameters: last_download, of datatype datetime, and ml_username, of type varchar(128). You can use these parameters to restrict the download not only to rows that have changed since the last synchronization, but also to rows that belong to the current user.

In this sample download_cursor script, only those rows are downloaded that have been modified since the last synchronization, and that apply to the sales representative whose ID matches the MobiLink user ID:

```
SELECT order_id, cust_id, order_date
   FROM Sales_Order
WHERE last_modified >= ?
   AND sales_rep = ?
```

For this to work correctly, the MobiLink user ID must match the sales_rep ID. If this is not the case, you might need to join a table that associates these two IDs.

Primary key uniqueness

In a conventional client/server environment where clients are always connected, referential integrity is directly imposed. In a mobile environment, you must ensure that primary keys are unique and that they are never updated. There are several techniques for achieving this, such as using primary key pools.

Handling conflicts

You need to handle conflicts that arise when, for example, two remote users update the same rows but synchronize at different intervals, so that the latest synchronization might not be the latest update. MobiLink provides mechanisms to detect and resolve conflicts.

Deleting rows from the remote database only

By default, when a user starts a synchronization, the net result of all the changes made to the database since the last synchronization is uploaded to the consolidated database. However, sometimes a remote user deletes certain rows from the remote database to recapture space, perhaps because the data is old or a customer has transferred to another sales agent. Usually, those deleted rows should not be deleted from the consolidated database.

One way to handle this is to use the command STOP SYNCHRONIZATION DELETE in a script in your PowerBuilder application to hide the SQL DELETE statements that follow it from the transaction log. None of the subsequent DELETE operations on the connection will be synchronized until the START SYNCHRONIZATION DELETE statement is executed.

For example, you might provide a menu item called Delete Local where the code that handles the delete is wrapped, as in this example:

```
STOP SYNCHRONIZATION DELETE;
// call code to perform delete operation
START SYNCHRONIZATION DELETE;
COMMIT;
```

There are other approaches to handling deletes. For more information, see the chapter on synchronization techniques in the online *MobiLink - Server Administration* book.

CHAPTER 14 Using PowerBuilder XML Services

About this chapter

This chapter presents an overview of XML services in PowerBuilder. It describes the PowerBuilder Document Object Model (PBDOM), and describes how to use it in a PowerBuilder application.

Contents

About XML and PowerBuilder

PowerBuilder provides several features that enable you to work with the Extensible Markup Language (XML). You can:

- Export the data in a DataWindow object to XML, and import data in an XML document or string into a DataWindow object
- Determine whether an XML document or string is well-formed or conforms to a schema or DTD using the XMLParseFile and XMLParseString PowerScript functions
- Build applications and components that can produce and process XML documents

For an overview of XML and information about the export and import capabilities in the DataWindow, see the chapter on exporting and importing XML in the PowerBuilder *User's Guide*.

For information about the XML parsing functions, see their descriptions in the online Help.

This chapter describes how you can produce and process XML documents using the PowerBuilder Document Object Model.

About PBDOM

PBDOM is the PowerBuilder implementation of the Document Object Model (DOM), a programming interface defining the means by which XML documents can be accessed and manipulated.

Although PBDOM is not an implementation of the World Wide Web Consortium (W3C) DOM API, it is very similar. The PBDOM PowerBuilder API can be used for reading, writing, and manipulating standard-format XML from within PowerScript code. PBDOM portrays an XML document as a collection of interconnected objects and provides intuitive methods indicating the use and functionality of each object.

PBDOM is also similar to JDOM, which is a Java-based document object model for XML files.

For information on the W3C DOM and JDOM objects and hierarchies, refer to their respective specifications. The W3C DOM specification is available at http://www.w3.org/DOM/. The JDOM specification, or a link to it, is available at http://www.jdom.org/docs/.

With PBDOM, your applications can parse existing XML documents and extract the information contained as part of a business process or in response to an external request. Applications can also produce XML documents that conform to the type or schema required by other applications, processes, or systems. Existing XML documents can be read and modified by manipulating or transforming the PBDOM tree of objects instead of having to edit XML strings directly.

You can also build components that can produce or process XML documents for use in multitier applications or as part of a Web service.

Node trees

PBDOM interacts with XML documents according to a tree-view model consisting of parent and child nodes. A document element represents the top-level node of an XML document. Each child node of the document element has one or many child nodes that represent the branches of the tree. Nodes in the tree are accessible through PBDOM class methods.

XML parser

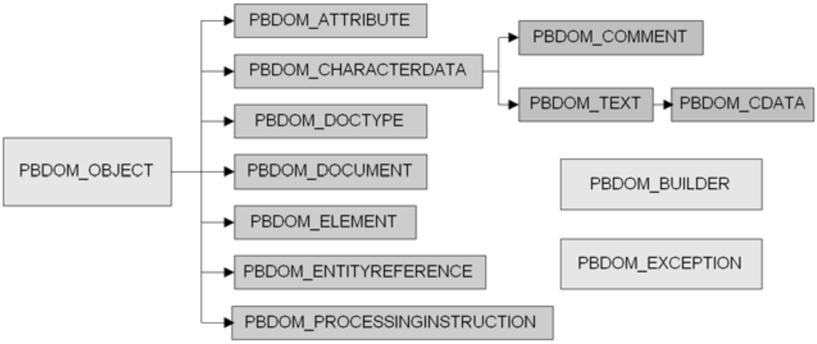
The PBDOM XML parser is used to load and parse an XML document, and also to generate XML documents based on user-specified DOM nodes.

PBDOM provides all the methods you need to traverse the node tree, access the nodes and attribute values (if any), insert and delete nodes, and convert the node tree to an XML document so that it can be used by other systems.

PBDOM object hierarchy

The following figure shows the PBDOM object hierarchy:

Figure 14-1: The PBDOM object hierarchy



PBDOM_OBJECT and its descendants

The base class for PBDOM objects that represent XML nodes, PBDOM_OBJECT, inherits from the PowerBuilder NonVisualObject class. Each of the node types is represented by a PBDOM class whose methods you use to access objects in a PBDOM node tree. PBDOM_OBJECT and its descendants are described in "PBDOM node objects" next. You can also find some information about XML node types in the chapter on exporting and importing XML data in the *PowerBuilder User's Guide*.

PBDOM_BUILDER

The PBDOM_BUILDER class also inherits from NonVisualObject. It serves as a factory class that creates a PBDOM_DOCUMENT from various XML input sources including a string, a file, and a DataStore.

Building a PBDOM_DOCUMENT from scratch
To build a PBDOM_DOCUMENT without a source that contains existing XML, use the PBDOM_DOCUMENT NewDocument methods.

PBDOM_EXCEPTION

The PBDOM_EXCEPTION class inherits from the PowerBuilder Exception class. It extends the Exception class with a method that returns a predefined exception code when an exception is raised in a PBDOM application. For more information about this class, see “Handling PBDOM exceptions” on page 243.

PBDOM node objects

This section describes the PBDOM_OBJECT class and all of the classes that descend from it:

- PBDOM_OBJECT
- PBDOM_DOCUMENT
- PBDOM_DOCTYPE
- PBDOM_ELEMENT
- PBDOM_ATTRIBUTE
- PBDOM_ENTITYREFERENCE
- PBDOM_CHARACTERDATA
- PBDOM_TEXT
- PBDOM_CDATA
- PBDOM_COMMENT
- PBDOM_PROCESSINGINSTRUCTION

For detailed descriptions of PBDOM class methods, see the *PowerBuilder Extension Reference.*

PBDOM_OBJECT

The PBDOM_OBJECT class represents any node in an XML node tree and serves as the base class for specialized PBDOM classes that represent specific node types. The DOM class that corresponds to PBDOM_OBJECT is the Node object. PBDOM_OBJECT contains all the basic features required by derived classes. A node can be an element node, a document node, or any of the node types listed above that derive from PBDOM_OBJECT.

Methods

The PBDOM_OBJECT base class has the following methods:

- AddContent, GetContent, InsertContent, RemoveContent, and SetContent to allow you to manipulate the children of the PBDOM_OBJECT
- Clone to allow you to make shallow or deep clones of the PBDOM_OBJECT
- Detach to detach the PBDOM_OBJECT from its parent
- Equals to test for equality with another PBDOM_OBJECT
- GetName and SetName to get and set the name of the PBDOM_OBJECT
- GetObjectClass and GetObjectClassString to identify the class of the PBDOM_OBJECT
- GetOwnerDocumentObject to identify the owner PBDOM_DOCUMENT of the current PBDOM_OBJECT
- GetParentObject and SetParentObject to get and set the parent of the PBDOM_OBJECT
- GetText, GetTextNormalize, and GetTextTrim to obtain the text data of the PBDOM_OBJECT
- HasChildren to determine whether the PBDOM_OBJECT has any children
- IsAncestorObjectOf to determine whether the PBDOM_OBJECT is the ancestor of another PBDOM_OBJECT

PBDOM_OBJECT inheritance

The PBDOM_OBJECT class is similar to a virtual class in C++ in that it is not expected to be directly instantiated and used. For example, although a PBDOM_OBJECT can be created using the PowerScript CREATE statement, its methods cannot be used directly:

```
PBDOM_OBJECT pbdom_obj
pbdom_obj = CREATE PBDOM_OBJECT
pbdom_obj.SetName("VIRTUAL_PBDOM_OBJ") //exception!
```

The third line of code above throws an exception because it attempts to directly access the SetName method for the base class PBDOM_OBJECT. A similar implementation is valid, however, when the SetName method is accessed from a derived class, such as PBDOM_ELEMENT:

```
PBDOM_OBJECT pbdom_obj
pbdom_obj = CREATE PBDOM_ELEMENT
pbdom_obj.SetName ("VIRTUAL_PBDOM_OBJ")
```

Using the base PBDOM_OBJECT as a placeholder

The PBDOM_OBJECT class can be used as a placeholder for an object of a derived class:

```
PBDOM_DOCUMENT pbdom_doc
PBDOM_OBJECT pbdom_obj

pbdom_doc = CREATE PBDOM_DOCUMENT
pbdom_doc.NewDocument ("", "", &
   "Root_Element_From_Doc_1", "", "")
pbdom_obj = pbdom_doc.GetRootElement
pbdom_obj.SetName &
   ("Root_Element_From_Doc_1_Now_Changed")
```

The instantiated PBDOM_OBJECT pbdom_obj is assigned to a PBDOM_DOCUMENT object, which holds the return value of the GetRootElement method. Here, pbdom_obj holds a reference to a PBDOM_ELEMENT and can be operated on legally like any object of a class derived from PBDOM_OBJECT.

Standalone objects

A PBDOM_OBJECT can be created as a self-contained object independent of any document or parent PBDOM_OBJECT. Such a PBDOM_OBJECT is known as a standalone object. For example:

```
PBDOM_ELEMENT pbdom_elem_1
pbdom_elem_1 = Create PBDOM_ELEMENT
pbdom_elem_1.SetName("pbdom_elem_1")
```

pbdom_elem_1 is instantiated in the derived class PBDOM_ELEMENT using the Create keyword. The SetName method can then be invoked from the pbdom_elem_1 object, which is a standalone object not contained within any document.

Standalone objects can perform any legal PBDOM operations, but standalone status does not give the object any special advantages or disadvantages.

Parent-owned and document-owned objects

A PBDOM_OBJECT can be assigned a parent by appending it to another standalone PBDOM_OBJECT, as in the following example:

```
PBDOM_ELEMENT pbdom_elem_1
PBDOM_ELEMENT pbdom_elem_2

pbdom_elem_1 = Create PBDOM_ELEMENT
pbdom_elem_2 = Create PBDOM_ELEMENT

pbdom_elem_1.SetName("pbdom_elem_1")
pbdom_elem_2.SetName("pbdom_elem_2")
pbdom_elem_1.AddContent(pbdom_elem_2)
```

Two PBDOM_ELEMENT objects, pbdom_elem_1 and pbdom_elem_2, are instantiated. The pbdom_elem_2 object is appended as a child object of pbdom_elem_1 using the AddContent method.

In this example, neither pbdom_elem_1 nor pbdom_elem_2 is owned by any document, and the pbdom_elem_1 object is still standalone. If pbdom_elem_1 were assigned to a parent PBDOM_OBJECT owned by a document, pbdom_elem_1 would cease to be a standalone object.

PBDOM_DOCUMENT

The PBDOM_DOCUMENT class derives from PBDOM_OBJECT and represents an XML DOM document. The PBDOM_DOCUMENT methods allow access to the root element, processing instructions, and other document-level information.

Methods

In addition to the methods inherited from PBDOM_OBJECT, the PBDOM_DOCUMENT class has the following methods:

- DetachRootElement, GetRootElement, HasRootElement, and SetRootElement to manipulate the root element of the PBDOM_DOCUMENT
- GetDocType and SetDocType to get and set the DOCTYPE declaration of the XML document
- NewDocument to build a new PBDOM_DOCUMENT from scratch
- SaveDocument to save the content of the DOM tree in the PBDOM_DOCUMENT to a file

PBDOM_DOCTYPE

The PBDOM_DOCTYPE class represents the document type declaration object of an XML DOM document. The PBDOM_DOCTYPE methods allow access to the root element name, the internal subset, and the system and public IDs.

Methods

In addition to the methods inherited from PBDOM_OBJECT, the PBDOM_DOCTYPE class has the following methods:

- GetPublicID, SetPublicID, GetSystemID, and SetSystemID to get and set the public and system IDs of an externally-referenced ID declared in the PBDOM_DOCTYPE

- GetInternalSubset and SetInternalSubset to get and set the internal subset data of the PBDOM_DOCTYPE

PBDOM_ELEMENT

The PBDOM_ELEMENT represents an XML element modeled in PowerScript. The PBDOM_ELEMENT methods allow access to element attributes, children, and text.

Methods

In addition to the methods inherited from PBDOM_OBJECT, the PBDOM_ELEMENT class has the following methods:

- AddNamespaceDeclaration and RemoveNamespaceDeclaration to add namespace declarations to and remove them from the PBDOM_ELEMENT
- GetAttribute, GetAttributes, GetAttributeValue, HasAttributes, RemoveAttribute, SetAttribute, and SetAttributes to manipulate the attributes of the PBDOM_ELEMENT
- GetChildElement, GetChildElements, HasChildElements, RemoveChildElement, and RemoveChildElements to manipulate the children of the PBDOM_ELEMENT
- GetNamespacePrefix and GetNamespaceURI to get the prefix and URI of the namespace associated with the PBDOM_ELEMENT
- GetQualifiedName to get the full name of the PBDOM_ELEMENT including the prefix (if any)
- SetDocument to set a PBDOM_DOCUMENT as the parent of the PBDOM_ELEMENT
- SetNamespace to set the namespace of the PBDOM_ELEMENT
- SetText to set the text content of the PBDOM_ELEMENT

The relationship between PBDOM_ELEMENT and PBDOM_ATTRIBUTE

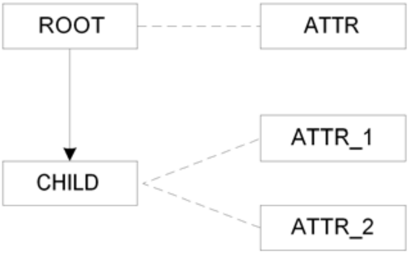
In PBDOM, an XML element's attributes are not its children. They are properties of elements rather than having a separate identity from the elements they are associated with.

Consider the following simple XML document :

```
<root attr="value1">
   <child attr_1="value1" attr_2="value2"/>
</root>
```

The equivalent PBDOM tree is shown in Figure 14-2:

Figure 14-2: Relationship between PBDOM_ELEMENTs and PBDOM_ATTRIBUTEs



The solid line joining root with child represents a parent-child relationship. The dashed lines represent a "property-of" relationship between an attribute and its owner element.

The PBDOM_ELEMENT content management methods do not apply to PBDOM_ATTRIBUTE objects. There are separate get, set, and remove methods for attributes.

Because they are not children of their owner elements, PBDOM does not consider attributes as part of the overall PBDOM document tree, but they are linked to it through their owner elements.

An attribute can contain child objects (XML text and entity reference nodes), so an attribute forms a subtree of its own.

Because an element's attributes are not considered its children, they have no sibling relationship among themselves as child objects do. In the sample XML document and in Figure 14-2, attr_1 and attr_2 are not siblings. The order of appearance of attributes inside its owner element has no significance.

Attribute setting and creation

In PBDOM, an XML element's attribute is set using the PBDOM_ELEMENT SetAttribute and SetAttributes methods. These methods always attempt to create new attributes for the PBDOM_ELEMENT and attempt to replace existing attributes with the same name and namespace URI.

If the PBDOM_ELEMENT already contains an existing attribute with the same name and namespace URI, these methods first remove the existing attribute and then insert a new attribute into the PBDOM_ELEMENT. Calling the SetAttribute method can cause a PBDOM_ATTRIBUTE (representing an existing attribute of the PBDOM_ELEMENT) to become detached from its owner PBDOM_ELEMENT.

For example, consider the following element:

```
<an_element an_attr="some_value"/>
```

If a PBDOM_ELEMENT object pbdom_an_elem represents the element an_element and the following statement is issued, the method first attempts to create a new attribute for the an_element element:

```
pbdom_an_elem.SetAttribute("an_attr",
"some_other_value")
```

Then, because an_element already contains an attribute with the name an_attr, the attribute is removed. If there is an existing PBDOM_ATTRIBUTE object that represents the original an_attr attribute, this PBDOM_ATTRIBUTE is detached from its owner element (an_element).

For more information about attributes and namespaces, see “XML namespaces” on page 244.

PBDOM_ATTRIBUTE

The PBDOM_ATTRIBUTE class represents an XML attribute modeled in PowerScript. The PBDOM_ATTRIBUTE methods allow access to element attributes and namespace information.

Methods

In addition to the methods inherited from PBDOM_OBJECT, the PBDOM_ATTRIBUTE class has the following methods:

- GetBooleanValue, SetBooleanValue, GetDateValue, SetDateValue, GetDateTimeValue, SetDateTimeValue, GetDoubleValue, SetDoubleValue, GetIntValue, SetIntValue, GetLongValue, SetLongValue, GetRealValue, SetRealValue, GetTimeValue, SetTimeValue, GetUIntValue, SetUintValue, GetULongValue,and SetULongValue to get and set the value of the PBDOM_ATTRIBUTE as the specified datatype
- GetNamespacePrefix and GetNamespaceURI to get the prefix and URI of the namespace associated with the PBDOM_ATTRIBUTE
- GetOwnerElementObject and SetOwnerElementObject to get and set the owner PBDOM_ELEMENT of the PBDOM_ATTRIBUTE
- GetQualifiedName to get the full name of the PBDOM_ATTRIBUTE including the prefix, if any
- SetNamespace to set the namespace of the PBDOM_ATTRIBUTE
- SetText to set the text content of the PBDOM_ATTRIBUTE

Child PBDOM_OBJECTs

A PBDOM_ATTRIBUTE contains a subtree of child PBDOM_OBJECTs. The child objects can be a combination of PBDOM_TEXT and PBDOM_ENTITYREFERENCE objects.

The following example produces a PBDOM_ELEMENT named elem that contains a PBDOM_ATTRIBUTE named attr:

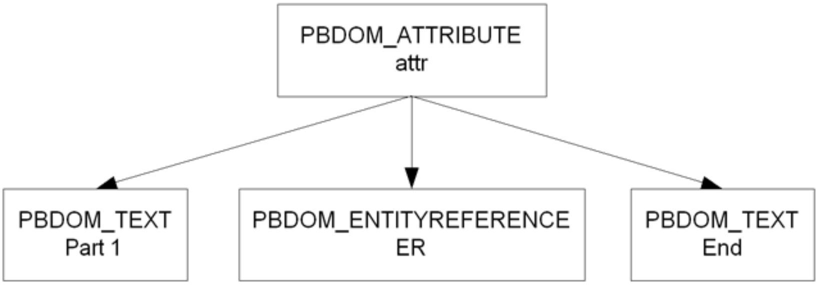
```
PBDOM_ATTRIBUTE pbdom_attr
PBDOM_TEXT pbdom_txt
PBDOM_ENTITYREFERENCE pbdom_er
PBDOM_ELEMENT pbdom_elem

pbdom_elem  = Create PBDOM_ELEMENT
pbdom_elem.SetName ("elem")

pbdom_attr = Create PBDOM_ATTRIBUTE
pbdom_attr.SetName("attr")
pbdom_attr.SetText("Part 1 ")

pbdom_txt = Create PBDOM_TEXT
pbdom_txt.SetText (" End.")

pbdom_er = Create PBDOM_ENTITYREFERENCE
pbdom_er.SetName("ER")

pbdom_attr.AddContent(pbdom_er)
pbdom_attr.AddContent(pbdom_txt)

pbdom_elem.SetAttribute(pbdom_attr)
```

The element tag in the XML looks like this:

```
<elem attr="Part 1 &ER; End.">
```

In Figure 14-3, the arrows indicate a parent-child relationship between the PBDOM_ATTRIBUTE and the other PBDOM_OBJECTs:

Figure 14-3: PBDOM_ATTRIBUTE subtree example



The Default PBDOM_TEXT child

A PBDOM_ATTRIBUTE generally always contains at least one PBDOM_TEXT child that might contain an empty string. This is the case unless the RemoveContent method has been called to remove all contents of the PBDOM_ATTRIBUTE.

The following examples show how a PBDOM_TEXT object with an empty string can become the child of a PBDOM_ATTRIBUTE.

Example 1 The following example uses the PBDOM_ELEMENT SetAttribute method. The name of the PBDOM_ATTRIBUTE is set to attr but the text value is an empty string. The PBDOM_ATTRIBUTE will have one child PBDOM_TEXT that will contain an empty string:

```
PBDOM_DOCUMENT  pbdom_doc
PBDOM_ATTRIBUTE  pbdom_attr
PBDOM_OBJECT    pbdom_obj_array[]

try

  pbdom_doc = Create PBDOM_DOCUMENT
  pbdom_doc.NewDocument("root")

  // Note that the name of the attribute is set to
  // "attr" and its text value is the empty string ""
  pbdom_doc.GetRootElement().SetAttribute("attr", "")

  pbdom_attr = &
     pbdom_doc.GetRootElement().GetAttribute("attr")

  MessageBox ("HasChildren", &
     string(pbdom_attr.HasChildren()))
```

```
catch(PBDOM_EXCEPTION pbdom_except)
  MessageBox ("PBDOM_EXCEPTION", &
     pbdom_except.GetMessage())
end try
```

When you use the SaveDocument method to render pbdom_doc as XML, it looks like this:

```
<root attr="" />
```

Example 2 The following example creates a PBDOM_ATTRIBUTE and sets its name to attr. No text value is set, but a PBDOM_TEXT object is automatically created and attached to the PBDOM_ATTRIBUTE. This is the default behavior for every PBDOM_ATTRIBUTE created in this way:

```
PBDOM_DOCUMENT  pbdom_doc
PBDOM_ATTRIBUTE  pbdom_attr

try
  pbdom_doc = Create PBDOM_DOCUMENT
  pbdom_doc.NewDocument("root")

  // Create a PBDOM_ATTRIBUTE and set its name to "attr"
  pbdom_attr = Create PBDOM_ATTRIBUTE
  pbdom_attr.SetName("attr")

  pbdom_doc.GetRootElement().SetAttribute(pbdom_attr)

  MessageBox ("HasChildren", &
     string(pbdom_attr.HasChildren()))

catch(PBDOM_EXCEPTION pbdom_except)
  MessageBox ("PBDOM_EXCEPTION", &
     pbdom_except.GetMessage())
end try
```

When you call the SetText method (or any of the other Set* methods except SetNamespace), the default PBDOM_TEXT is replaced by a new PBDOM_TEXT. If you call the SetContent method, you can replace the default PBDOM_TEXT by a combination of PBDOM_TEXT and PBDOM_ENTITYREFERENCE objects.

PBDOM_ENTITYREFERENCE

The PBDOM_ENTITYREFERENCE class defines behavior for an XML entity reference node. It is a simple class intended primarily to help you insert entity references within element nodes as well as attribute nodes.

When the PBDOM_BUILDER class parses an XML document and builds up the DOM tree, it completely expands entities as they are encountered in the DTD. Therefore, immediately after a PBDOM_DOCUMENT object is built using any of the PBDOM_BUILDER build methods, there are no entity reference nodes in the resulting document tree.

A PBDOM_ENTITYREFERENCE object can be created at any time and inserted into any document whether or not there is any corresponding DOM entity node representing the referenced entity in the document.

Methods

The PBDOM_ENTITYREFERENCE class has only methods that are inherited from PBDOM_OBJECT.

PBDOM_CHARACTERDATA

The PBDOM_CHARACTERDATA class derives from PBDOM_OBJECT and represents character-based content (not markup) within an XML document. The PBDOM_CHARACTERDATA class extends PBDOM_OBJECT with methods specifically designed for manipulating character data.

Methods

In addition to the methods inherited from PBDOM_OBJECT, the PBDOM_CHARACTERDATA class has the following methods:

- Append to append a text string or the text data of a PBDOM_CHARACTERDATA object to the text in the current object
- SetText to set the text content of the PBDOM_CHARACTERDATA object

Parent of three classes

The PBDOM_CHARACTERDATA class is the parent class of three other PBDOM classes:

- PBDOM_TEXT
- PBDOM_CDATA
- PBDOM_COMMENT

The PBDOM_CHARACTERDATA class, like its parent class PBDOM_OBJECT, is a "virtual" class (similar to a virtual C++ class) in that it is not expected to be directly instantiated and used. For example, creating a PBDOM_CHARACTERDATA with the CREATE statement is legal in PowerScript, but operating on it directly by calling its SetText method is not. The last line in this code raises an exception:

```
PBDOM_CHARACTERDATA pbdom_chrdata
pbdom_chrdata = CREATE PBDOM_CHARACTERDATA

pbdom_chrdata.SetText("character string") //exception!
```

In this example, pbdom_chrdata is declared as a PBDOM_CHARACTERDATA but is instantiated as a PBDOM_TEXT. Calling SetText on pbdom_chrdata is equivalent to calling the PBDOM_TEXT SetText method:

```
PBDOM_CHARACTERDATA pbdom_chrdata
pbdom_chrdata = CREATE PBDOM_TEXT

pbdom_chrdata.SetText("character string")
```

PBDOM_TEXT

The PBDOM_TEXT class derives from PBDOM_CHARACTERDATA and represents a DOM text node in an XML document.

Methods

The PBDOM_TEXT class has no methods that are not inherited from PBDOM_OBJECT or PBDOM_CHARACTERDATA.

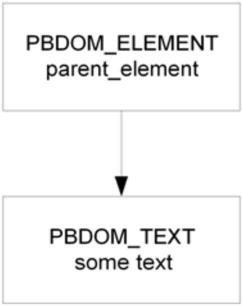
Using PBDOM_TEXT objects

PBDOM_TEXT objects are commonly used to represent the textual content of a PBDOM_ELEMENT or a PBDOM_ATTRIBUTE. Although PBDOM_TEXT objects are not delimited by angle brackets, they are objects and *do not* form the value of a parent PBDOM_ELEMENT.

A PBDOM_TEXT object represented in graphical form in a PBDOM tree is a leaf node and contains no child objects. For example, Figure 14-4 represents the following PBDOM_ELEMENT:

```
<parent_element>some text</parent_element>
```

Figure 14-4: PBDOM_TEXT parent-child relationship



The arrow indicates a parent-child relationship.

Occurrence of PBDOM_TEXTs

When an XML document is first parsed, if there is no markup inside an element's content, the text within the element is represented as a single PBDOM_TEXT object. This PBDOM_TEXT object is the only child of the element. If there is markup, it is parsed into a list of PBDOM_ELEMENT objects and PBDOM_TEXT objects that form the list of children of the element.

For example, parsing the following XML produces one PBDOM_ELEMENT that represents `<element_1>` and one PBDOM_TEXT that represents the textual content `Some Text`:

```
<root>
  <element_1>Some Text</element_1>
</root>
```

The `<element_1>` PBDOM_ELEMENT has the PBDOM_TEXT object as its only child.

Consider this document:

```
<root>
  <element_1>
  Some Text
    <element_1_1>Sub Element Text</element_1_1>
    More Text
    <element_1_2/>
  Yet More Text
  </element_1>
</root>
```

Parsing this XML produces a PBDOM_ELEMENT that represents `<element_1>` and its five children:

- A PBDOM_TEXT representing `Some Text`
- A PBDOM_ELEMENT representing `<element_1_1/>`
- A PBDOM_TEXT representing `More Text`
- A PBDOM_ELEMENT representing `<element_1_2/>`
- A PBDOM_TEXT representing `Yet More Text`

Adjacent PBDOM_TEXT objects

You can create adjacent PBDOM_TEXT objects that represent the contents of a given element without any intervening markup. For example, suppose you start with this document:

```
<root>
  <element_1>Some Text</element_1>
</root>
```

Calling `AddContent("More Text")` on the element_1 PBDOM_ELEMENT produces the following result:

```
<root>
  <element_1>Some TextMore Text</element_1>
</root>
```

There are now two PBDOM_TEXT objects representing "Some Text" and "More Text" that are adjacent to each other. There is nothing between them, and there is no way to represent the separation between them.

Persistence of PBDOM_TEXT objects

The separation of adjacent PBDOM_TEXT objects does not usually persist between DOM editing sessions. When the document produced by adding "More Text" shown in the preceding example is reopened and reparsed, only one PBDOM_TEXT object represents "Some TextMore Text".

PBDOM_CDATA

The PBDOM_CDATA class derives from PBDOM_TEXT and represents an XML DOM CDATA section.

Methods

The PBDOM_CDATA class has no methods that are not inherited from PBDOM_OBJECT or PBDOM_CHARACTERDATA.

Using CDATA objects

You can think of a PBDOM_CDATA object as an extended PBDOM_TEXT object. A PBDOM_CDATA object is used to hold text that can contain characters that are prohibited in XML, such as < and &. Their primary purpose is to allow you to include these special characters inside a large block of text without using entity references.

This example contains a PBDOM_CDATA object:

```
<some_text>
<![CDATA[ (x < y) & (y < z) => x < z ]]>
</some_text>
```

To express the same textual content as a PBDOM_TEXT object, you would need to write this:

```
<some_text>
(x &lt; y) &amp; (y &lt; z) =&gt; x &lt; z
</some_text>
```

Although the PBDOM_CDATA class is derived from PBDOM_TEXT, a PBDOM_CDATA object cannot always be inserted where a PBDOM_TEXT can be inserted. For example, a PBDOM_TEXT object can be added as a child of a PBDOM_ATTRIBUTE, but a PBDOM_CDATA object cannot.

PBDOM_COMMENT

The PBDOM_COMMENT class represents a DOM comment node within an XML document. The PBDOM_COMMENT class is derived from the PBDOM_CHARACTERDATA class.

Methods

The PBDOM_COMMENT class has no methods that are not inherited from PBDOM_OBJECT or PBDOM_CHARACTERDATA.

Using comments

Comments are useful for annotating parts of an XML document with user-readable information.

When a document is parsed, any comments found within the document persist in memory as part of the DOM tree. A PBDOM_COMMENT created at runtime also becomes part of the DOM tree.

An XML comment does not usually form part of the content model of a document. The presence or absence of comments has no effect on a document's validity, and there is no requirement that comments be declared in a DTD.

PBDOM_PROCESSINGINSTRUCTION

The PBDOM_PROCESSINGINSTRUCTION class represents an XML processing instruction (PI). The PBDOM_PROCESSINGINSTRUCTION methods allow access to the processing instruction target and its data. The data can be accessed as a string or, where appropriate, as name/value pairs.

The actual processing instruction of a PI is a string. This is so even if the instruction is cut up into separate `name="value"` pairs. PBDOM, however, does support such a PI format. If the PI data does contain these pairs, as is commonly the case, then PBDOM_PROCESSINGINSTRUCTION parses them into an internal list of name/value pairs.

Methods

In addition to the methods inherited from PBDOM_OBJECT, the PBDOM_PROCESSINGINSTRUCTION class has the following methods:

- GetData and SetData to get and set the raw data of the PBDOM_PROCESSINGINSTRUCTION object
- GetNames to get a list of names taken from the part of the PBDOM_PROCESSINGINSTRUCTION data that is separated into name="value" pairs
- GetValue, RemoveValue, and SetValue to get, remove, and set the value of a specified name/value pair in the PBDOM_PROCESSINGINSTRUCTION object
- GetTarget to get the target of a PBDOM_PROCESSINGINSTRUCTION. For example, the target of the XML declaration, which is a special processing instruction, is the string `xml`.

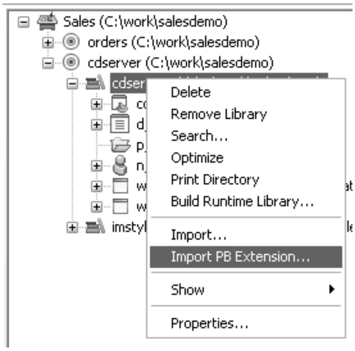
Adding pbdom110.pbx to your application

The PBDOM classes are implemented in a DLL file with the suffix PBX (for PowerBuilder extension). The simplest way to add the PBDOM classes to a PowerBuilder target is to import the object descriptions in the *pbdom110.pbx* PBX file into a library in the PowerBuilder System Tree. You can also the add *pbdom110.pbd* file, which acts as a wrapper for the classes, to the target’s library search path.

The *pbdom110.pbx* and *pbdom110.pbd* files are placed in the *Shared\PowerBuilder* directory when you install PowerBuilder. When you are building a PBDOM application, you do not need to copy *pbdom110.pbx* to another location, but you do need to deploy it with the application in a directory in the application’s search path.

❖ **To import the descriptions in an extension into a library:**

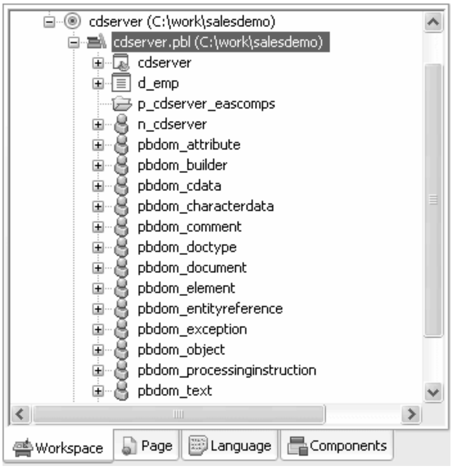
1 In the System Tree, expand the target in which you want to use the extension, right-click a library, and select Import PB Extension from the pop-up menu.



2 Navigate to the location of the PBX file and click Open.

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

After you import *pbdom110.pbx*, the PBDOM objects display in the System Tree:



Using PBDOM

This section describes how to accomplish basic tasks using PBDOM classes and methods. To check for complete code samples that you can download and test, select Programs>Sybase>PowerBuilder 11.0>PB 11 Code Samples from the Windows Start menu.

Validating the XML

Before you try to build a document from a file or string, you can test whether the XML is well formed or, optionally, whether it conforms to a DTD or Schema using the XMLParseFile or XMLParseString PowerScript functions. For example, this code tests whether the XML in a file is well formed:

```
long ll_ret
ll_ret = XMLParseFile("c:\temp\mydoc.xml", ValNever!)
```

By default, these functions display a message box if errors occur. You can also provide a *parsingerrors* string argument to handle them yourself. For more information about these functions, see their descriptions in the *PowerScript Reference* or the online Help.

Creating an XML document from XML

The PBDOM_BUILDER class provides three methods for creating a PBDOM_DOCUMENT from an existing XML source. It also provides the GetParseErrors method to get a list of any parsing errors that occur.

Using BuildFromString

The following example uses an XML string and the PBDOM_BUILDER class to create a PBDOM_DOCUMENT. First the objects are declared:

```
PBDOM_BUILDER pbdom_builder_new
PBDOM_DOCUMENT pbdom_doc
```

The objects are then instantiated using the constructor and the PBDOM_BUILDER BuildFromString method:

```
pbdombuilder_new = Create PBDOM_Builder
pbdom_doc = pbdombuilder_new.BuildFromString(Xml_doc)
```

XML can also be loaded directly into a string variable, as in the following example:

```
string Xml_str
Xml_str = "<?xml version="1.0" ?>"
Xml_str += "<WHITEPAPER>"
Xml_str += "<TITLE>Document Title</TITLE>"
Xml_str += "<AUTHOR>Author Name</AUTHOR>"
Xml_str += "<PARAGRAPH>Document text.</PARAGRAPH>"
Xml_str += "</WHITEPAPER>"
```

Using BuildFromFile

You can create an XML file using the BuildFromFile method and a string containing the path to a file from which to create a PBDOM_DOCUMENT:

```
PBDOM_BUILDER     pbdombuilder_new
PBDOM_DOCUMENT     pbdom_doc
pbdombuilder_new = Create PBDOM_Builder
pbdom_doc = pbdombuilder_new.BuildFromFile &
   ("c:\pbdom_doc_1.xml")
```

Using BuildFromDataStore

The following PowerScript code fragment demonstrates how to use the BuildFromDataStore method with a referenced DataStore object.

```
PBDOM_Builder pbdom_bldr
pbdom_document pbdom_doc
datastore ds

ds = Create datastore
ds.DataObject = "d_customer"
ds.SetTransObject (SQLCA)
ds.Retrieve
```

```
pbdom_doc = pbdom_bldr.BuildFromDataStore(ds)
```

Using GetParseErrors

After a call to any of the Build methods, you can obtain a list of parsing and validating errors encountered by the Build methods with the GetParseErrors method:

```
PBDOM_Builder pbdom_bldr
pbdom_document pbdom_doc
string strParseErrors[]
BOOLEAN bRetTemp = FALSE

pbdom_buildr = Create PBDOM_BUILDER
pbdom_doc = pbdom_buildr.BuildFromFile("D:\temp.xml")
bRetTemp = pbdom_buildr.GetParseErrors(strParseErrors)
if bRetTemp = true then
   for l = 1 to UpperBound(strParseErrors)
      MessageBox ("Parse Error", strParseErrors[l])
   next
end if
```

Parsing errors
If parsing errors are found and GetParseErrors returns true, a complete PBDOM node tree that can be inspected might still be created.

Creating an XML document from scratch

You can create an XML document in a script using the appropriate PBDOM_OBJECT subclasses and methods. The following code uses the PBDOM_ELEMENT and PBDOM_DOCUMENT classes and some of their methods to create a simple XML document.

First, the objects are declared and instantiated:

```
PBDOM_ELEMENT pbdom_elem_1
PBDOM_ELEMENT pbdom_elem_2
PBDOM_ELEMENT pbdom_elem_3
PBDOM_ELEMENT pbdom_elem_root
PBDOM_DOCUMENT pbdom_doc1

pbdom_elem_1 = Create PBDOM_ELEMENT
pbdom_elem_2 = Create PBDOM_ELEMENT
pbdom_elem_3 = Create PBDOM_ELEMENT
```

The instantiated objects are assigned names. Note that the PBDOM_DOCUMENT object pbdom_doc1 is not named:

```
pbdom_elem_1.SetName("pbdom_elem_1")
pbdom_elem_2.SetName("pbdom_elem_2")
pbdom_elem_3.SetName("pbdom_elem_3")
```

The objects are arranged into a node tree using the AddContent method. The AddContent method adds the referenced object as a child node under the object from which AddContent is invoked:

```
pbdom_elem_1.AddContent(pbdom_elem_2)
pbdom_elem_2.AddContent(pbdom_elem_3)
```

Use the NewDocument method to create a new XML document. The parameter value supplied to the NewDocument method becomes the name of the root element. This name is then accessed from the PBDOM_DOCUMENT object pbdom_doc1 and assigned to the PBDOM_ELEMENT object pbdom_elem_root using the GetRootElement method:

```
pbdom_doc1.NewDocument("Root_Element_From_Doc_1")
pbdom_elem_root = pbdom_doc1.GetRootElement()
```

The ELEMENT object pbdom_elem_1 and all its child nodes are placed in the new XML document node tree under the root element using the AddContent method. Note that as the ancestor node pbdom_elem_1 is placed in the node tree, all its child nodes move as well:

```
pbdom_elem_root.AddContent(pbdom_elem_1)
```

The XML document created looks like this:

```
<!DOCTYPE Root_Element_From_Doc_1>
<Root_Element_From_Doc_1>
    <pbdom_elem_1>
        <pbdom_elem_2>
            <pbdom_elem_3/>
        </pbdom_elem_2>
    </pbdom_elem_1>
</Root_Element_From_Doc_1>
```

Accessing node data

An XML document can be read by accessing the elements of its node tree using the appropriate PBDOM_OBJECT subclasses and methods. The following code uses an array, the PBDOM_OBJECT, and its descendant class PBDOM_DOCUMENT, and the GetContent and GetRootElement methods of the PBDOM_DOCUMENT class to access node data on an XML document.

A PBDOM_DOCUMENT object named pbdom_doc contains the following XML document:

```
<Root>
    <Element_1>
        <Element_1_1/>
        <Element_1_2/>
        <Element_1_3/>
    </Element_1>
    <Element_2/>
    <Element_3/>
</Root>
```

The following code declares an array to hold the elements returned from the GetContent method, which reads the PBDOM_DOCUMENT object named pbdom_doc:

```
PBDOM_OBJECT pbdom_obj_array[]
...
pbdom_doc.GetContent(ref pbdom_obj_array)
```

The pbdom_obj_array array now contains one value representing the root element of pbdom_doc: `<Root>`.

To access the other nodes in pbdom_doc, the GetRootElement method is used with the GetContent method.

```
pbdom_doc.GetRootElement().GetContent &
   (ref pbdom_obj_array)
```

The pbdom_obj_array array now contains three values corresponding to the three child nodes of the root element of pbdom_doc: `<Element_1>`, `<Element_2>`, and `<Element_3>`.

PBDOM provides other methods for accessing data, including InsertContent, AddContent, RemoveContent, and SetContent.

Changing node content with arrays

You can use the AddContent method to change node content:

```
pbdom_obj_array[3].AddContent("This is Element 3.")
```

This line of code changes the node tree as follows:

```
<Root>
    <Element_1>
        <Element_1_1/>
        <Element_1_2/>
        <Element_1_3/>
    </Element_1>
    <Element_2/>
    <Element_3>This is Element 3.</Element_3>
</Root>
```

Arrays and object references
When you use a method such as the GetContent method of the PBDOM_DOCUMENT class to return an array of PBDOM_OBJECT references, the references are to instantiated PBDOM objects. If you modify any of these objects through its array item, the changes are permanent and are reflected in any other arrays that hold the same object reference.

Manipulating the node-tree hierarchy

You can restructure an XML node tree by rearranging its nodes. One means of manipulating nodes involves detaching a child node from its parent node. This can be accomplished with the Detach method, as in the following example.

The root element of a PBDOM_DOCUMENT object named pbdom_doc is obtained using the GetRootElement method:

```
pbdom_obj = pbdom_doc.GetRootElement()
```

The root element is detached from the PBDOM_DOCUMENT object, which is the parent node of the root element:

```
pbdom_obj.Detach()
```

PBDOM provides the SetParentObject method to make an object a child of another object.

Checking for parent node

The GetParentObject method can be used to determine whether an element has a parent object, as in the following example:

```
pbdom_parent_obj = pbdom_obj.GetParentObject()
if not IsValid(pbdom_parent_obj) then
   MessageBox ("Invalid", "Root Element has no Parent")
end if
```

If the object on which GetParentObject is called has no parent object, the function returns NULL.

PBDOM provides similar methods that return information about an element's place in an XML node tree. These methods include HasChildren, which returns a boolean indicating whether an object has child objects, and IsAncestorObjectOf, which indicates whether an object is the ancestor of another object.

Handling PBDOM exceptions

PBDOM defines an exception class, PBDOM_EXCEPTION, derived from the standard PowerBuilder Exception class. The standard Text property of the Exception class can be used to obtain more detail on the nature of the exception being thrown. The class extends the PowerBuilder Exception class with one method, GetExceptionCode, that returns the unique code that identifies the exception being thrown.

For a list of exception codes, see the *PowerBuilder Extension Reference* or the topic PBDOM exceptions in the online Help.

PBDOM is a PowerBuilder extension, built using PBNI. The extension itself might throw a PBXRuntimeError exception. In the following example, the try-catch block checks first for a PBDOM exception, then for a PBXRuntimeError.

The example builds a PBDOM_DOCUMENT from a passed-in file name and uses a user-defined function called ProcessData to handle the DOM nodes. Process data could be a recursive function that extracts information from the DOM elements for further processing:

```
Long ll_ret

ll_ret = XMLParseFile(filename, ValNever!)
if ll_ret < 0 then return

PBDOM_Builder   domBuilder

TRY
   domBuilder = CREATE PBDOM_Builder
   PBDOM_Document domDoc
   PBDOM_Element   root
   domDoc = domBuilder.BuildFromFile( filename )
   IF IsValid( domDoc ) THEN
      IF domDoc.HasChildren() THEN
         PBDOM_Object data[]
         IF domDoc.GetContent( data ) THEN
            Long   ll_index, ll_count
            ll_count = UpperBound( data )
            FOR ll_index = 1 TO ll_count
               ProcessData( data[ll_index], 0 )
            NEXT
         END IF
      END IF
   END IF

CATCH ( PBDOM_Exception pbde )
   MessageBox( "PBDOM Exception", pbde.getMessage() )
CATCH ( PBXRuntimeError re )
   MessageBox( "PBNI Exception", re.getMessage() )
END TRY
```

XML namespaces

XML namespaces provide a way to create globally unique names to distinguish between elements and attributes with the same name but of different terminologies. For example, in an XML invoice document for a bookstore, the name "date" could be used by accounting for the date of the order and by order fulfillment for the date of publication.

An XML namespace is identified by a Uniform Resource Identifier (URI), a short string that uniquely identifies resources on the Web. The elements and attributes in each namespace can be uniquely identified by prefixing the element or attribute name (the local name) with the URI of the namespace.

Associating a prefix with a namespace

You declare an XML namespace using xmlns as part of a namespace declaration attribute. With the namespace declaration attribute, you can associate a prefix with the namespace.

For example, the following namespace declaration attribute declares the *http://www.pre.com* namespace and associates the prefix `pre` with this namespace:

```
xmlns:pre="http://www.pre.com"
```

Default XML namespace

If an XML namespace declaration does not specify a prefix, the namespace becomes a default XML namespace. For example, the following element, digicom, declares the namespace *http://www.digital_software.com*:

```
<digicom xmlns="http://www.digital_software.com" />
```

The namespace *http://www.digital_software.com* is the in-scope default namespace for the element digicom and any child elements that digicom might contain. The child elements of digicom will automatically be in this namespace.

The NONAMESPACE declaration

The following namespace declaration is known as the NONAMESPACE declaration:

```
xmlns=""
```

The containing element and its child elements are declared to be in no namespace. An element that is in the NONAMESPACE namespace has its namespace prefix and URI set to empty strings.

Initial state

When a PBDOM_ELEMENT or a PBDOM_ATTRIBUTE is first created, it has no name, and the namespace information is by default set to the NONAMESPACE namespace (that is, its namespace prefix and URI are both empty strings). The SetName method is used to set the local name and the SetNamespace method is used to set the namespace prefix and URI.

The name is required
The name is a required property of a PBDOM_ELEMENT and PBDOM_ATTRIBUTE, but the namespace information is not.

Retrieving from a parsed document

If a PBDOM_ELEMENT or PBDOM_ATTRIBUTE is retrieved programmatically from a parsed document, then its name and namespace information are inherited from the Element or Attribute contained in the parsed document. However, even after parsing, the name and namespace information of the PBDOM_ELEMENT and PBDOM_ATTRIBUTE can be further modified with the SetName and SetNamespace methods.

The name and namespace information are stored separately internally. Changing the name of a PBDOM_ELEMENT or PBDOM_ATTRIBUTE does not affect its namespace information, and changing its namespace information has no effect on its name.

Setting the name and namespace of a PBDOM_ATTRIBUTE

The W3C "Namespaces in XML" specification (in section 5.3) places restrictions on setting the name and namespace of a PBDOM_ATTRIBUTE. No tag can contain two attributes with identical names, or with qualified names that have the same local name and have prefixes that are bound to identical namespace names.

The specification provides the following examples of illegal and legal attributes:

```
<!-- http://www.w3.org is bound to n1 and n2 -->
<x xmlns:n1="http://www.w3.org"
   xmlns:n2="http://www.w3.org" >
   <bad a="1"  a="2" />
   <bad n1:a="1"  n2:a="2" />
</x>
```

```
<!-- http://www.w3.org is bound to n1 and is the default
-->
<x xmlns:n1="http://www.w3.org"
   xmlns="http://www.w3.org" >
   <good a="1"  b="2" />
   <good a="1"  n1:a="2" />
</x>
```

In the first example, `<bad a="1"  a="2" />` violates the rule that no tag can contain two attributes with identical names. In the second tag, the attributes have the same local name but different prefixes, so that their names are different. However, their prefixes point to the same namespace URI, *http://www.w3.org*, so it is illegal to place them inside the same owner element.

PBDOM scenarios

The following scenarios illustrate how PBDOM conforms to these requirements.

- When the PBDOM_ATTRIBUTE SetName method is invoked:

  If the PBDOM_ATTRIBUTE pbdom_attr1 has an owner PBDOM_ELEMENT that contains an existing PBDOM_ATTRIBUTE with the same name that is to be set for pbdom_attr1 and has the same namespace URI as pbdom_attr1, the EXCEPTION_INVALID_NAME exception is thrown.

- When the PBDOM_ATTRIBUTE SetNamespace method is invoked:

  If the PBDOM_ATTRIBUTE pbdom_attr1 has an owner PBDOM_ELEMENT that contains an existing PBDOM_ATTRIBUTE with the same name as pbdom_attr1 and the same namespace URI that is to be set for pbdom_attr1, the EXCEPTION_INVALID_NAME exception is thrown.

- When the PBDOM_ELEMENT SetAttribute(pbdom_attribute *pbdom_attribute_ref*) method is invoked:

  If the PBDOM_ELEMENT already contains an attribute that has the same name and namespace URI as the input PBDOM_ATTRIBUTE, the existing attribute is replaced by the input PBDOM_ATTRIBUTE. The existing attribute is thus removed (detached) from the owner element.

- When the PBDOM_ELEMENT SetAttributes(pbdom_attribute *pbdom_attribute_array[]*) method is invoked:

  If any two PBDOM_ATTRIBUTE objects in the array have the same name and namespace URI, the EXCEPTION_INVALID_NAME exception is thrown. If there is no name or namespace conflict within the array, all the existing attributes of the PBDOM_ELEMENT are replaced by the PBDOM_ATTRIBUTE objects in the array.

Note
All the above scenarios apply to PBDOM_ATTRIBUTE objects that are contained in the NONAMESPACE namespace.

- When the PBDOM_ELEMENT SetAttribute(string *strName*, string *strValue*) method is invoked:

  A new PBDOM_ATTRIBUTE with the specified name and value is created and set into the PBDOM_ELEMENT. If the PBDOM_ELEMENT already contains an attribute that has the same name and that is contained within the NONAMESPACE namespace, it is removed (detached) from the PBDOM_ELEMENT.

- When the PBDOM_ELEMENT SetAttribute(string *strName*, string *strValue*, string *strNamespacePrefix*, string *strNamespaceUri*, boolean *bVerifyNamespace*) method is invoked:

  A new PBDOM_ATTRIBUTE with the specified name, value, and namespace information is created and set into the PBDOM_ELEMENT. If the PBDOM_ELEMENT already contains a PBDOM_ATTRIBUTE that has the same name and namespace URI as the input namespace URI, it is removed (detached) from the PBDOM_ELEMENT.

Example

The following example demonstrates the impact of setting a PBDOM_ATTRIBUTE for a PBDOM_ELEMENT where the PBDOM_ELEMENT already contains an attribute of the same name and namespace URI as the input PBDOM_ATTRIBUTE.

The example creates a PBDOM_DOCUMENT based on the following document:

```
<root xmlns:pre1="http://www.pre.com"
xmlns:pre2="http://www.pre.com">
   <child1 pre1:a="123"/>
</root>
```

Then it creates a PBDOM_ATTRIBUTE object and set its name to `a` and its prefix and URI to `pre2` and `http://www.pre.com`. The *bVerifyNamespace* argument is set to FALSE because this PBDOM_ATTRIBUTE has not been assigned an owner PBDOM_ELEMENT yet, so that the verification for a predeclared namespace would fail. The text value is set to `456`.

The child1 element already contains an attribute named a that belongs to the namespace http://www.pre.com, as indicated by the prefix pre1. The new PBDOM_ATTRIBUTE uses the prefix pre2, but it represents the same namespace URI, so setting the new PBDOM_ATTRIBUTE to child1 successfully replaces the existing pre1:a with the new PBDOM_ATTRIBUTE pre2:a.

```
PBDOM_BUILDER pbdom_buildr
PBDOM_DOCUMENT pbdom_doc
PBDOM_ATTRIBUTE pbdom_attr

string strXML = "<root
xmlns:pre1=~"http://www.pre.com~"
xmlns:pre2=~"http://www.pre.com~"><child1
pre1:a=~"123~"/></root>"
```

```
try

  pbdom_buildr = Create PBDOM_BUILDER
  pbdom_doc = pbdom_buildr.BuildFromString (strXML)

  // Create a PBDOM_ATTRIBUTE and set its properties
  pbdom_attr = Create PBDOM_ATTRIBUTE
  pbdom_attr.SetName ("a")
  pbdom_attr.SetNamespace ("pre2", &
     "http://www.pre.com", false)
  pbdom_attr.SetText("456")

  // Attempt to obtain the child1 element and
  // set the new attribute to it
  pbdom_doc.GetRootElement(). &
    GetChildElement("child1").SetAttribute(pbdom_attr)

  pbdom_doc.SaveDocument &
     ("pbdom_elem_set_attribute_1.xml")

catch (PBDOM_EXCEPTION except)
  MessageBox ("PBDOM_EXCEPTION", except.GetMessage())
end try
```

The XML output from SaveDocument looks like the following :

```
<root xmlns:pre1="http://www.pre.com"
xmlns:pre2="http://www.pre.com">
   <child1 pre2:a="456"/>
</root>
```

CHAPTER 15 Manipulating Graphs

About this chapter

This chapter describes how to write code that allows you to access and change a graph in your application at runtime.

Contents

Using graphs

In PowerBuilder, there are two ways to display graphs:

- In a DataWindow, using data retrieved from the DataWindow data source
- In a graph control in a window or user object, using data supplied by your application code

This chapter discusses the graph control and describes how your application code can supply data for the graph and manipulate its appearance.

For information about graphs in DataWindows, see the *DataWindow Programmer's Guide* and the *DataWindow Reference*.

To learn about designing graphs and setting graph properties in the painters, see the PowerBuilder *User's Guide*.

Working with graph controls in code

Graph controls in a window can be enabled or disabled, visible or invisible, and can be used in drag and drop. You can also write code that uses events of graph controls and additional graph functions.

Properties of graph controls

You can access (and optionally modify) a graph by addressing its properties in code at runtime. There are two kinds of graph properties:

- **Properties of the graph definition itself** These properties are initially set in the painter when you create a graph. They include a graph's type, title, axis labels, whether axes have major divisions, and so on.
- **Properties of the data** These properties are relevant only at runtime, when data has been loaded into the graph. They include the number of series in a graph (series are created at runtime), colors of bars or columns for a series, whether the series is an overlay, text that identifies the categories (categories are created at runtime), and so on.

Events of graph controls

Graph controls have the events listed in Table 15-1.

Table 15-1: Graph control events

Clicked	DragLeave
Constructor	DragWithin
Destructor	GetFocus
DoubleClicked	LoseFocus
DragDrop	Other
DragEnter	RButtonDown

So, for example, you can write a script that is invoked when a user clicks a graph or drags an object on a graph (as long as the graph is enabled).

Functions for graph controls

You use the PowerScript graph functions in Table 15-2 to manipulate data in a graph.

Table 15-2: PowerScript graph functions

Function	Action
AddCategory	Adds a category
AddData	Adds a data point
AddSeries	Adds a series
DeleteCategory	Deletes a category
DeleteData	Deletes a data point
DeleteSeries	Deletes a series
ImportClipboard	Copies data from the clipboard to a graph
ImportFile	Copies the data in a text file to a graph

Function	Action
ImportString	Copies the contents of a string to a graph
InsertCategory	Inserts a category before another category
InsertData	Inserts a data point before another data point in a series
InsertSeries	Inserts a series before another series
ModifyData	Changes the value of a data point
Reset	Resets the graph's data

Populating a graph with data

This section shows how you can populate an empty graph with data.

Using AddSeries

You use AddSeries to create a series. AddSeries has this syntax:

graphName.**AddSeries** (*seriesName*)

AddSeries returns an integer that identifies the series that was created. The first series is numbered 1, the second is 2, and so on. Typically you use this number as the first argument in other graph functions that manipulate the series.

So to create a series named Stellar, code:

```
int SNum
SNum = gr_1.AddSeries("Stellar")
```

Using AddData

You use AddData to add data points to a specified series. AddData has this syntax:

graphName.**AddData** (*seriesNumber, value*, *categoryLabel*)

The first argument to AddData is the number assigned by PowerBuilder to the series. So to add two data points to the Stellar series, whose number is stored by the variable *SNum* (as shown above), code:

```
gr_1.AddData(SNum, 12, "Q1") // Category is Q1
gr_1.AddData(SNum, 14, "Q2") // Category is Q2
```

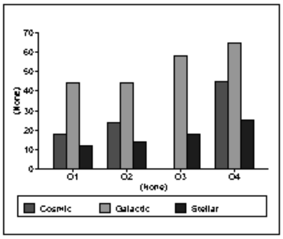
Getting a series number
You can use the FindSeries function to determine the number PowerBuilder has assigned to a series. FindSeries returns the series number. This is useful when you write general-purpose functions to manipulate graphs.

An example

Say you want to graph quarterly printer sales. Here is a script that populates the graph with data:

```
gr_1.Reset( All! ) // Resets the graph.
// Create first series and populate with data.

int SNum
SNum = gr_1.AddSeries("Stellar")
gr_1.AddData(SNum, 12, "Q1") // Category is Q1.
gr_1.AddData(SNum, 14, "Q2") // Category is Q2.
gr_1.Adddata(SNum, 18, "Q3") // Category is Q3.
gr_1.AddData(SNum, 25, "Q4") // Category is Q4.
// Create second series and populate with data.
SNum = gr_1.AddSeries("Cosmic")

// Use the same categories as for series 1 so the data
// appears next to the series 1 data.
gr_1.AddData(SNum, 18, "Q1")
gr_1.AddData(SNum, 24, "Q2")
gr_1.Adddata(SNum, 38, "Q3")
gr_1.AddData(SNum, 45, "Q4")

// Create third series and populate with data.
SNum = gr_1.AddSeries("Galactic")
gr_1.AddData(SNum, 44, "Q1")
gr_1.AddData(SNum, 44, "Q2")
gr_1.Adddata(SNum, 58, "Q3")
gr_1.AddData(SNum, 65, "Q4")
```

Here is the resulting graph:



You can add, modify, and delete data in a graph in a window through graph functions anytime during execution.

For more information

For complete information about each graph function, see the *PowerScript Reference*.

Modifying graph properties

When you define a graph in the Window or User Object painter, you specify its behavior and appearance. For example, you might define a graph as a column graph with a certain title, divide its Value axis into four major divisions, and so on. Each of these entries corresponds to a property of a graph. For example, all graphs have an enumerated attribute GraphType, which specifies the type of graph.

When dynamically changing the graph type
If you change the graph type, be sure to change other properties as needed to define the new graph properly.

You can change these graph properties at runtime by assigning values to the graph's properties in scripts. For example, to change the type of the graph gr_emp to Column, you could code:

```
gr_emp.GraphType = ColGraph!
```

To change the title of the graph at runtime, you could code:

```
gr_emp.Title = "New title"
```

How parts of a graph are represented

Graphs consist of parts: a title, a legend, and axes. Each of these parts has a set of display properties. These display properties are themselves stored as properties in a subobject (structure) of Graph called grDispAttr.

For example, graphs have a Title property, which specifies the title's text. Graphs also have a property TitleDispAttr, of type grDispAttr, which itself contains properties that specify all the characteristics of the title text, such as the font, size, whether the text is italicized, and so on.

Similarly, graphs have axes, each of which also has a set of properties. These properties are stored in a subobject (structure) of Graph called grAxis. For example, graphs have a property Values of type grAxis, which contains properties that specify the Value axis's properties, such as whether to use autoscaling of values, the number of major and minor divisions, the axis label, and so on.

Here is a representation of the properties of a graph:

```
Graph
      int Height
      int Depth
      grGraphType GraphType
      boolean Border
      string Title
      …
grDispAttr TitleDispAttr, LegendDispAttr, PieDispAttr
      string FaceName
      int TextSize
      boolean Italic
      …
grAxis Values, Category, Series
      boolean AutoScale
      int MajorDivisions
      int MinorDivisions
      string Label
      …
```

Referencing parts of a graph

You use dot notation to reference these display properties. For example, one of the properties of a graph's title is whether the text is italicized or not. That information is stored in the boolean Italic property in the TitleDispAttr property of the graph.

For example, to italicize title of graph gr_emp, code:

```
gr_emp.TitleDispAttr.Italic = TRUE
```

Similarly, to turn on autoscaling of a graph's Values axis, code:

```
gr_emp.Values.Autoscale = TRUE
```

To change the label text for the Values axis, code:

```
gr_emp.Values.Label = "New label"
```

To change the alignment of the label text in the Values axis, code:

```
gr_emp.Values.LabelDispAttr.Alignment = Left!
```

For a complete list of graph properties, see *Objects and Controls* or use the Browser.

For more about the Browser, see the *PowerBuilder User's Guide*.

Accessing data properties

To access properties related to a graph's data during execution, you use PowerScript graph functions. The graph functions related to data fall into several categories:

- Functions that provide information about a graph's data
- Functions that save data from a graph
- Functions that change the color, fill patterns, and other visual properties of data

How to use the functions

To call functions for a graph in a graph control, use the following syntax:

graphControlName.FunctionName (*Arguments*)

For example, to get a count of the categories in the window graph gr_printer, code:

```
Ccount = gr_printer.CategoryCount()
```

Different syntax for graphs in DataWindows

The syntax for the same functions is more complex when the graph is in a DataWindow, like this:

DataWindowName.FunctionName ("*graphName*", *otherArguments…*)

For more information, see the *DataWindow Programmer's Guide*.

Getting information about the data

The PowerScript functions in Table 15-3 allow you to get information about data in a graph at runtime.

Table 15-3: PowerScript functions for information at runtime

Function	Information provided
CategoryCount	The number of categories in a graph
CategoryName	The name of a category, given its number
DataCount	The number of data points in a series
FindCategory	The number of a category, given its name
FindSeries	The number of a series, given its name
GetData	The value of a data point, given its series and position (superseded by GetDataValue, which is more flexible)
GetDataPieExplode	The percentage by which a pie slice is exploded

Function	Information provided
GetDataStyle	The color, fill pattern, or other visual property of a specified data point
GetDataValue	The value of a data point, given its series and position
GetSeriesStyle	The color, fill pattern, or other visual property of a specified series
ObjectAtPointer	The graph element over which the mouse was positioned when it was clicked
SeriesCount	The number of series in a graph
SeriesName	The name of a series, given its number

Saving graph data

The PowerScript functions in Table 15-4 allow you to save data from the graph.

Table 15-4: PowerScript functions for saving graph data

Function	Action
Clipboard	Copies a bitmap image of the specified graph to the clipboard
SaveAs	Saves the data in the underlying graph to the clipboard or to a file in one of a number of formats

Modifying colors, fill patterns, and other data

The PowerScript functions in Table 15-5 allow you to modify the appearance of data in a graph.

Table 15-5: PowerScript functions for changing appearance of data

Function	Action
ResetDataColors	Resets the color for a specific data point
SetDataPieExplode	Explodes a slice in a pie graph
SetDataStyle	Sets the color, fill pattern, or other visual property for a specific data point
SetSeriesStyle	Sets the color, fill pattern, or other visual property for a series

Using point and click

Users can click graphs during execution. PowerScript provides a function called ObjectAtPointer that stores information about what was clicked. You can use this function in a number of ways in Clicked scripts. For example, you can provide the user with the ability to point and click on a data value in a graph and see information about the value in a message box. This section shows you how.

Clicked events and graphs

To cause actions when a user clicks a graph, you write a Clicked script for the graph control. The control must be enabled. Otherwise, the Clicked event does not occur.

Using ObjectAtPointer

ObjectAtPointer has the following syntax.

graphName.**ObjectAtPointer** (*seriesNumber, dataNumber*)

You should call ObjectAtPointer in the first statement of a Clicked script.

When called, ObjectAtPointer does three things:

- It returns the kind of object clicked on as a grObjectType enumerated value. For example, if the user clicks on a data point, ObjectAtPointer returns TypeData!. If the user clicks on the graph's title, ObjectAtPointer returns TypeTitle!.

  For a complete list of the enumerated values of grObjectType, open the Browser and click the Enumerated tab.

- It stores the number of the series the pointer was over in the variable *seriesNumber*, which is an argument passed by reference.
- It stores the number of the data point in the variable *dataNumber*, also an argument passed by reference.

After you have the series and data point numbers, you can use other graph functions to get or provide information. For example, you might want to report to the user the value of the clicked data point.

Example

Assume there is a graph gr_sale in a window. The following script for its Clicked event displays a message box:

- If the user clicks on a series (that is, if ObjectAtPointer returns TypeSeries!), the message box shows the name of the series clicked on. The script uses the function SeriesName to get the series name, given the series number stored by ObjectAtPointer.

- If the user clicks on a data point (that is, if ObjectAtPointer returns TypeData!), the message box lists the name of the series and the value clicked on. The script uses GetData to get the data's value, given the data's series and data point number:

```
int SeriesNum, DataNum
double Value
grObjectType ObjectType
string SeriesName, ValueAsString

// The following function stores the series number
// clicked on in SeriesNum and stores the number
// of the data point clicked on in DataNum.
ObjectType = &
       gr_sale.ObjectAtPointer (SeriesNum, DataNum)

IF ObjectType = TypeSeries! THEN
       SeriesName = gr_sale.SeriesName (SeriesNum)
       MessageBox("Graph", &
          "You clicked on the series " + SeriesName)

ELSEIF ObjectType = TypeData! THEN
       Value = gr_sale. GetData (SeriesNum, DataNum)
       ValueAsString = String(Value)
       MessageBox("Graph", &
          gr_sale. SeriesName (SeriesNum) + &
          " value is " + ValueAsString)
END IF
```

CHAPTER 16 Implementing Rich Text

About this chapter

This chapter explains how to use rich text in an application, either in a RichText DataWindow object or in a RichTextEdit control.

Contents

Before you begin

This chapter assumes you know how to create RichText DataWindow objects and RichTextEdit controls, as described in the PowerBuilder *User's Guide*.

Using rich text in an application

Rich text format (RTF) is a standard for specifying formatting instructions and document content in a single ASCII document. An editor that supports rich text format interprets the formatting instructions and displays the text with formatting.

In an application, you may want to:

- Provide a window for preparing rich text documents

  Although not a full-fledged word processor, the RichTextEdit control allows the user to apply formatting to paragraphs, words, and characters.

- Create a mail-merge application

  You or the user can set up boilerplate text with input fields associated with database data.

- Display reports with formatted text

  A RichText DataWindow object is designed for viewing data, rather than entering data. It does not have the edit styles of other DataWindow presentation styles.

- Store rich text as a string in a database and display it in a RichTextEdit control

Sources of rich text

Any word processor

You can prepare rich text in any word processor that can save or export rich text format.

Input fields in PowerBuilder only

Although many word processors support some kinds of fields, the fields are usually incompatible with other rich text interpreters. If you want to specify input fields for a PowerBuilder application, you will have to insert them using the PowerBuilder RichTextEdit control.

Rich text in the database

Since rich text is represented by ASCII characters, you can also store rich text in a string database column or string variable. You can retrieve rich text from a string database column and use the PasteRTF function to display the text with formatting in a RichTextEdit control.

Deploying a rich text application

When you deploy a rich text application to a server or client machine, you must also copy the rich text DLL and OCX files in the *Sybase\Shared\PowerBuilder\RTC* directory to the PowerBuilder VM directory on the deployment machine or to a directory in the application's path. Table 41-5 on page 776 provides the list of files you must copy.

You can use the PowerBuilder Runtime Packager to deploy the required rich text files with your application. However, you must also register the *tp4ole11.ocx* file on the deployment machine.

For more information on the runtime packager, see "PowerBuilder Runtime Packager" on page 771.

Using a RichText DataWindow object

This section discusses:

- How scrolling differs from other DataWindow styles
- Problems you may encounter with default values for new rows
- What happens when the user makes changes

Scrolling

In a RichText DataWindow object, the rich text can consist of more than one page. A row of data can be associated with several pages, making a row larger than a page. In other DataWindow styles, a page consists of one or more than one row—a page is larger than a row.

For a RichText DataWindow object, the scrolling functions behave differently because of this different meaning for a page:

- ScrollNextRow and ScrollPriorRow still scroll from row to row so that another row's data is displayed within the document template.
- ScrollNextPage and ScrollPriorPage scroll among pages of the document rather than pages of rows.

Page flow As you scroll, the pages appear to flow from one row to the next. Scrolling to the next page when you are on the last page of the document takes you to the first page for the next row. The user gets the effect of scrolling through many instances of the document.

New rows: default data and validation rules

Input fields are invisible when they have no value. Before data is retrieved, PowerBuilder displays question marks (??) in fields to make them visible. For new rows, PowerBuilder assigns an initial value based on the datatype.

If you have specified an initial value for the column, PowerBuilder uses that value; if no value is specified, PowerBuilder uses spaces for string columns or zero for numeric columns.

Possible validation errors If the default initial value provided by PowerBuilder *does not satisfy* the validation rule, the user gets a validation error as soon as the new row is inserted. To avoid this, you should specify initial values that meet your validation criteria.

When the user makes changes

Display only When you check Display Only on the General property page for the Rich Text Object, the user cannot make any changes to the data or the rich text.

If you leave the pop-up menu enabled, the user can turn off the display-only setting and make the DataWindow object editable.

Input fields In an editable DataWindow object, users change the value of a column input field by displaying the input field's property sheet and editing the Data Value text box. For a computed field input field, the Data Value text box is read-only.

You can let the user display input field names instead of data. You might do this if you were providing an editing environment in which users were developing their own RichText DataWindow object. However, the RichTextEdit control is better suited to a task like this, because you have more scripting control over the user's options.

Rich text If users edit the text or formatting, they are changing the document template. The changes are seen for every row.

The changes apply to that session only, unless you take extra steps to save the changes and restore them.

To save the changes, you can write a script that uses the CopyRTF function to get all the text, including the named input fields but not the row data, and save the contents of that string in a file or database. Whenever users view the RichText DataWindow object, you can restore their latest version or let them return to the original definition of the DataWindow object's text.

Functions for RichText DataWindow objects

The DataWindow control has many functions.

Functions that behave the same DataWindow control functions that operate on data, such as Update or Retrieve, have the same behavior for all types of DataWindow objects.

When the object in the control is a RichText DataWindow object, some of the functions do not apply or they behave differently.

Functions that do not apply Some functions are not applicable when the object is a RichText DataWindow object. The following functions return an error or have no effect:

- Functions for graph and crosstab DataWindow objects
- Functions for grouping: GroupCalc, FindGroupChange
- Functions for code tables: GetValue, SetValue
- Functions for selecting rows: SelectRow, SetRowFocusIndicator, GetSelectedRow
- Functions that affect column and detail band appearance: SetBorderStyle, SetDetailHeight

- ObjectAtPointer
- OLEActivate

Functions that behave differently Some functions have different behavior when the object is a RichText DataWindow object:

- Functions for the clipboard: Copy, Clear, and so on
- Functions for editable text (they apply to the edit control in other DataWindow styles): LineCount, Position, SelectText, and so on
- Find and FindNext (the arguments you specify for Find determine whether you want the general DataWindow Find function or the RichText version)
- Scrolling

Using a RichTextEdit control

A RichTextEdit control in a window or user object lets the user view or edit formatted text. Functions allow you to manipulate the contents of the control by inserting text, getting the selected text, managing input fields, and setting properties for all or some of the contents.

You define RichTextEdit controls in the Window painter or the User Object painter.

Giving the user control

In the Window or User Object painter, on the Document page of the RichTextEdit control's property sheet, you can enable or disable the features in Table 16-1.

Table 16-1: RichTextEdit control features

Features	Details
Editing bars	A toolbar for text formatting, a ruler bar, and a status bar.
Pop-up menu	Provides access to the InsertFile and clipboard commands, as well as the property sheet.
Display of nonprinting characters	Carriage returns, tabs, and spaces.

Features	Details
Display of fields	Whether fields are visible at all, or whether the field name or data displays. You can also change the background color for fields.
Wordwrap	Affects newly entered text only. If the user enters new text in an existing paragraph, word wrap is triggered when the text reaches the right edge of the control. To get existing text to wrap within the display, the user can tweak the size of the control (if it is resizable).
Print margins	Print margins can be set relative to the default page size.

You can also specify a name for the document that is displayed in the print queue. The document name has nothing to do with a text file you might insert in the control.

Users can change the available tools

When users display the property sheet for the rich text document, they can change the tools that are available to them, which you might not want. For example, they might:

- Remove the display-only setting so that they can begin editing a document you set up as protected
- Turn off the tool, ruler, or status bars
- View input fields' names instead of data
- Disable the pop-up menu so that they cannot restore tools they turn off

You might want to guard against some of these possibilities. You can reset the property values for these settings in a script. For example, this statement restores the pop-up menu when triggered in an event script:

```
rte_1.PopMenu = TRUE
```

Undoing changes

The user can press Ctrl+Z to undo a change. You can also program a button or menu item that calls the Undo function.

If Undo is called repeatedly, it continues to undo changes to a maximum of 50 changes. The script can check whether there are changes that can be undone (meaning the maximum depth has not been reached) by calling the CanUndo function:

```
IF rte_1.CanUndo() THEN
   rte_1.Undo()
ELSE
   MessageBox("Stop", "Nothing to undo.")
END IF
```

Text for the control

In the Window painter, you do not enter text in the control. Instead, in your application you can programmatically insert text or let the user enter text using the editing tools.

Setting a default font
The Font tab page in the Properties view for a RichTextEdit control lets you set default font characteristics for the control. When the control first displays at runtime, and you include the toolbar with a RichTextEdit control, the toolbar indicates the default font characteristics that you selected on the Font tab page at design time. Although the application user can change fonts at runtime, or you can use PowerScript to change the font style, you can set the default font at design time only.

Inserting text

From a file If you have prepared a text file for your application, you can insert it with the InsertDocument function. The file can be rich text or ASCII:

```
li_rtn = rte_1.InsertDocument  &
   ("c:\mydir\contacts.rtf", FALSE, FileTypeRichText!)
```

The boolean *clearflag* argument lets you specify whether to insert the file into existing text or replace it. If you want to include headers and footers from a document that you insert, you must replace the existing text by setting the *clearflag* argument to TRUE. (The InsertFile command on the runtime pop-up menu is equivalent to the InsertDocument function with the *clearflag* argument set to FALSE.)

DisplayOnly property must be set to false
You cannot insert a document into a rich text control when the control's DisplayOnly property is set to true. If you try to do this, PowerBuilder displays a runtime error message.

From a database If you have saved rich text as a string in a database, you can use a DataStore to retrieve the text.

After retrieving data, paste the string into the RichTextEdit control:

```
ls_desc = dw_1.Object.prod_desc.Primary[1]
rte_1.PasteRTF(ls_desc)
```

Rich text and the clipboard
The CopyRTF and PasteRTF functions let you get rich text *with formatting instructions* and store it in a string. If you use the clipboard by means of the Copy, Cut, and Paste functions, you get the text only—the formatting is lost.

Example of saving rich text in a database

Suppose you have a database table that records tech support calls. Various fields record each call's date, support engineer, and customer. Another field stores notes about the call. You can let the user record notes with bold and italic formatting for emphasis by storing rich text instead of plain text.

The window for editing call information includes these controls:

- A DataWindow control that retrieves all the data and displays everything except the call notes
- A RichTextEdit control that displays the call notes
- A button for updating the database

RowFocusChanged event As row focus changes, the notes for the current row are pasted into the RichTextEdit control. The RowFocusChanged event has this script:

```
string ls_richtext

// Get the string from the call_notes column
ls_richtext = dw_1.Object.call_notes[currentrow]

// Prevent flicker
rte_1.SetRedraw(FALSE)

// Replace the old text with text for the current row
rte_1.SelectTextAll()
rte_1.Clear()
rte_1.PasteRTF(ls_richtext)
rte_1.SetRedraw(TRUE)
```

LoseFocus event When the user makes changes, the changes are transferred to the DataWindow control. It is assumed that the user will click on the button or the DataWindow control when the user is through editing, triggering the LoseFocus event, which has this script:

```
string ls_richtext
long l_currow
GraphicObject l_control

// Check whether RichTextEdit still has focus
// If so, don't transfer the text
l_control = GetFocus()

IF TypeOf(l_control) = RichTextEdit! THEN RETURN 0

// Prevent flicker
rte_1.SetRedraw(FALSE)

// Store all the text in string ls_richtext
ls_richtext = rte_1.CopyRTF()

// Assign the rich text to the call_notes column
// in the current row
l_currow = dw_1.GetRow()
dw_1.Object.call_notes[l_currow] = ls_richtext
rte_1.SetRedraw(TRUE)
```

LoseFocus and the toolbars
A LoseFocus event occurs for the RichTextEdit control even when the user clicks a RichTextEdit toolbar. Technically, this is because the toolbars are in their own windows. However, the RichTextEdit control still has focus, which you can check with the GetFocus function.

Saving rich text in a file

You can save the rich text in the control, with the input field definitions, with the SaveDocument function. You have the choice of rich text format (RTF) or ASCII:

```
rte_1.SaveDocument("c:\...\contacts.rtf", &
   FileTypeRichText!)
```

SaveDocument does not save the data in the input fields. It saves the document template.

Does the file exist? If the file exists, calling SaveDocument triggers the FileExists event. In the event script, you might ask users if they want to overwrite the file.

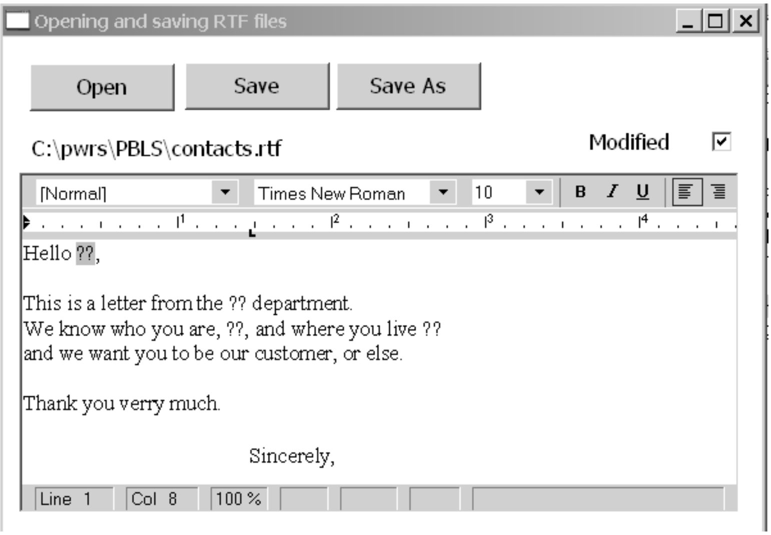
To cancel the saving process, specify a return code of 1 in the event script.

Are there changes that need saving? The Modified property indicates whether any changes have been made to the contents of the control. It indicates that the contents are in an unsaved state. When the first change occurs, PowerBuilder triggers the Modified event and sets the Modified property to TRUE. Calling SaveDocument sets Modified to FALSE, indicating that the document is clean.

Opening a file triggers the Modified event and sets the property because the control's contents changed. Usually, though, what you really want to know is whether the contents of the control still correspond to the contents of the file. Therefore, in the script that opens the file, you can set the Modified property to FALSE yourself. Then when the user begins editing, the Modified event is triggered again and the property is reset to TRUE.

Opening and saving files: an example

This example consists of several scripts that handle opening and saving files. Users can open existing files and save changes. They can also save the contents to another file. If users save the file they opened, saving proceeds without interrupting the user. If users save to a file name that exists, but is not the file they opened, they are asked whether to overwrite the file:



Hello ??,

This is a letter from the ?? department.
We know who you are, ??, and where you live ??
and we want you to be our customer, or else.

Thank you verry much.

Sincerely,

The example includes instance variable declarations, scripts, functions, and events.

Instance variable declarations

ib_saveas A flag for the FileExists event. When FALSE, the user is saving to the file that was opened, so overwriting is expected:

```
boolean ib_saveas=FALSE
```

is_filename The current file name for the contents, initially set to "Untitled":

```
string is_filename
```

Open Document script

This script opens a file chosen by the user. Since opening a file triggers the Modified event and sets the Modified property, the script resets Modified to FALSE. The Checked property of the Modified check box is set to FALSE too:

```
integer li_answer, li_result
string ls_name, ls_path

li_answer = GetFileOpenName("Open File", ls_path, &
   ls_name, "rtf", &
   "Rich Text(*.RTF),*.RTF, Text files(*.TXT),*.TXT")

IF li_answer = 1 THEN
   // User did not cancel
   li_result = rte_1.InsertDocument(ls_path, TRUE)

   IF li_result = 1 THEN  // Document open successful
      // Save and display file name
      is_filename = ls_path
      st_filename.Text = is_filename

      // Save and display modified status
      rte_1.Modified = FALSE

      cbx_modified.Checked = rte_1.Modified
   ELSE
      MessageBox("Error", "File not opened.")
   END IF

   END IF
RETURN 0
```

Scripts that save the document

The user might choose to save the document to the same name or to a new name. These scripts could be assigned to menu items as well as buttons. The Save button script checks whether the instance variable *is_filename* holds a valid name. If so, it passes that file name to the of_save function. If not, it triggers the SaveAs button's script instead:

```
integer li_result
string ls_name

// If not associated with file, get file name
IF is_filename = "Untitled" THEN
   cb_saveas.EVENT Clicked()
```

```
ELSE
   li_result = Parent.of_save(is_filename)
END IF
RETURN 0
```

The SaveAs script sets the instance variable *ib_saveas* so that the FileExists event, if triggered, knows to ask about overwriting the file. It calls of_getfilename to prompt for a file name before passing that file name to the of_save function.

```
integer li_result
string ls_name

ib_saveas = TRUE

ls_name = Parent.of_getfilename()
// If the user canceled or an error occurred, abort
IF ls_name = "" THEN RETURN -1

li_result = Parent.of_save(ls_name)

ib_saveas = FALSE
RETURN 0
```

Functions for saving and getting a file name

of_save function This function accepts a file name argument and saves the document. It updates the file name instance variable with the new name and sets the check box to correspond with the Modified property, which is automatically set to FALSE after you call SaveDocument successfully:

```
integer li_result

MessageBox("File name", as_name)

// Don't need a file type because the extension
// will trigger the correct type of save
li_result = rte_1.SaveDocument(as_name)

IF li_result = -1 THEN
   MessageBox("Warning", "File not saved.")
   RETURN -1
ELSE
   // File saved successfully
   is_filename = as_name
   st_filename.Text = is_filename
   cbx_modified.Checked = rte_1.Modified
   RETURN 1
END IF
```

of_getfilename function The function prompts the user for a name and returns the file name the user selects. It is called when a file name has not yet been specified or when the user chooses Save As. It returns a file name:

```
integer li_answer
string ls_name, ls_path

li_answer = GetFileSaveName("Document Name", ls_path, &
   ls_name, "rtf", &
   "Rich Text(*.RTF),*.RTF,Text files(*.TXT),*.TXT")

IF li_answer = 1 THEN
   // Return specified file name
   RETURN ls_path
ELSE
   RETURN ""
END IF
```

Events for saving and closing

FileExists event When the user has selected a file name and the file already exists, this script warns the user and allows the save to be canceled. The event occurs when SaveDocument tries to save a file and it already exists. The script checks whether *ib_saveas* is TRUE and, if so, asks if the user wants to proceed with overwriting the existing file:

```
integer li_answer

// If user asked to Save to same file,
// don't prompt for overwriting
IF ib_saveas = FALSE THEN RETURN 0

li_answer = MessageBox("FileExists", &
   filename + " already exists. Overwrite?", &
   Exclamation!, YesNo!)

// Returning a non-zero value cancels save
IF li_answer = 2 THEN RETURN 1
```

Modified event This script sets a check box so the user can see that changes have not been saved. The Modified property is set automatically when the event occurs. The event is triggered when the first change is made to the contents of the control:

```
cbx_modified.Checked = TRUE
```

CloseQuery event This script for the window's CloseQuery event checks whether the control has unsaved changes and asks whether to save the document before the window closes:

```
integer li_answer

// Are there unsaved changes? No, then return.
IF rte_1.Modified = FALSE THEN RETURN 0

// Ask user whether to save
li_answer = MessageBox("Document not saved", &
   "Do you want to save " + is_filename + "?", &
   Exclamation!, YesNo! )

IF li_answer = 1 THEN
   // User says save. Trigger Save button script.
   cb_save.EVENT Clicked()
END IF
RETURN 0
```

Using an ActiveX spell checking control

ActiveX controls can be used to spell check text in a RichTextEdit control. The supported ActiveX spell checking controls include VSSpell from ComponentOne and WSpell from Wintertree Software.

You can use the SelectedStartPos and SelectedTextLength properties of the RichTextEdit control to highlight the current position of a misspelled word in a text string that you are parsing with a supported ActiveX spell checking control. The following procedure uses an ActiveX control to spell check the entire text of the current band of a RichTextEdit control.

❖ **To spell check selected text in a RichTextEdit control:**

1. On a window with a RichTextEdit control, select Insert>Control>OLE from the window menu.
2. Click the Insert Control tab of the Insert Object dialog box, select the installed ActiveX spell checking control, and click OK.
3. Click inside the window in the Window painter to insert the ActiveX control.

   By default, the name of the inserted control is ole_*n*, where *n* = 1 when there are no other OLE controls on the window.

4 Add a menu item to a menu that you associate with the current window and change its Text label to `Check Spelling`.

5 Add the following code the the Clicked event of the menu item, where *windowName* is the name of the window containing the RichTextEdit and ActiveX controls:

```
string ls_selected
//get the current band context, and leave select mode
windowName.rte_1.selecttext(0,0,0,0)
windowName.rte_1.SelectTextAll()
ls_selected = windowName.rte_1.SelectedText()
windowName.rte_1.SelectedTextLength = 0
//assign the string content to the ActiveX control
windowName.ole_1.object.text = ls_selected
windowName.ole_1.object.start()
```

6 Select the ActiveX control in the Window painter and select ReplaceWord from the event list for the control.

7 Add the following code to the ReplaceWord event script:

```
string str
str = this.object.MisspelledWord
rte_1.SelectedStartPos = this.object.WordOffset
rte_1.SelectedTextLength  = Len(str)
rte_1.ReplaceText(this.object.ReplacementWord)
messagebox("misspelled word", "replaced")
```

The next time you run the application, you can click the Check Spelling menu item to spell check the entire contents of the current band of the RichTextEdit control.

Formatting of rich text

In a RichText control, there are several user-addressable objects:

- The whole document
- Selected text and paragraphs
- Input fields
- Pictures

The user can make selections, use the toolbars, and display the property sheets for these objects.

Input fields get values either because the user or you specify a value or because you have called DataSource to associate the control with a DataWindow object or DataStore.

Input fields

An input field is a named value. You name it and you determine what it means by setting its value. The value is associated with the input field name. You can have several fields with the same name and they all display the same value. If the user edits one of them, they all change.

In this sample text, an input field for the customer's name is repeated throughout:

```
Hello {customer}!
```

We know that you, {customer}, will be excited about our new deal. Please call soon, {customer}, and save money now.

In a script, you can set the value of the customer field:

```
rte_1.InputFieldChangeData("customer", "Mary")
```

Then the text would look like this:

> Hello Mary!
> We know that you, Mary, will be excited about our new deal. Please call soon, Mary, and save money now.

The user can also set the value. There are two methods:

- Selecting it and typing a new value
- Displaying the Input Field property sheet and editing the Data Value text box

Inserting input fields in a script The InputFieldInsert function inserts a field at the insertion point:

```
rtn = rte_1.InputFieldInsert("datafield")
```

In a rich text editing application, you might want the user to insert input fields. The user needs a way to specify the input field name.

In this example, the user selects a name from a ListBox containing possible input field names. The script inserts an input field at the insertion point using the selected name:

```
string ls_field
integer rtn
```

```
ls_field = lb_fields.SelectedItem()
IF ls_field <> "" THEN
   rtn = rte_1.InputFieldInsert( ls_field )
   IF rtn = -1 THEN
   MessageBox("Error", "Cannot insert field.")
   END IF
ELSE
   MessageBox("No Selection", &
      "Please select an input field name.")
END IF
```

Input fields for dates and page numbers

To display a date or a page number in a printed document, you define an input field and set the input field's value.

❖ **To include today's date in the opening of a letter, you might:**

1 Create an input field in the text. Name it anything you want.

2 In the script that opens the window or some other script, set the value of the input field to the current date.

For example, if the body of the letter included an input field called TODAY, you would write a script like the following to set it:

```
integer li_rtn
li_rtn = rte_1.InputFieldChangeData( "today", &
   String(Today()) )
```

For information about setting page number values when printing, see "Preview and printing" on page 280.

Using database data

You can make a connection between a RichTextEdit control and a DataWindow control or DataStore object. When an input field in the RichTextEdit control has the same name as a column or computed column in the DataWindow object, it displays the associated data.

Whether or not the RichTextEdit control has a data source, there is always only *one copy* of the rich text content. While editing, you might visualize the RichTextEdit contents as a template into which row after row of data can be inserted. While scrolling from row to row, you might think of many instances of the document in which the text is fixed but the input field data changes.

To share data between a DataWindow object or DataStore, use the DataSource function:

```
rte_1.DataSource(ds_empdata)
```

Example of sharing data

If the DataWindow object associated with the DataStore ds_empdata has the four columns emp_id, emp_lname, emp_fname, and state, the RichTextEdit content might include text and input fields like this:

Sample letter with columns from the employee table

ID: {emp_id}

Dear {emp_fname} {emp_lname}:

We are opening a new plant in Mexico. If you would like to transfer from {state} to Mexico, the company will cover all expenses.

Navigating rows and pages

For the RichTextEdit control, navigation keys let the user move among the pages of the document. However, you must provide scrolling controls so that the user can move from row to row.

You should provide Prior Row and Next Row buttons. The scripts for the buttons are simple. For Next Row:

```
rte_1.ScrollNextRow()
```

For Prior Row:

```
rte_1.ScrollPriorRow()
```

If you also provide page buttons, then when the user is on the last page of the document for one row, scrolling to the next page moves to the first page for the next row:

```
rte_1.ScrollNextPage()
```

Cursor position in the RichTextEdit control

Functions provide several ways to find out what is selected and to select text in the RichTextEdit control.

Where is the insertion point or what is selected?

The text always contains an insertion point and it can contain a selection, which is shown as highlighted text. When there is a selection, the position of the insertion point can be at the start or the end of the selection, depending on how the selection is made. If the user drags from beginning to end, the insertion point is at the end. If the user drags from end to beginning, the insertion point is at the beginning.

The Position function provides information about the selection and the insertion point.

For more information, see Position in the *PowerScript Reference*.

Changing the cursor image

The Pointer page of the Rich Text Object property sheet has a list box with stock pointers that can be used to indicate cursor position in a RichTextEdit control or RichText DataWindow. Users can change the cursor image at runtime by selecting one of these pointers and clicking OK in the Rich Text Object property sheet.

Selecting text programmatically

There are several functions that select portions of the text relative to the position of the insertion point:

- SelectTextWord
- SelectTextLine
- SelectTextAll

A more general text selection function is SelectText. You specify the line and character number of the start and end of the selection.

Passing values to SelectText Because values obtained with Position provide more information than simply a selection range, you cannot pass the values directly to SelectText. In particular, zero is not a valid character position when selecting text, although it is meaningful in describing the selection.

For more information, see Position in the *PowerScript Reference*.

For an example of selecting words one by one for the purposes of spell checking, see the SelectTextWord function in the *PowerScript Reference*.

Tab order, focus, and the selection

Tab order For a window or user object, you include the RichTextEdit control in the tab order of controls. However, after the user tabs to the RichTextEdit control, pressing the TAB key inserts tabs into the text. The user cannot tab out to other controls. Keep this in mind when you design the tab order for a window.

Focus and the selection When the user tabs to the RichTextEdit control, the control gets focus and the current insertion point or selection is maintained. If the user clicks the RichTextEdit control to set focus, the insertion point moves to the place the user clicks.

LoseFocus event When the user clicks on a RichTextEdit toolbar, a LoseFocus event occurs. However, the RichTextEdit control still has focus. You can check whether the control has lost focus with the GetFocus function.

Preview and printing

The user can preview the layout and print the contents of the RichTextEdit control. In print preview mode, users see a view of the document reduced so that it fits inside the control. However, you must set the print margins and page size before you display the control in print preview mode.

There are two ways to enter print preview mode:

- The user can press CTRL+F2 to switch between editing and print preview mode
- You can call the Preview function in a script:

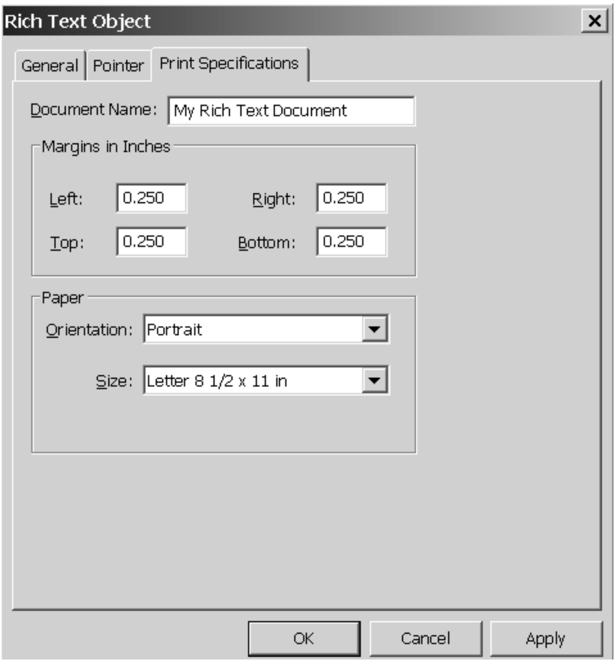
```
rte_1.Preview(TRUE)
```

Users can page through the control contents in print preview mode by using the up arrow and down arrow keys or the Page Up and Page Down keys.

Adjusting the print margins

If you set page margins at design time, or enable headers and footers for a rich text control, application users can adjust the margins of the control at runtime. Users can do this by opening the property sheet for the RichTextEdit control to the Print Specifications tab and modifying the left, right, top, or bottom margins, or by triggering an event that changes the margins in PowerScript code. Adjusting the margins in the Rich Text Object dialog box also affects the display of the RichTextEdit control content in print preview mode.

If you do not set page margins at design time or leave them at 0, any changes the user makes to the margins at runtime are visible in print preview mode only.



Document Name: My Rich Text Document

Margins in Inches

Left: 0.250 Right: 0.250

Top: 0.250 Bottom: 0.250

Paper

Orientation: Portrait

Size: Letter 8 1/2 x 11 in

Setting page size and orientation
You cannot set the default page size and page orientation at design time. However, users can set these properties at runtime from the Print Specifications tab of the Rich Text Object dialog box. This dialog box is available from the standard view only. You must also enable the pop-up menu on a RichTextEdit control to enable application users to display this dialog box.

Printing

If the RichTextEdit is using DataWindow object data, you can limit the number of rows printed by setting the Print.Page.Range property for the DataWindow control. Its value is a string that lists the page numbers that you want to print. A dash indicates a range.

Example of a page range Suppose your RichTextEdit control has a data source in the control dw_source. Your rich text document is three pages and you want to print the information for rows 2 and 5. You can set the page range property before you print:

```
dw_source.Object.DataWindow.Print.Page.Range = &
   "4-6,13-15"
```

You can also filter or discard rows so that they are not printed.

For more information, see the SetFilter, Filter, RowsMove, and RowsDiscard functions in the *PowerScript Reference* and the Print DataWindow object property in the *DataWindow Reference*.

Setting the page number

To print page numbers, you can use an input field in the header or footer. Although the page number field can be a string or numeric expression, when you insert the page number field in a header or footer, you must use it as a string expression only. For example, if you use `page()*2` for an input field in the header or footer, the control or report is likely to display an incorrect result for the value of the numeric expression. However, the following string expression should display the correct page number and page count:

```
'Page ' + page() + ' of ' + pageCount())
```

Inserting footer text programmatically

This sample code sets the insertion point in the footer and inserts two blank lines, text, and two input fields:

```
rte_1.SelectText(1, 1, 0, 0, Footer!)
rte_1.ReplaceText("~r~n~r~nRow ")
rte_1.InputFieldInsert("row")
rte_1.ReplaceText(" Page ")
rte_1.InputFieldInsert("page")
rte_1.SetAlignment(Center!)
```

Rich text and the end user

All the editing tools described throughout this chapter and in the chapter on working with rich text in the *PowerBuilder User's Guide* can be made available to your users.

What users can do

Users can:

- Use the toolbars for text formatting
- Use the pop-up menu, which includes using the clipboard and opening other rich text and ASCII files
- Edit the contents of input fields
- Turn the editing tools on and off

What you can make available to users in your code

You can program an application to allow users to:

- Insert and delete input fields

- Insert pictures
- Switch to header and footer editing
- Preview the document for printing

If a RichTextEdit control shares data with a DataWindow object or DataStore, you can program:

- Scrolling from row to row (you do not need to program page-to-page scrolling, although you can)
- Updating the database with changes made in input fields

The best way for you to prepare rich text for use in your application is to become a user yourself and edit the text in an application designed for the purpose. During execution, all the tools for text preparation are available.

What the user sees

The default view is the body text. You can also show header and footer text and a print preview.

Header and footer text For either a RichText DataWindow object or the RichTextEdit control, you can call the ShowHeadFoot function in a menu or button script. To display the header editing panel, you can call:

```
dw_1.ShowHeadFoot(TRUE)
```

To display the footer editing panel, you must call:

```
dw_1.ShowHeadFoot(TRUE, FALSE)
```

Inserting the current page number in a footer
The following script inserts the current page number in the footer, then returns the focus to the body of the document in the rich text control:

```
rte_1.ShowHeadFoot(true,false)
rte_1.SetAlignment ( Center! )
rte_1.InputFieldInsert("PAGENO")
rte_1.ShowHeadFoot(false,false)
```

In the overloaded function ShowHeadFoot, the second argument defaults to TRUE if a value is not provided. Call the function again to return to normal view.

```
dw_1.ShowHeadFoot(FALSE)
```

The document as it would be printed The user can press CTRL+F2 to switch print preview mode on and off. You can also control print preview mode programmatically.

For a RichTextEdit control, call the Preview function:

```
rte_1.Preview(TRUE)
```

For a RichText DataWindow object, set the Preview property:

```
dw_1.Object.DataWindow.Print.Preview = TRUE
```

Text elements and formatting

The user can specify formatting for:

- Selected text
- Paragraphs
- Pictures
- The whole rich text document

❖ **To display the property sheet for an object, the user can:**

1. Select the object. For example:
   - Drag or use editing keys to select text
   - Click on a picture
   - Set an insertion point (nothing selected) for the rich text document
2. Right-click in the workspace and select Properties from the pop-up menu.

❖ **To make settings for the paragraphs in the selection:**

- Double-click on the ruler bar
  or
  Type Ctrl+Shift+S.

Modifying input fields

Unless you have made the rich text object display only, the user can modify the values of input fields.

❖ **To modify the value of an input field:**

1. Click the input field to select it.
2. Right-click in the workspace and choose Properties from the pop-up menu.

   The Input Field Object property sheet displays.
3. On the Input Field page, edit the Data Value text box.

Text formatting for input fields There are several ways to select the input field and apply text formatting. When the input field is selected, the Font page of the property sheet and the toolbar affect the text. When the input field is part of a text selection, changes affect all the text, including the input field.

The user cannot apply formatting to individual characters or words within the field. When the user selects the input field, the entire field is selected.

Inserting and deleting input fields You write scripts that let the user insert and delete input fields. The user can also copy and paste existing input fields. All copies of an input field display the same data.

Formatting keys and toolbars

When the toolbar is visible, users can use its buttons to format text, or they can use designated keystrokes to format text in the RichTextEdit control.

For a list of keystrokes for formatting rich text, see the chapter on working with rich text in the *PowerBuilder User's Guide*.

CHAPTER 17

Piping Data Between Data Sources

About this chapter

This chapter tells you how you can use a Pipeline object in your application to pipe data from one or more source tables to a new or existing destination table.

Contents

Sample applications

This chapter uses a simple order entry application to illustrate the use of a data pipeline. To see working examples using data pipelines, look at the examples in the Data Pipeline category in the Code Examples sample application.

For information on how to use the sample applications, see Chapter 1, “Using Sample Applications.”

About data pipelines

PowerBuilder provides a feature called the **data pipeline** that you can use to migrate data between database tables. This feature makes it possible to copy rows from one or more source tables to a new or existing destination table—either within a database, or across databases, or even across DBMSs.

Two ways to use data pipelines

You can take advantage of data pipelines in two different ways:

- **As a utility service for developers**

  While working in the PowerBuilder development environment, you might occasionally want to migrate data for logistical reasons (such as to create a small test table from a large production table). In this case, you can use the Data Pipeline painter interactively to perform the migration immediately.

  For more information on using the Data Pipeline painter this way, see the *PowerBuilder User's Guide*.

- **To implement data migration capabilities in an application**

  If you are building an application whose requirements call for migrating data between tables, you can design an appropriate data pipeline in the Data Pipeline painter, save it, and then enable users to execute it from within the application.

  This technique can be useful in many different situations, such as: when you want the application to download local copies of tables from a database server to a remote user, or when you want it to roll up data from individual transaction tables to a master transaction table.

Walking through the basic steps

If you determine that you need to use a data pipeline in your application, you must determine what steps this involves. At the most general level, there are five basic steps that you typically have to perform.

❖ **To pipe data in an application:**

1. Build the objects you need.
2. Perform some initial housekeeping.
3. Start the pipeline.
4. Handle row errors.
5. Perform some final housekeeping.

The remainder of this chapter gives you the details of each step.

Building the objects you need

To implement data piping in an application, you need to build a few different objects:

- A Pipeline object
- A supporting user object
- A window

Building a Pipeline object

You must build a Pipeline object to specify the data definition and access aspects of the pipeline that you want your application to execute. Use the Data Pipeline painter in PowerBuilder to create this object and define the characteristics you want it to have.

Characteristics to define

Among the characteristics you can define in the Data Pipeline painter are:

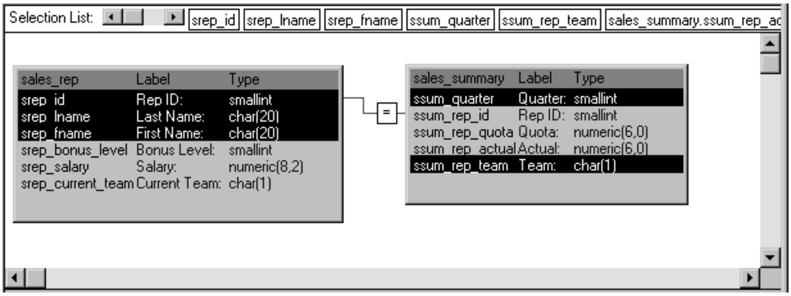
- *The source tables* to access and the data to retrieve from them (you can also access database stored procedures as the data source)
- *The destination table* to which you want that data piped
- *The piping operation* to perform (create, replace, refresh, append, or update)
- *The frequency of commits* during the piping operation (after every *n* rows are piped, or after all rows are piped, or not at all—if you plan to code your own commit logic)
- *The number of errors* to allow before the piping operation is terminated
- *Whether or not to pipe extended attributes* to the destination database (from the PowerBuilder repository in the source database)

For full details on using the Data Pipeline painter to build your Pipeline object, see the *PowerBuilder User's Guide*.

Example

Here is an example of how you would use the Data Pipeline painter to define a Pipeline object named pipe_sales_extract1 (one of two Pipeline objects employed by the w_sales_extract window in a sample order entry application).

The source data to pipe This Pipeline object joins two tables (Sales_rep and Sales_summary) from the company's sales database to provide the source data to be piped. It retrieves just the rows from a particular quarter of the year (which the application must specify by supplying a value for the retrieval argument named *quarter*):

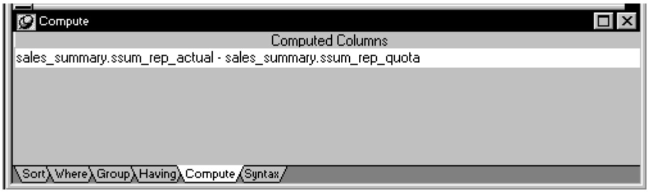


Selection List: srep_id srep_lname srep_fname ssum_quarter ssum_rep_team sales_summary.ssum_rep_ac

sales_rep	Label	Type
srep_id	Rep ID:	smallint
srep_lname	Last Name:	char(20)
srep_fname	First Name:	char(20)
srep_bonus_level	Bonus Level:	smallint
srep_salary	Salary:	numeric(8,2)
srep_current_team	Current Team:	char(1)

sales_summary	Label	Type
ssum_quarter	Quarter:	smallint
ssum_rep_id	Rep ID:	smallint
ssum_rep_quota	Quota:	numeric(6,0)
ssum_rep_actual	Actual:	numeric(6,0)
ssum_rep_team	Team:	char(1)

Notice that this Pipeline object also indicates specific columns to be piped from each source table (srep_id, srep_lname, and srep_fname from the Sales_rep table, as well as ssum_quarter and ssum_rep_team from the Sales_summary table). In addition, it defines a computed column to be calculated and piped. This computed column subtracts the ssum_rep_quota column of the Sales_summary table from the ssum_rep_actual column:



Compute

Computed Columns

sales_summary.ssum_rep_actual - sales_summary.ssum_rep_quota

How to pipe the data The details of how pipe_sales_extract1 is to pipe its source data are specified here:

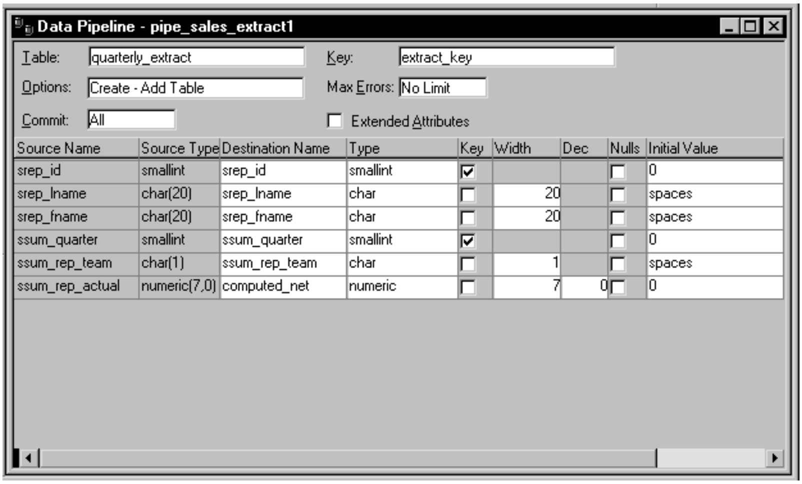


Table: quarterly_extract

Key: extract_key

Options: Create - Add Table

Max Errors: No Limit

Commit: All

□ Extended Attributes

Source Name	Source Type	Destination Name	Type	Key	Width	Dec	Nulls	Initial Value
srep_id	smallint	srep_id	smallint	☑			□	0
srep_lname	char(20)	srep_lname	char	□	20		□	spaces
srep_fname	char(20)	srep_fname	char	□	20		□	spaces
ssum_quarter	smallint	ssum_quarter	smallint	☑			□	0
ssum_rep_team	char(1)	ssum_rep_team	char	□	1		□	spaces
ssum_rep_actual	numeric(7,0)	computed_net	numeric	□	7	0	□	0

Notice that this Pipeline object is defined to create a new destination table named Quarterly_extract. A little later you will learn how the application specifies the destination database in which to put this table (as well as how it specifies the source database in which to look for the source tables).

Also notice that:

- *A commit* will be performed only after all appropriate rows have been piped (which means that if the pipeline's execution is terminated early, all changes to the Quarterly_extract table will be rolled back).
- *No error limit* is to be imposed by the application, so any number of rows can be in error without causing the pipeline's execution to terminate early.
- *No extended attributes* are to be piped to the destination database.
- *The primary key* of the Quarterly_extract table is to consist of the srep_id column and the ssum_quarter column.
- *The computed column* that the application is to create in the Quarterly_extract table is to be named computed_net.

Building a supporting user object

So far you have seen how your Pipeline object defines the details of the data and access for a pipeline, but a Pipeline object does not include the logistical supports—properties, events, and functions—that an application requires to handle pipeline execution and control.

About the Pipeline system object

To provide these logistical supports, you must build an appropriate user object inherited from the PowerBuilder **Pipeline system object**. Table 17-1 shows some of the system object's properties, events, and functions that enable your application to manage a Pipeline object at runtime.

Table 17-1: Pipeline system object properties, events, and functions

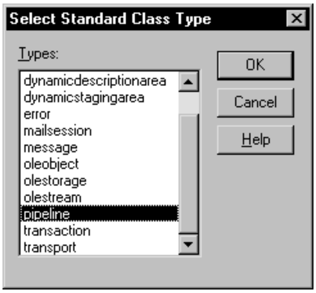
Properties	Events	Functions
DataObject	PipeStart	Start
RowsRead	PipeMeter	Repair
RowsWritten	PipeEnd	Cancel
RowsInError		
Syntax		

A little later in this chapter you will learn how to use most of these properties, events, and functions in your application.

❖ **To build the supporting user object for a pipeline:**

1 Select Standard Class from the PB Object tab of the New dialog box.

The Select Standard Class Type dialog box displays, prompting you to specify the name of the PowerBuilder system object (class) from which you want to inherit your new user object:



2 Select pipeline and click OK.

3. Make any changes you want to the user object (although none are required). This might involve coding events, functions, or variables for use in your application.

   To learn about one particularly useful specialization you can make to your user object, see "Monitoring pipeline progress" on page 300.

   Planning ahead for reuse
   As you work on your user object, keep in mind that it can be reused in the future to support any other pipelines you want to execute. It is not automatically tied in any way to a particular Pipeline object you have built in the Data Pipeline painter.

   To take advantage of this flexibility, make sure that the events, functions, and variables you code in the user object are generic enough to accommodate any Pipeline object.

4. Save the user object.

For more information on working with the User Object painter, see the *PowerBuilder User's Guide*.

Building a window

One other object you need when piping data in your application is a window. You use this window to provide a user interface to the pipeline, enabling people to interact with it in one or more ways. These include:

- *Starting* the pipeline's execution
- *Displaying and repairing* any errors that occur
- *Canceling* the pipeline's execution if necessary

Required features for your window

When you build your window, you must include a DataWindow control that the pipeline itself can use to display error rows (that is, rows it cannot pipe to the destination table for some reason). You do not have to associate a DataWindow object with this DataWindow control—the pipeline provides one of its own at runtime.

To learn how to work with this DataWindow control in your application, see "Starting the pipeline" on page 298 and "Handling row errors" on page 304.

Optional features for your window

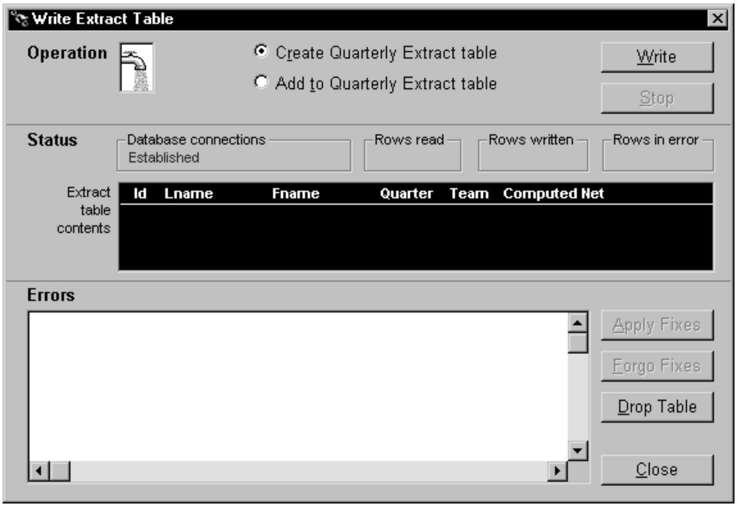
Other than including the required DataWindow control, you can design the window as you like. You will typically want to include various other controls, such as:

- *CommandButton or PictureButton controls* to let the user initiate actions (such as starting, repairing, or canceling the pipeline)
- *StaticText controls* to display pipeline status information
- *Additional DataWindow controls* to display the contents of the source and/or destination tables

If you need assistance with building a window, see the *PowerBuilder User's Guide*.

Example

The following window handles the user-interface aspect of the data piping in the order entry application. This window is named w_sales_extract:



Operation

● Create Quarterly Extract table
○ Add to Quarterly Extract table

Status

Database connections: Established

Id	Lname	Fname	Quarter	Team	Computed Net

Several of the controls in this window are used to implement particular pipeline-related capabilities. Table 17-2 provides more information about them.

Table 17-2: Window controls to implement pipeline capabilities

Control type	Control name	Purpose
RadioButton	rb_create	Selects pipe_sales_extract1 as the Pipeline object to execute
	rb_insert	Selects pipe_sales_extract2 as the Pipeline object to execute
CommandButton	cb_write	Starts execution of the selected pipeline
	cb_stop	Cancels pipeline execution or applying of row repairs
	cb_applyfixes	Applies row repairs made by the user (in the dw_pipe_errors DataWindow control) to the destination table
	cb_forgofixes	Clears all error rows from the dw_pipe_errors DataWindow control (for use when the user decides not to make repairs)
DataWindow	dw_review_extract	Displays the current contents of the destination table (Quarterly_extract)
	dw_pipe_errors	(*Required*) Used by the pipeline itself to automatically display the PowerBuilder pipeline-error DataWindow (which lists rows that cannot be piped due to some error)
StaticText	st_status_read	Displays the count of rows that the pipeline reads from the source tables
	st_status_written	Displays the count of rows that the pipeline writes to the destination table or places in dw_pipe_errors
	st_status_error	Displays the count of rows that the pipeline places in dw_pipe_errors (because they are in error)

Performing some initial housekeeping

Now that you have the basic objects you need, you are ready to start writing code to make your pipeline work in the application. To begin, you must take care of some setup chores that will prepare the application to handle pipeline execution.

❖ **To get the application ready for pipeline execution:**

1. Connect to the source and destination databases for the pipeline.

   To do this, write the usual connection code in an appropriate script. Just make sure you use one Transaction object when connecting to the source database and a different Transaction object when connecting to the destination database (even if it is the same database).

   For details on connecting to a database, see Chapter 12, "Using Transaction Objects."

2. Create an instance of your supporting user object (so that the application can use its properties, events, and functions).

   To do this, first declare a variable whose type is that user object. Then, in an appropriate script, code the CREATE statement to create an instance of the user object and assign it to that variable.

3. Specify the particular Pipeline object you want to use.

   To do this, code an Assignment statement in an appropriate script; assign a string containing the name of the desired Pipeline object to the DataObject property of your user-object instance.

For more information on coding the CREATE and Assignment statements, see the *PowerScript Reference*.

Example

The following sample code takes care of these pipeline setup chores in the order entry application.

Connecting to the source and destination database In this case, the company's sales database (ABNCSALE.DB) is used as both the source and the destination database. To establish the necessary connections to the sales database, write code in a user event named uevent_pipe_setup (which is posted from the Open event of the w_sales_extract window).

The following code establishes the *source database connection*:

```
// Create a new instance of the Transaction object
// and store it in itrans_source (a variable
// declared earlier of type transaction).
itrans_source = CREATE transaction

// Next, assign values to the properties of the
// itrans_source Transaction object.
...

// Now connect to the source database.
CONNECT USING itrans_source;
```

The following code establishes the *destination database connection*:

```
// Create a new instance of the Transaction object
// and store it in itrans_destination (a variable
// declared earlier of type transaction).

itrans_destination = CREATE transaction

// Next, assign values to the properties of the
// itrans_destination Transaction object.
...
// Now connect to the destination database.

CONNECT USING itrans_destination;
```

Setting USERID for native drivers

When you execute a pipeline in the Pipeline painter, if you are using a native driver, PowerBuilder automatically qualifies table names with the owner of the table. When you execute a pipeline in an application, if you are using a native driver, you must set the USERID property in the Transaction object so that the table name is properly qualified.

Failing to set the USERID property in the Transaction object for the destination database causes pipeline execution errors. If the source database uses a native driver, extended attributes are not piped if USERID is not set.

Creating an instance of the user object Earlier you learned how to develop a supporting user object named u_sales_pipe_logistics. To use u_sales_pipe_logistics in the application, first declare a variable of its type:

```
// This is an instance variable for the
// w_sales_extract window.

u_sales_pipe_logistics iuo_pipe_logistics
```

Then write code in the uevent_pipe_setup user event to create an instance of u_sales_pipe_logistics and store this instance in the variable *iuo_pipe_logistics*:

```
iuo_pipe_logistics = CREATE u_sales_pipe_logistics
```

Specifying the Pipeline object to use The application uses one of two different Pipeline objects, depending on the kind of piping operation the user wants to perform:

- *pipe_sales_extract1* (which you saw in detail earlier) creates a new Quarterly_extract table (and assumes that this table *does not* currently exist)

- *pipe_sales_extract2* inserts rows into the Quarterly_extract table (and assumes that this table *does* currently exist)

To choose a Pipeline object and prepare to use it, write the following code in the Clicked event of the cb_write CommandButton (which users click when they want to start piping):

```
// Look at which radio button is checked in the
// w_sales_extract window. Then assign the matching
// Pipeline object to iuo_pipe_logistics.

IF rb_create.checked = true THEN
 iuo_pipe_logistics.dataobject =
      "pipe_sales_extract1"
ELSE
 iuo_pipe_logistics.dataobject =
               "pipe_sales_extract2"
END IF
```

This code appears at the beginning of the script, before the code that starts the chosen pipeline.

Deploying Pipeline objects for an application
Because an application must always reference its Pipeline objects *dynamically* at runtime (through string variables), you must package these objects in one or more dynamic libraries when deploying the application. You cannot include Pipeline objects in an executable (*EXE*) file.

For more information on deployment, see Part 9, “Deployment Techniques.”

Starting the pipeline

With the setup chores taken care of, you can now start the execution of your pipeline.

❖ **To start pipeline execution:**

1 Code the Start function in an appropriate script. In this function, you specify:

- The Transaction object for the source database
- The Transaction object for the destination database

- The DataWindow control in which you want the Start function to display any error rows

  The Start function automatically associates the PowerBuilder pipeline-error DataWindow object with your DataWindow control when needed.

- Values for retrieval arguments you have defined in the Pipeline object

  If you omit these values, the Start function prompts the user for them automatically at runtime.

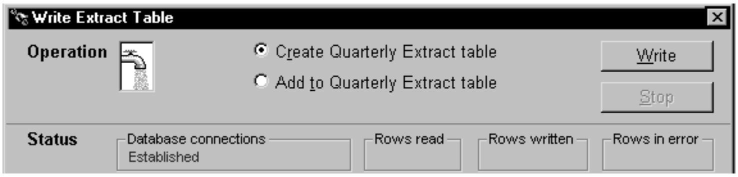
2 Test the result of the Start function.

For more information on coding the Start function, see the *PowerScript Reference*.

Example

The following sample code starts pipeline execution in the order entry application.

Calling the Start function When users want to start their selected pipeline, they click the cb_write CommandButton in the w_sales_extract window:


Write Extract Table

Operation

● Create Quarterly Extract table

○ Add to Quarterly Extract table

Status

Database connections: Established

Rows read

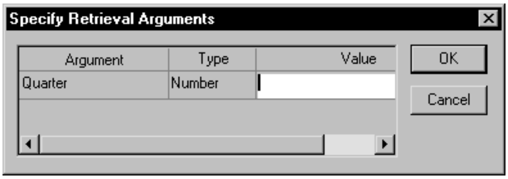
Rows written

Rows in error

This executes the Clicked event of cb_write, which contains the Start function:

```
// Now start piping.
integer li_start_result
li_start_result = iuo_pipe_logistics.Start &
   (itrans_source,itrans_destination,dw_pipe_errors)
```

Notice that the user did not supply a value for the pipeline's retrieval argument (*quarter*). As a consequence, the Start function prompts the user for it:


Specify Retrieval Arguments

Argument	Type	Value
Quarter	Number	

Testing the result The next few lines of code in the Clicked event of cb_write check the Start function's return value. This lets the application know whether it succeeded or not (and if not, what went wrong):

```
CHOOSE CASE li_start_result

   CASE -3
   Beep (1)
   MessageBox("Piping Error", &
      "Quarterly_Extract table already exists ...
   RETURN

   CASE -4
   Beep (1)
   MessageBox("Piping Error", &
      "Quarterly_Extract table does not exist ...
   RETURN
   ...
END CHOOSE
```

Monitoring pipeline progress

Testing the Start function's return value is not the only way to monitor the status of pipeline execution. Another technique you can use is to retrieve statistics that your supporting user object keeps concerning the number of rows processed. They provide a live count of:

- *The rows read* by the pipeline from the source tables
- *The rows written* by the pipeline to the destination table or to the error DataWindow control
- *The rows in error* that the pipeline has written to the error DataWindow control (but not to the destination table)

By retrieving these statistics from the supporting user object, you can dynamically display them in the window and enable users to watch the pipeline's progress.

❖ **To display pipeline row statistics:**

1 Open your supporting user object in the User Object painter.

The User Object painter workspace displays, enabling you to edit your user object.

2 Declare three instance variables of type StaticText:

```
statictext ist_status_read, ist_status_written, &
      ist_status_error
```

You will use these instance variables later to hold three StaticText controls from your window. This will enable the user object to manipulate those controls directly and make them dynamically display the various pipeline row statistics.

3 In the user object's PipeMeter event script, code statements to assign the values of properties inherited from the pipeline system object to the Text property of your three StaticText instance variables.

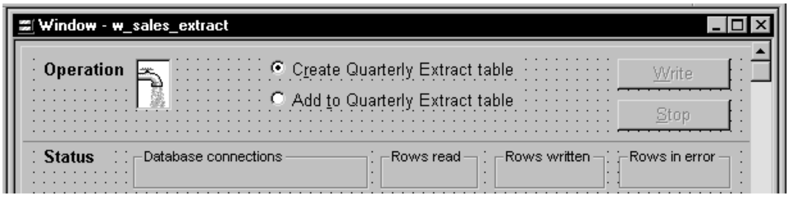
```
ist_status_read.text    = string(RowsRead)
ist_status_written.text = string(RowsWritten)
ist_status_error.text   = string(RowsInError)
```

4 Save your changes to the user object, then close the User Object painter.

5 Open your window in the Window painter.

6 Insert three StaticText controls in the window:

One to display the RowsRead value
One to display the RowsWritten value
One to display the RowsInError value



7 Edit the window's Open event script (or some other script that executes right after the window opens).

In it, code statements to assign the three StaticText controls (which you just inserted in the window) to the three corresponding StaticText instance variables you declared earlier in the user object. This enables the user object to manipulate these controls directly.

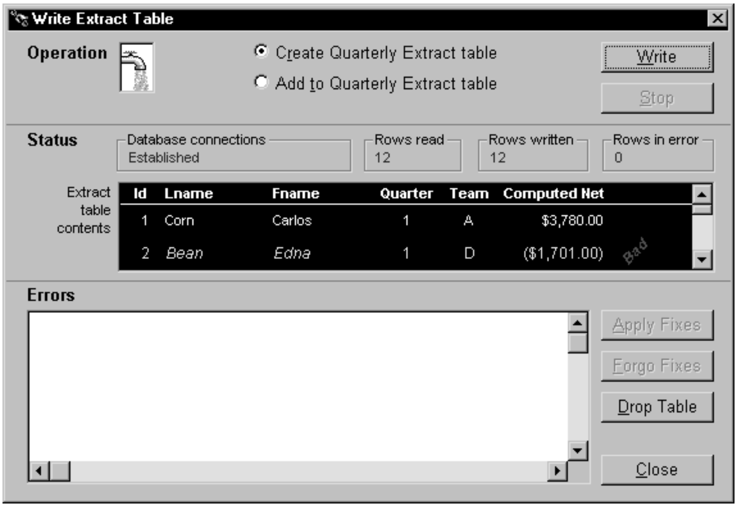
In the sample order entry application, this logic is in a user event named uevent_pipe_setup (which is posted from the Open event of the w_sales_extract window):

```
iuo_pipe_logistics.ist_status_read =
               st_status_read
```

```
iuo_pipe_logistics.ist_status_written = &
      st_status_written
iuo_pipe_logistics.ist_status_error = &
      st_status_error
```

8 Save your changes to the window. Then close the Window painter.

When you start a pipeline in the w_sales_extract window of the order entry application, the user object's PipeMeter event triggers and executes its code to display pipeline row statistics in the three StaticText controls:



● Create Quarterly Extract table
○ Add to Quarterly Extract table

Database connections: Established
Rows read: 12
Rows written: 12
Rows in error: 0

Id	Lname	Fname	Quarter	Team	Computed Net
1	Corn	Carlos	1	A	$3,780.00
2	*Bean*	*Edna*	1	D	($1,701.00)

Canceling pipeline execution

In many cases you will want to provide users (or the application itself) with the ability to stop execution of a pipeline while it is in progress. For instance, you may want to give users a way out if they start the pipeline by mistake or if execution is taking longer than desired (maybe because many rows are involved).

❖ **To cancel pipeline execution:**

1 Code the Cancel function in an appropriate script

Make sure that either the user or your application can execute this function (if appropriate) once the pipeline has started. When Cancel is executed, it stops the piping of any more rows after that moment.

Rows that have already been piped up to that moment may or may not be committed to the destination table, depending on the Commit property you specified when building your Pipeline object in the Data Pipeline painter. You will learn more about committing in the next section.

2 Test the result of the Cancel function

For more information on coding the Cancel function, see the *PowerScript Reference*.

Example

The following example uses a command button to let users cancel pipeline execution in the order entry application.

Providing a CommandButton When creating the w_sales_extract window, include a CommandButton control named cb_stop. Then write code in a few of the application's scripts to enable this CommandButton when pipeline execution starts and to disable it when the piping is done.

Calling the Cancel function Next write a script for the Clicked event of cb_stop. This script calls the Cancel function and tests whether or not it worked properly:

```
IF iuo_pipe_logistics.Cancel() = 1 THEN
   Beep (1)
   MessageBox("Operation Status", &
   "Piping stopped (by your request).")
ELSE
   Beep (1)
   MessageBox("Operation Status", &
   "Error when trying to stop piping.", &
   Exclamation!)
END IF
```

Together, these features let a user of the application click the cb_stop CommandButton to cancel a pipeline that is currently executing.

Committing updates to the database

When a Pipeline object executes, it commits updates to the destination table according to your specifications in the Data Pipeline painter. You do not need to write any COMMIT statements in your application's scripts (unless you specified the value *None* for the Pipeline object's Commit property).

Example

For instance, both of the Pipeline objects in the order entry application (pipe_sales_extract1 and pipe_sales_extract2) are defined in the Data Pipeline painter to commit *all* rows. As a result, the Start function (or the Repair function) will pipe every appropriate row and then issue a commit.

You might want instead to define a Pipeline object that *periodically* issues commits as rows are being piped, such as after every 10 or 100 rows.

If the Cancel function is called

A related topic is what happens with committing if your application calls the Cancel function to stop a pipeline that is currently executing. In this case too, the Commit property in the Data Pipeline painter determines what to do, as shown in Table 17-3.

Table 17-3: Commit property values

If your Commit value is	Then Cancel does this
All	Rolls back every row that was piped by the current Start function (or Repair function)
A particular number of rows (such as *1, 10*, or *100*)	Commits every row that was piped up to the moment of cancellation

This is the same commit/rollback behavior that occurs when a pipeline reaches its Max Errors limit (which is also specified in the Data Pipeline painter).

For more information on controlling commits and rollbacks for a Pipeline object, see the *PowerBuilder User's Guide*.

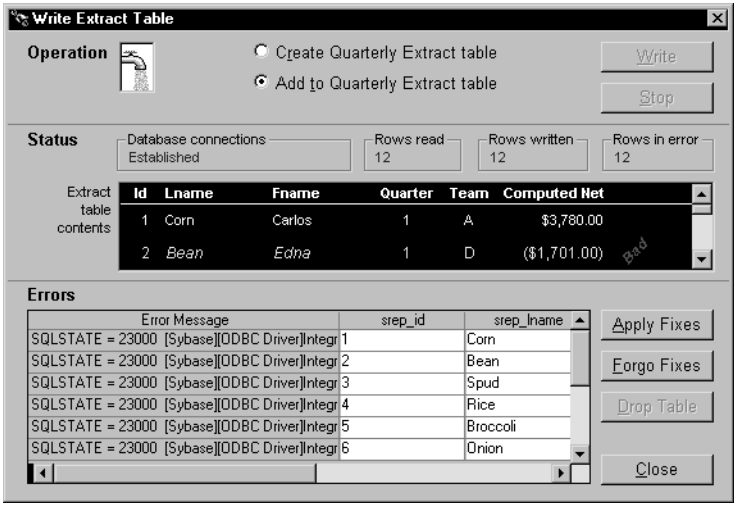
Handling row errors

When a pipeline executes, it may be unable to write particular rows to the destination table. For instance, this could happen with a row that has the same primary key as a row already in the destination table.

Using the pipeline-error DataWindow

To help you handle such error rows, the pipeline places them in the DataWindow control you painted in your window and specified in the Start function. It does this by automatically associating its own special DataWindow object (the PowerBuilder pipeline-error DataWindow) with your DataWindow control.

Consider what happens in the order entry application. When a pipeline executes in the w_sales_extract window, the Start function places all error rows in the dw_pipe_errors DataWindow control. It includes an error message column to identify the problem with each row:



Write Extract Table

Operation

○ Create Quarterly Extract table
● Add to Quarterly Extract table

Status

Database connections: Established
Rows read: 12
Rows written: 12
Rows in error: 12

Extract table contents

Id	Lname	Fname	Quarter	Team	Computed Net
1	Corn	Carlos	1	A	$3,780.00
2	Bean	Edna	1	D	($1,701.00) Bad

Errors

Error Message	srep_id	srep_lname
SQLSTATE = 23000 [Sybase][ODBC Driver]Integr	1	Corn
SQLSTATE = 23000 [Sybase][ODBC Driver]Integr	2	Bean
SQLSTATE = 23000 [Sybase][ODBC Driver]Integr	3	Spud
SQLSTATE = 23000 [Sybase][ODBC Driver]Integr	4	Rice
SQLSTATE = 23000 [Sybase][ODBC Driver]Integr	5	Broccoli
SQLSTATE = 23000 [Sybase][ODBC Driver]Integr	6	Onion

Making the error messages shorter

If the pipeline's destination Transaction object points to an ODBC data source, you can set its DBParm MsgTerse parameter to make the error messages in the DataWindow shorter. Specifically, if you type:

```
MsgTerse = 'Yes'
```

then the SQLSTATE error number does not display.

For more information on the MsgTerse DBParm, see the online Help.

Deciding what to do with error rows

Once there are error rows in your DataWindow control, you need to decide what to do with them. Your alternatives include:

- *Repairing* some or all of those rows
- *Abandoning* some or all of those rows

Repairing error rows

In many situations it is appropriate to try fixing error rows so that they can be applied to the destination table. Making these fixes typically involves modifying one or more of their column values so that the destination table will accept them. You can do this in a couple of different ways:

- *By letting the user edit* one or more of the rows in the error DataWindow control (the easy way for you, because it does not require any coding work)
- *By executing script code* in your application that edits one or more of the rows in the error DataWindow control for the user

In either case, the next step is to apply the modified rows from this DataWindow control to the destination table.

❖ **To apply row repairs to the destination table:**

1 Code the Repair function in an appropriate script. In this function, specify the Transaction object for the destination database.

2 Test the result of the Repair function.

For more information on coding the Repair function, see the *PowerScript Reference*.

Example

In the following example, users can edit the contents of the dw_pipe_errors DataWindow control to fix error rows that appear. They can then apply those modified rows to the destination table.

Providing a CommandButton When painting the w_sales_extract window, include a CommandButton control named cb_applyfixes. Then write code in a few of the application's scripts to enable this CommandButton when dw_pipe_errors contains error rows and to disable it when no error rows appear.

Calling the Repair function Next write a script for the Clicked event of cb_applyfixes. This script calls the Repair function and tests whether or not it worked properly:

```
IF iuo_pipe_logistics.Repair(itrans_destination) &
   <> 1 THEN
   Beep (1)
   MessageBox("Operation Status", "Error when &
   trying to apply fixes.", Exclamation!)
END IF
```

Together, these features let a user of the application click the cb_applyfixes CommandButton to try updating the destination table with one or more corrected rows from dw_pipe_errors.

Canceling row repairs

Earlier in this chapter you learned how to let users (or the application itself) stop writing rows to the destination table during the initial execution of a pipeline. If appropriate, you can use the same technique while row repairs are being applied.

For details, see “Canceling pipeline execution” on page 302.

Committing row repairs

The Repair function commits (or rolls back) database updates in the same way the Start function does.

For details, see “Committing updates to the database” on page 303.

Handling rows that still are not repaired

Sometimes after the Repair function has executed, there may still be error rows left in the error DataWindow control. This may be because these rows:

- Were modified by the user or application but still have errors
- Were not modified by the user or application
- Were never written to the destination table because the Cancel function was called (or were rolled back from the destination table following the cancellation)

At this point, the user or application can try again to modify these rows and then apply them to the destination table with the Repair function. There is also the alternative of abandoning one or more of these rows. You will learn about that technique next.

Abandoning error rows

In some cases, you may want to enable users or your application to completely discard one or more error rows from the error DataWindow control. This can be useful for dealing with error rows that it is not desirable to repair.

Table 17-4 shows some techniques you can use for abandoning such error rows.

Table 17-4: Abandoning error rows

If you want to abandon	Use
All error rows in the error DataWindow control	The Reset function
One or more particular error rows in the error DataWindow control	The RowsDiscard function

For more information on coding these functions, see the *PowerScript Reference*.

Example

In the following example, users can choose to abandon all error rows in the dw_pipe_errors DataWindow control.

Providing a CommandButton When painting the w_sales_extract window, include a CommandButton control named cb_forgofixes. Write code in a few of the application's scripts to enable this CommandButton when dw_pipe_errors contains error rows and to disable it when no error rows appear.

Calling the Reset function Next write a script for the Clicked event of cb_forgofixes. This script calls the Reset function:

```
dw_pipe_errors.Reset()
```

Together, these features let a user of the application click the cb_forgofixes CommandButton to discard all error rows from dw_pipe_errors.

Performing some final housekeeping

When your application has finished processing pipelines, you need to make sure it takes care of a few cleanup chores. These chores basically involve releasing the resources you obtained at the beginning to support pipeline execution.

Garbage collection
You should avoid using the DESTROY statement to clean up resources unless you are sure that the objects you are destroying are not used elsewhere. PowerBuilder's garbage collection mechanism automatically removes unreferenced objects. For more information, see "Garbage collection and memory management" on page 45.

❖ **To clean up when you have finished using pipelines:**

1 Destroy the instance that you created of your supporting user object.

To do this, code the DESTROY statement in an appropriate script and specify the name of the variable that contains that user-object instance.

2 Disconnect from the pipeline's source and destination databases.

To do this, code two DISCONNECT statements in an appropriate script. In one, specify the name of the variable that contains your source transaction-object instance. In the other, specify the name of the variable that contains your destination transaction-object instance.

Then test the result of each DISCONNECT statement.

3 Destroy your source transaction-object instance and your destination transaction-object instance.

To do this, code two DESTROY statements in an appropriate script. In one, specify the name of the variable that contains your source transaction-object instance. In the other, specify the name of the variable that contains your destination transaction-object instance.

For more information on coding the DESTROY and DISCONNECT statements, see the *PowerScript Reference*.

Example

The following code in the Close event of the w_sales_extract window takes care of these cleanup chores.

Destroying the user-object instance At the beginning of the Close event script, code the following statement to destroy the instance of the user object u_sales_pipe_logistics (which is stored in the *iuo_pipe_logistics* variable):

```
DESTROY iuo_pipe_logistics
```

Disconnecting from the source database Next, code these statements to disconnect from the source database, test the result of the disconnection, and destroy the source transaction-object instance (which is stored in the *itrans_source* variable):

```
DISCONNECT USING itrans_source;

   // Check result of DISCONNECT statement.
IF itrans_source.SQLCode = -1 THEN
   Beep (1)
   MessageBox("Database Connection Error", &
   "Problem when disconnecting from the source " &
   + "database. Please call technical support. " &
   + "~n~r~n~rDetails follow: " + &
   String(itrans_source.SQLDBCode) + " " + &
   itrans_source.SQLErrText, Exclamation!)
END IF

DESTROY itrans_source
```

Disconnecting from the destination database Finally, code these statements to disconnect from the destination database, test the result of the disconnection, and destroy their destination transaction-object instance (which is stored in the *itrans_destination* variable):

```
DISCONNECT USING itrans_destination;

   // Check result of DISCONNECT statement.
IF itrans_destination.SQLCode = -1 THEN
   Beep (1)
   MessageBox("Database Connection Error", &
   "Problem when disconnecting from " + &
   "the destination (Sales) database. " + &
   "Please call technical support." + &
   "~n~r~n~rDetails follow: " + &
   String(itrans_destination.SQLDBCode) + " " + &
      itrans_destination.SQLErrText, Exclamation!)
END IF

DESTROY itrans_destination
```

PART 5

Program Access Techniques

This part presents a collection of techniques you can use to implement program access features in the applications you develop with PowerBuilder. It includes using DDE in an application, using OLE in an application, building a mail-enabled application, and adding other processing extensions.

CHAPTER 18

Using DDE in an Application

About this chapter

This chapter describes how PowerBuilder supports DDE.

Contents

About DDE

Dynamic Data Exchange (DDE) makes it possible for two Windows applications to communicate with each other by sending and receiving commands and data. Using DDE, the applications can share data, execute commands remotely, and check error conditions.

PowerBuilder supports DDE by providing PowerScript events and functions that enable a PowerBuilder application to send messages to other DDE-supporting applications and to respond to DDE requests from other DDE applications.

Clients and servers

A DDE-supporting application can act as either a client or a server.

About the terminology
Used in connection with DDE, these terms are not related to *client/server architecture*, in which a PC or workstation client communicates with a database server.

A **client** application makes requests of another DDE-supporting application (called the server). The requests can be commands (such as open, close, or save) or requests for data.

A **server** application is the opposite of a client application. It responds to requests from another DDE-supporting application (called the client). As with client applications, the requests can be commands or requests for specific data.

A PowerBuilder application can function as a DDE client or as a DDE server.

In PowerBuilder, DDE clients and servers call built-in functions and process events. DDE events occur when a command or data is sent from a client to a server (or from a server to a client).

DDE functions and events

The following tables list the DDE functions and events separated into those functions and events used by DDE clients and those used by DDE servers. For more information on DDE support, see the *PowerScript Reference*.

Return values
Every DDE function returns an integer.

DDE client

Table 18-1: DDE client functions

Function	Action
CloseChannel	Closes a channel to a DDE server application that was opened using OpenChannel.
ExecRemote	Asks a DDE server application to execute a command.
GetDataDDE	Obtains the new data from a hot-linked DDE server application and moves it into a specified string.
GetDataDDEOrigin	Determines the origin of data that has arrived from a hot-linked DDE server application.
GetRemote	Asks a DDE server application for data. This function has two formats: one that uses a channel and one that does not.
OpenChannel	Opens a DDE channel to a specified DDE server application.
RespondRemote	Indicates to the DDE server application whether the command or data received from the DDE application was acceptable to the DDE client.
SetRemote	Asks a DDE server application to set an item such as a cell in a worksheet or a variable to a specific value. This function has two formats: one that uses a DDE channel and one that does not.
StartHotLink	Initiates a hot link to a DDE server application so that PowerBuilder is immediately notified of specific data changes in the DDE server application.
StopHotLink	Ends a hot link with a DDE server application.

Table 18-2: DDE client event

Event	Occurs when
HotLinkAlarm	A DDE server application has sent new (changed) data.

DDE server

Table 18-3: DDE server functions

Function	Action
GetCommandDDE	Obtains the command sent by a DDE client application
GetCommandDDEOrigin	Determines the origin of a command from a DDE client
GetDataDDE	Gets data that a DDE client application has sent and moves it into a specified string
GetDataDDEOrigin	Determines the origin of data that has arrived from a hot-linked DDE client application
RespondRemote	Indicates to the sending DDE client application whether the command or data received from the DDE application was acceptable to the DDE server
SetDataDDE	Sends specified data to a DDE client application
StartServerDDE	Causes a PowerBuilder application to begin acting as a DDE server
StopServerDDE	Causes a PowerBuilder application to stop acting as a DDE server

Table 18-4: DDE server events

Event	Occurs when
RemoteExec	A DDE client application has sent a command
RemoteHotLinkStart	A DDE client application wants to start a hot link
RemoteHotLinkStop	A DDE client application wants to end a hot link
RemoteRequest	A DDE client application has requested data
RemoteSend	A DDE client application has sent data

CHAPTER 19 Using OLE in an Application

About this chapter

This chapter describes several ways of implementing OLE in your PowerBuilder applications.

Contents

OLE support in PowerBuilder

OLE, originally an acronym for Object Linking and Embedding, is a facility that allows Windows programs to share data and program functionality. PowerBuilder OLE controls are **containers** that can call upon OLE **server** applications to display and manipulate OLE objects.

OLE control

The OLE control in the Window painter allows you to link or embed components from several applications in a window. For most servers, you can also control the server application using functions and properties defined by that server.

In PowerBuilder, the OLE control is a container for an OLE object. The user can activate the control and edit the object using functionality supplied by the server application. You can also automate OLE interactions by programmatically activating the object and sending commands to the server. OLE servers might be either DLLs or separate *EXE* files. They could be running on a different computer.

You can use PowerScript automation on an OLE control that is visible in a window, or use it invisibly on an object whose reference is stored in an OLEObject variable. The OLEObject datatype lets you create an OLE object without having an OLE container visible in a window.

OLECustomControl

A second control, OLECustomControl, is a container for an ActiveX control (also called an OLE custom control or OCX control). ActiveX controls are DLLs (sometimes with the extension *OCX*) that always run in the same process as the application that contains them.

Managing OLE objects

You can manage OLE objects by storing them in variables and saving them in files. There are two object types for this purpose: OLEStorage and OLEStream. Most applications will not require these objects, but if you need to do something complicated (such as combining several OLE objects into a single data structure), you can use these objects and their associated functions.

Other areas of OLE support

For information about OLE objects in a DataWindow object, see the PowerBuilder *User's Guide*.

For information about PowerBuilder as a COM or OLE server, see Chapter 27, “Building a COM or COM+ Component,” and Chapter 20, “PowerBuilder Runtime Automation Server.”

OLE controls in a window

You can add OLE objects and ActiveX controls to a window or user object. To do so, you use one of the PowerBuilder OLE controls, which acts as an OLE container. This section explains how you select the control you want by choosing whether it holds an OLE object (also called an insertable object) or an ActiveX control:

- An **insertable OLE object** is a document associated with a server application. The object can be activated and the server provides commands and toolbars for modifying the object.
- An **ActiveX control** or **OLE custom control** is itself a server that processes user actions according to scripts you program in PowerBuilder. You can write scripts for ActiveX control events and for events of the PowerBuilder container. Those scripts call functions and set properties that belong to the ActiveX control. When appropriate, the ActiveX control can present its own visual interface for user interaction.

ActiveX controls range from simple visual displays (such as a meter or a gauge) to single activities that are customizable (spellchecking words or phrases) to working environments (image acquisition with annotation and editing).

OLE control container features

All OLE control containers support a set of required interfaces. PowerBuilder provides some additional support:

- **Extended control** An OLE control can determine and modify its location at runtime using its extended control properties. PowerBuilder supports the X (Left), Y (Top), Width, and Height properties, all of which are measured in PowerBuilder units. The control writer can access these properties using the IDispatch-based interface returned from the GetExtendedControl method on the IOleControlSite interface.
- **Window as OLE container** PowerBuilder implements the IOleContainer class at the window level, so that all OLE controls on a window are siblings and can obtain information about each other. The control writer can access this information using the OLE EnumObjects method. Information about siblings is useful when the controls are part of a suite of controls. Unlike other controls, the OLE controls on a window are stored in a flat hierarchy.

  OLE objects and controls only
  Only OLE objects and controls are visible to this object enumerator. You cannot use this technique to manipulate other controls on the window.

- **Message reflection** If a control container does not support message reflection, a reflector window is created when an OLE control sends a message to its parent. The reflector window reflects the message back to the control so that the control can process the message itself. If the container supports message reflection, the need for a reflector window, and the associated runtime overhead, is eliminated. PowerBuilder OLE control containers perform message reflection for a specific set of messages.

Defining the control

This procedure describes how to create an OLE control and select its contents.

❖ **To place an OLE control in a window or user object:**

1 Open the window or user object that will contain the OLE control.

2 Select Insert>Control>OLE from the menu bar.

PowerBuilder displays the Insert Object dialog box. There are three tabs to choose from.

3 Choose a server application or a specific object for the control (which embeds or links an object in the control), select a custom control, or leave the control empty for now:

- *To create and embed a new object*, click the Create New tab. After you have chosen a server application, click OK.
- *To choose an existing object for the control,* click the Create From File tab. After you have specified the file, click OK.
- *To insert a custom control* (ActiveX control), click the Insert Control tab. After you have chosen an ActiveX control, click OK.
- *To leave the control empty*, click Cancel.

  If you click Cancel, the control becomes an OLE control rather than an OLE custom control, and you can choose to link or embed an OLE object in it at any time; you cannot insert an ActiveX control later.

4 Click where you want the control.

If you inserted an object, PowerBuilder opens the server application so you can view and edit the object. ActiveX controls cannot be opened.

If you want to insert an object that belongs to an OLE server application that is not in the list, see the server documentation to find out how to install it.

For more information about using the Insert Object dialog box, see the section on inserting OLE objects in DataWindow objects in the PowerBuilder *User's Guide*.

OLE controls and insertable objects

The OLE control contains an insertable OLE object. You can change the object in the control in the painter or in a script. You specify what is allowed in the control by setting PowerBuilder properties.

Setting up the OLE control

When you create an OLE control and insert an object, PowerBuilder activates the server application to allow you to modify the object. After you deactivate it (by clicking outside the object's borders in the Layout view), you can use the control's property sheets to set up the control.

- ❖ **To specify the control's appearance and behavior:**

1 Double-click the control, or select Properties from the control's pop-up menu.

2 In the Properties view, give the control a name that is relevant to your application.

You will use this name in scripts. The default name is ole_ followed by a number.

3 Specify a value for Display Name for use by the OLE server. The OLE server can use this name in window title bars.

4 Specify the control's appearance and behavior by choosing appropriate settings in the Properties view.

In addition to the standard Visible, Enabled, Focus Rectangle, and Border properties, which are available for most controls, there are several options that control the object's interaction with the server:

Option	Meaning
Activation	How the user activates the control. Options are: • Double Click – When the user double-clicks the control, the server application is activated. • Get Focus – When the user clicks or tabs to the control, the server is activated. If you also write a script for the GetFocus event, do not call MessageBox or any function that results in a change in focus. • Manual – The control can be activated only programmatically with the Activate function. The control can always be activated programmatically, regardless of the Activation setting.
Display Type	What the control displays. Options are: • Contents – Display a representation of the object, reduced to fit within the control. • Icon – Display the icon associated with the data. This is usually an icon provided by the server application. • ActiveX document – Display as an ActiveX document. ActiveX documents fill the space of the container and have access to all the features of the server application.

Option	Meaning
Contents	What the user can insert in the control at runtime. Options are: • Any – The user can insert either a linked or embedded object. • Embedded – The user can insert an embedded object. • Linked – The user can insert a linked object. Setting Contents changes the value of the ContentsAllowed property.
Link Update	When the object in the control is linked, the method for updating link information. Options are: • Automatic – If the link is broken and PowerBuilder cannot find the linked file, it displays a dialog box in which the user can specify the file. • Manual – If the link is broken, the object cannot be activated. You can re-establish the link in a script using the LinkTo or UpdateLinksDialog function. Setting Link Update changes the value of the LinkUpdateOptions property.
Size Mode	How the object is displayed in the container. Options are: • Clip – The object's image displays full size. If it is larger than the OLE control, it is clipped by the control's borders. • Stretch – The object's image is resized to fit into and fill the OLE control (default).

Activating the object in the painter

The object in the OLE control needs to be activated so that the server application can manipulate it. For the user, double-clicking is the default method for activating the object. You can choose other methods by setting the control's Activation property, as described in the preceding table. During development, you activate the object in the Window painter.

❖ **To activate an OLE object in the Window painter:**

1. Select Open from the control's pop-up menu.

   If the control is empty, Open is unavailable. You must select Insert to assign an object to the control first.

PowerBuilder invokes the server application and activates the object offsite.

2 Use the server application to modify the object.

3 When you have finished, deactivate the object by clicking outside its hatched border.

You can also choose Exit or Return on the server's File menu, if available.

Changing the object in the control

In the painter, you can change or remove the object in the control.

❖ **To delete the object in the control:**

- Select Delete from the control's pop-up menu.

  The control is now empty and cannot be activated. Do not select Clear—it deletes the control from the window.

❖ **To insert a different object in the control:**

1 Select Insert from the control's pop-up menu.

PowerBuilder displays the Insert Object dialog box.

2 Select Create New and select a server application, or select Create from File and specify a file, as you did when you defined the control.

3 Click OK.

During execution

You can insert a different object in the control by calling the InsertObject, InsertFile, InsertClass, or LinkTo function. You can delete the object in the control by calling Cut or Clear.

How the user interacts with the control

When the window containing the OLE control opens, the data is displayed using the information stored with the control in the *PBL* (or *PBD* or *EXE* file if the application has been built).

When the object is activated, either because the user double-clicks or tabs to it or because a script calls Activate, PowerBuilder starts the server application and enables in-place editing if possible. If not, it enables offsite editing.

As the user changes the object, the data in the control is kept up to date with the changes. This is obvious when the object is being edited in place, but it is also true for offsite editing. Because some server applications update the object periodically, rather than continually, the user might see only periodic changes to the contents of the control. Most servers also do a final update automatically when you end the offsite editing session. However, to be safe, the user should select the server's Update command before ending the offsite editing session.

Linking versus embedding

An OLE object can be linked or embedded in your application. The method you choose depends on how you want to maintain the data.

Embedding data

The data for an **embedded** object is stored in your application. During development, it is stored in your application's *PBL*. When you build your application, it is stored in the *EXE* or *PBD* file. This data is a template or a starting point for the user. Although the user can edit the data during a session, the changes cannot be saved because the embedded object is stored as part of your application.

Embedding is suitable for data that will not change (such as the body of a form letter) or as a starting point for data that will be changed and stored elsewhere.

To save the data at runtime, you can use the SaveAs and Open functions to save the user's data to a file or OLE storage.

Linking data

When you **link** an object, your application contains a reference to the data, not the data itself. The application also stores an image of the data for display purposes. The server application handles the actual data, which is usually saved in a file. Other applications can maintain links to the same data. If any application changes the data, the changes appear in all the documents that have links to it.

Linking is useful for two reasons:

- More than one application can access the data.
- The server manages the saving of the data, which is useful even if your PowerBuilder application is the only one using the data.

Maintaining link information The server, not PowerBuilder, maintains the link information. Information in the OLE object tells PowerBuilder what server to start and what data file and item within the file to use. From then on, the server services the data: updating it, saving it back to the data file, updating information about the item (for example, remembering that you inserted a row in the middle of the range of linked rows).

Fixing a broken link Because the server maintains the link, you can move and manipulate an OLE object within your application without worrying about whether it is embedded or linked.

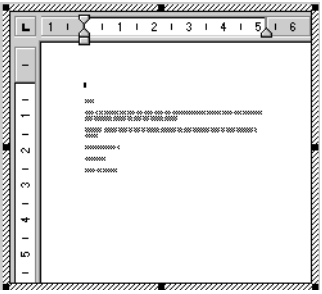
If the link is broken because the file has been moved, renamed, or deleted, the Update setting of the control determines how the problem is handled. When Update is set to Automatic, PowerBuilder displays a dialog box that prompts the user to find the file. You can call the UpdateLinksDialog function in a script to display the same dialog box. You can establish a link in a script without involving the user by calling the LinkTo function.

PowerBuilder displays a control with a linked object with the same shading that is used for an open object.

Offsite or in-place activation

During execution, when a user activates the object in the OLE control, PowerBuilder tries to activate an embedded object **in place**, meaning that the user interacts with the object inside the PowerBuilder window. The menus provided by the server application are merged with the PowerBuilder application's menus. You can control how the menus are merged in the Menu painter (see "Menus for in-place activation" on page 326).

When the control is active in place, it has a wide hatched border:



Offsite activation means that the server application opens and the object becomes an open document in the server's window. All the server's menus are available. The control itself is displayed with shading, indicating that the object is open in the server application.

Limits to in-place activation
The server's capabilities determine whether PowerBuilder can activate the object in place. OLE 1.0 objects cannot be activated in place. In addition, the OLE 2.0 standards specify that linked objects are activated offsite, not in place.

From the Window painter, the object is always activated offsite.

Changing the default behavior

You can change the default behavior in a script by calling the Activate function and choosing whether to activate an object in place or offsite. If you set the control's Activation setting to Manual, you can write a script that calls the Activate function for the DoubleClicked event (or some other event):

```
ole_1.Activate(Offsite!)
```

When the control will not activate
You cannot activate an empty control (a control that does not have an OLE object assigned to it). If you want the user to choose the OLE object, you can write a script that calls the InsertObject function.

If the object in the control is linked and the linked file is missing, the user cannot activate the control. If the Update property is set to Automatic, PowerBuilder displays a dialog box so that the user can find the file.

If the Update property is set to Manual, a script can call the UpdateLinksDialog function to display the dialog box, or call LinkTo to replace the contents with another file.

Menus for in-place activation

When an object is activated in place, menus for its server application are merged with the menus in your PowerBuilder application. The Menu Merge Option settings in the Menu painter let you control how the menus of the two applications are merged. The values are standard menu names, as well as the choices Merge and Exclude.

❖ **To control what happens to a menu in your application when an OLE object is activated:**

1. Open the menu in the Menu painter.
2. Select a menu item that appears on the menu bar. Menu Merge Option settings apply only to items on the menu bar, not items on drop-down menus.
3. On the Style property page, choose the appropriate Menu Merge Option setting. Table 19-1 lists these settings.

Table 19-1: Menu Merge Option settings

You can choose	Meaning	Source of menu in resulting menu bar
File	The menu from the container application (your PowerBuilder application) that will be leftmost on the menu bar. The server's File menu never displays.	Container
Edit	The menu identified as Edit never displays. The server's Edit menu displays.	Server
Window	The menu from the container application that has the list of open sheets. The server's Window menu never displays.	Container
Help	The menu identified as Help never displays. The server's Help menu displays.	Server
Merge	The menu will be displayed after the first menu of the server application.	Container
Exclude	The menu will be removed while the object is active.	

4. Repeat steps 2 and 3 for each item on the menu bar.

Standard assignments for standard menus

In general, you should assign the File, Edit, Window, and Help Menu Merge options to the File, Edit, Window, and Help menus. Because the actual menu names might be different in an international application, you use the Menu Merge Option settings to make the correct associations.

Resulting menu bar for activated object

The effect of the Menu Merge Option settings is that the menu bar displays the container's File and Window menus and the server's Edit and Help menus. Any menus that you label as Merge are included in the menu bar at the appropriate place. The menu bar also includes other menus that the server has decided are appropriate.

Modifying an object in an OLE control

When an OLE object is displayed in the OLE control, the user can interact with that object and the application that created it (the server). You can also program scripts that do the same things the user might do. This section describes how to:

- Activate the OLE object and send general commands to the server
- Change and save the object in the control
- Find out when data or properties have changed by means of events

For information about automation for the control, see "OLE objects in scripts" on page 346.

Activating the OLE object

Generally, the OLE control is set so that the user can activate the object by double-clicking. You can also call the Activate function to activate the object in a script. If the control's Activation property is set to Manual, you have to call Activate to start a server editing session:

```
ole_1.Activate(InPlace!)
```

You can initiate general OLE actions by calling the DoVerb function. A **verb** is an integer value that specifies an action to be performed. The server determines what each integer value means. The default action, specified as 0, is usually Edit, which also activates the object.

For example, if ole_1 contains a Microsoft Excel spreadsheet, the following statement activates the object for editing:

```
ole_1.DoVerb(0)
```

Check the server's documentation to see what verbs it supports. OLE verbs are a relatively limited means of working with objects; automation provides a more flexible interface. OLE 1.0 servers support verbs but not automation.

Changing the object in an OLE control

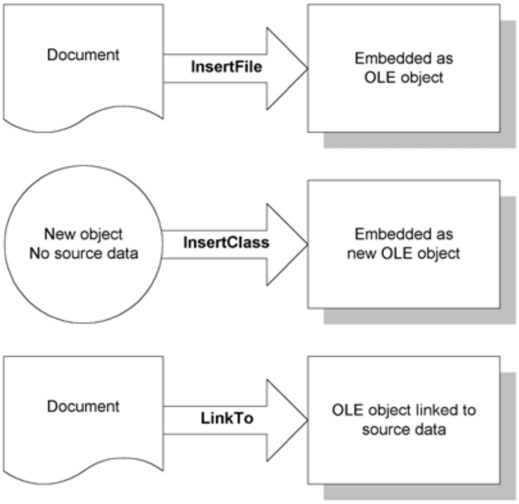
PowerBuilder provides several functions for changing the object in an OLE control. The function you choose depends on whether you want the user to choose an object and whether the object should be linked or embedded, as shown in Table 19-2.

Table 19-2: Functions for changing object in OLE control

When you want to	Choose this function
Let the user choose an object and, if the control's Contents property is set to Any, whether to link or embed it.	InsertObject
Create a new object for a specified server and embed it in the control.	InsertClass
Embed a copy of an existing object in the control.	InsertFile
Link to an existing object in the control.	LinkTo
Open an existing object from a file or storage. Information in the file determines whether the object is linked or embedded.	Open

Figure 19-1 illustrates the behavior of the three functions that do not allow a choice of linking or embedding.

Figure 19-1: Functions that do not allow a choice of linking or embedding


Document
InsertFile
Embedded as OLE object
New object No source data
InsertClass
Embedded as new OLE object
Document
LinkTo
OLE object linked to source data

You can also assign OLE object data stored in a blob to the ObjectData property of the OLE control:

```
blob myblob
... // Code to assign OLE data to the blob
ole_1.ObjectData = myblob
```

The Contents property of the control specifies whether the control accepts embedded and/or linked objects. It determines whether the user can choose to link or embed in the InsertObject dialog box. It also controls what the functions can do. If you call a function that chooses a method that the Contents property does not allow, the function will fail.

OLE information in the Browser

Use the Browser to find out the registered names of the OLE server applications installed on your system. You can use any of the names listed in the Browser as the argument for the InsertClass function, as well as the ConnectToObject and ConnectToNewObject functions (see "Programmable OLE Objects" on page 336).

For more information about OLE and the Browser, see "OLE information in the Browser" on page 363.

Using the clipboard

Using the Cut, Copy, and Paste functions in menu scripts lets you provide clipboard functionality for your user. Calling Cut or Copy for the OLE control puts the OLE object it contains on the clipboard. The user can also choose Cut or Copy in the server application to place data on the clipboard. (Of course, you can use these functions in any script, not just those associated with menus.)

There are several Paste functions that can insert an object in the OLE control. The difference is whether the pasted object is linked or embedded.

Table 19-3: Paste functions

When you want to	Choose this function
Embed the object on the clipboard in the control	Paste
Paste and link the object on the clipboard	PasteLink
Allow the user to choose whether to embed or link the pasted object	PasteSpecial

If you have a general Paste function, you can use code like the following to invoke PasteSpecial (or PasteLink) when the target of the paste operation is the OLE control:

```
graphicobject lg_obj
datawindow ldw_dw
olecontrol lole_ctl

// Get the object with the focus
lg_obj = GetFocus()

// Insert clipboard data based on object type
CHOOSE CASE TypeOf(lg_obj)
   CASE DataWindow!
      ldw_dw = lg_obj
   ldw_dw.Paste()
   ...
   CASE OLEControl!
   lole_ctl = lg_obj
   lole_ctl.PasteSpecial()
END CHOOSE
```

Saving an embedded object

If you embed an OLE object when you are designing a window, PowerBuilder saves the object in the library with the OLE control. However, when you embed an object during execution, that object cannot be saved with the control because the application's executable and libraries are read-only. If you need to save the object, you save the data in a file or in the database.

For example, the following script uses SaveAs to save the object in a file. It prompts the user for a file name and saves the object in the control as an OLE data file, not as native data of the server application. You can also write a script to open the file in the control in another session:

```
string myf
ilename, mypathname
integer result
GetFileSaveName("Select File", mypathname, &
   myfilename, "OLE", &
   "OLE Files (*.OLE),*.OLE")
result = ole_1.SaveAs(myfilename)
```

When you save OLE data in a file, you will generally not be able to open that data directly in the server application. However, you can open the object in PowerBuilder and activate the server application.

When you embed an object in a control, the actual data is stored as a blob in the control's ObjectData property. If you want to save an embedded object in a database for later retrieval, you can save it as a blob. To transfer data between a blob variable and the control, assign the blob to the control's ObjectData property or vice versa:

```
blob myblob
myblob = ole_1.ObjectData
```

You can use the embedded SQL statement UPDATEBLOB to put the blob data in the database (see the *PowerScript Reference*).

You can also use SaveAs and Save to store OLE objects in PowerBuilder's OLEStorage variables (see "Opening and saving storages" on page 368).

When the user saves a linked object in the server, the link information is not affected and you do not need to save the open object. However, if the user renames the object or affects the range of a linked item, you need to call the Save function to save the link information.

Events for the OLE control

There are several events that let PowerBuilder know when actions take place in the server application that affect the OLE object.

Events for data

Events that have to do with data are:

- **DataChange** The data has been changed
- **Rename** The object has been renamed
- **Save, SaveObject** The data has been saved
- **ViewChange** The user has changed the view of the data

When these events occur, the changes are reflected automatically in the control. If you need to perform additional processing when the object is renamed, saved, or changed, you can write the appropriate scripts.

Because of the architecture of OLE, you often cannot interact with the OLE object within these events. Trying to do so can generate a runtime error. A common workaround is to use the PostEvent function to post the event to an asynchronous event handler. You do not need to post the SaveObject event, which is useful if you want to save the data in the object to a file whenever the server application saves the object.

Events for properties

If the server supports property notifications, then when values for properties of the server change, the PropertyRequestEdit and PropertyChanged events will occur. You can write scripts that cancel changes, save old values, or read new values.

For more information about property notification, see “Creating hot links” on page 358.

OLE custom controls

The OLE control button in the Controls menu gives you the option of inserting an object or a custom control in an OLE container. When you select an OLE custom control (ActiveX control), you fix the container's type and contents. You cannot choose later to insert an object and you cannot select a different custom control.

Each ActiveX control has its own properties, events, and functions. Preventing the ActiveX control from being changed helps avoid errors later in scripts that address the properties and methods of a particular ActiveX control.

Setting up the custom control

The PowerBuilder custom control container has properties that apply to any ActiveX control. The ActiveX control itself has its own properties. This section describes the purpose of each type of property and how to set them.

PowerBuilder properties

For OLE custom controls, PowerBuilder properties have two purposes:

- To specify appearance and behavior of the container, as you do for any control

  You can specify position, pointer, and drag-and-drop settings, as well as the standard settings on the General property page (Visible, Enabled, and so on).

- To provide default information that the ActiveX control can use

  Font information and the display name are called **ambient properties** in OLE terminology. PowerBuilder does not display text for the ActiveX control, so it does not use these properties directly. If the ActiveX control is programmed to recognize ambient properties, it can use the values PowerBuilder provides when it displays text or needs a name to display in a title bar.

❖ **To modify the PowerBuilder properties for the custom control:**

1. Double-click the control, or select Properties from the control's pop-up menu.

   The OLE Custom Control property sheet displays.

2. Give the control a name that is relevant to your application. You will use this name in scripts. The default name is ole_ followed by a number.

3 Specify values for other properties on the General property page and other pages as appropriate.

4 Click OK when you are done.

Documenting the control
Put information about the ActiveX control you are using in a comment for the window or in the control's Tag property. Later, if another developer works with your window and does not have the ActiveX control installed, that developer can easily find out what ActiveX control the window was designed to use.

ActiveX control properties

An ActiveX control usually has its own properties and its own property sheet for setting property values. These properties control the appearance and behavior of the ActiveX control, not the PowerBuilder container.

❖ **To set property values for the ActiveX control in the control:**

1 Select OLE Control Properties from the control's pop-up menu or from the General property page.

2 Specify values for the properties and click OK when done.

The OLE control property sheet might present only a subset of the properties of the ActiveX control. You can set other properties in a script.

For more information about the ActiveX control's properties, see the documentation for the ActiveX control.

Programming the ActiveX control

You make an ActiveX control do its job by programming it in scripts, setting its properties, and calling its functions. Depending on the interface provided by the ActiveX control developer, a single function call might trigger a whole series of activities or individual property settings, and function calls may let you control every aspect of its actions.

An ActiveX control is always active—it does not contain an object that needs to be opened or activated. The user does not double-click and start an OLE server. However, you can program the DoubleClicked or any other event to call a function that starts ActiveX control processing.

Setting properties in scripts

Programming an ActiveX control is the same as programming automation for insertable objects. You use the container's *Object* property to address the properties and functions of the ActiveX control.

This syntax accesses a property value. You can use it wherever you use an expression. Its datatype is Any. When the expression is evaluated, its value has the datatype of the control property:

olecontrol.**Object**.*ocxproperty*

This syntax calls a function. You can capture its return value in a variable of the appropriate datatype:

{ *value* } = *olecontrol*.**Object**.*ocxfunction* ({ *argumentlist* })

Errors when accessing properties

The PowerBuilder compiler does not know the correct syntax for accessing properties and functions of an ActiveX control, so it does not check any syntax after the Object property. This provides the flexibility you need to program any ActiveX control. But it also leaves an application open to runtime errors if the properties and functions are misnamed or missing.

PowerBuilder provides two events (ExternalException and Error) for handling OLE errors. If the ActiveX control defines a stock error event, the PowerBuilder OLE control container has an additional event, ocx_event. These events allow you to intercept and handle errors without invoking the SystemError event and terminating the application. You can also use a TRY-CATCH exception handler.

For more information, see “Handling errors” on page 354.

Using events of the ActiveX control

An ActiveX control has its own set of events, which PowerBuilder merges with the events for the custom control container. The ActiveX control events appear in the Event List view with the PowerBuilder events. You write scripts for ActiveX control events in PowerScript and use the Object property to refer to ActiveX control properties and methods, just as you do for PowerBuilder event scripts.

The only difference between ActiveX control events and PowerBuilder events is where to find documentation about when the events get triggered. The ActiveX control provider supplies the documentation for its events, properties, and functions.

The PowerBuilder Browser provides lists of the properties and methods of the ActiveX control. For more information, see “OLE information in the Browser” on page 363.

New versions of the ActiveX control
If you install an updated version of an ActiveX control and it has new events, the event list in the Window painter does not add the new events. To use the new events, you have to delete and recreate the control, along with the scripts for existing events. If you do not want to use the new events, you can leave the control as is—it will use the updated ActiveX control with the pre-existing events.

Programmable OLE Objects

You do not need to place an OLE control on a window to manipulate an OLE object in a script. If the object does not need to be visible in your PowerBuilder application, you can create an OLE object independent of a control, connect to the server application, and call functions and set properties for that object. The server application executes the functions and changes the object's properties, which changes the OLE object.

For some applications, you can specify whether the application is visible. If it is visible, the user can activate the application and manipulate the object using the commands and tools of the server application.

OLEObject object type

PowerBuilder's OLEObject object type is designed for automation. OLEObject is a dynamic object type, which means that the compiler will accept any property names, function names, and parameter lists for the object. PowerBuilder does not have to know whether the properties and functions are valid. This allows you to call methods and set properties for the object that are known to the server application that created the object. If the functions or properties do not exist during execution, you will get runtime errors.

Using an OLEObject variable involves these steps:

1. Declare the variable and instantiate it.
2. Connect to the OLE object.

3 Manipulate the object as appropriate using the OLE server's properties and functions.

4 Disconnect from the OLE object and destroy the variable.

These steps are described next.

Declaring an OLEObject variable

You need to declare an OLEObject variable and allocate memory for it:

```
OLEObject myoleobject
myoleobject = CREATE OLEObject
```

The Object property of the OLE container controls (OLEControl or OLECustomControl) has a datatype of OLEObject.

Connecting to the server

You establish a connection between the OLEObject object and an OLE server with one of the ConnectToObject functions. Connecting to an object starts the appropriate server:

Table 19-4: ConnectToObject functions

When you want to	Choose this function
Create a new object for an OLE server that you specify. Its purpose is similar to InsertClass for a control.	ConnectToNewObject
Create a new OLE object in the specified remote server application if security on the server allows it and associate the new object with a PowerBuilder OLEObject variable.	ConnectToNewRemoteObject
Open an existing OLE object from a file. If you do not specify an OLE class, PowerBuilder uses the file's extension to determine what server to start.	ConnectToObject
Associate an OLE object with a PowerBuilder OLEObject variable and start the remote server application.	ConnectToRemoteObject

After you establish a connection, you can use the server's command set for automation to manipulate the object (see “OLE objects in scripts” on page 346).

You do not need to include application qualifiers for the commands. You already specified those qualifiers as the application's class when you connected to the server. For example, the following commands create an OLEObject variable, connect to Microsoft Word 's OLE interface (word.application), open a document and display information about it, insert some text, save the edited document, and shut down the server:

```
OLEObject o1
```

```
string s1
o1 = CREATE oleobject

o1.ConnectToNewObject("word.application")
o1.documents.open("c:\temp\temp.doc")

// Make the object visible and display the
// MS Word user name and file name
o1.Application.Visible = True
s1 = o1.UserName
MessageBox("MS Word User Name", s1)
s1 = o1.ActiveDocument.Name
MessageBox("MS Word Document Name", s1)

//Insert some text in a new paragraph
o1.Selection.TypeParagraph()
o1.Selection.typetext("Insert this text")
o1.Selection.TypeParagraph()

// Insert text at the first bookmark
o1.ActiveDocument.Bookmarks[1].Select
o1.Selection.typetext("Hail!")

// Insert text at the bookmark named End
o1.ActiveDocument.Bookmarks.item("End").Select
o1.Selection.typetext("Farewell!")

// Save the document and shut down the server
o1.ActiveDocument.Save()
o1.quit()
RETURN
```

For earlier versions of Microsoft Word, use word.basic instead of word.application. The following commands connect to the Microsoft Word 7.0 OLE interface (word.basic), open a document, go to a bookmark location, and insert the specified text:

```
myoleobject.ConnectToNewObject("word.basic")
myoleobject.fileopen("c:\temp\letter1.doc")
myoleobject.editgoto("NameAddress")
myoleobject.Insert("Text to insert")
```

Do *not* include word.application or word.basic (the class in ConnectToNewObject) as a qualifier:

```
// Incorrect command qualifier
myoleobject.word.basic.editgoto("NameAddress")
```

Microsoft Word 7.0 implementation
For an OLEObject variable, word.basic is the class name of Word 7.0 as a server application. For an object in a control, you must use the qualifier application.wordbasic to tell Word how to traverse its object hierarchy and access its wordbasic object.

Shutting down and disconnecting from the server

After your application has finished with the automation, you might need to tell the server explicitly to shut down. You can also disconnect from the server and release the memory for the object:

```
myoleobject.Quit()
rtncode = myoleobject.DisconnectObject()
DESTROY myoleobject
```

You can rely on garbage collection to destroy the OLEObject variable. Destroying the variable automatically disconnects from the server.

It is preferable to use garbage collection to destroy objects, but if you want to release the memory used by the variable immediately and you know that it is not being used by another part of the application, you can explicitly disconnect and destroy the OLEObject variable, as shown in the code above.

For more information, see “Garbage collection and memory management” on page 45.

Assignments among OLEControl, OLECustomControl, and OLEObject datatypes

You cannot assign an OLE control (object type OLEControl) or ActiveX control (object type OLECustomControl) to an OLEObject.

If the vendor of the control exposes a programmatic identifier (in the form *vendor.application*), you can specify this identifier in the ConnectToNewObject function to connect to the programmable interface without the visual control. For an ActiveX control with events, this technique makes the events unavailable. ActiveX controls are not meant to be used this way and would not be useful in most cases.

You can assign the Object property of an OLE control to an OLEObject variable or use it as an OLEObject in a function.

For example, if you have an OLEControl ole_1 and an OLECustomControl ole_2 in a window and you have declared this variable:

```
OLEObject oleobj_automate
```

then you can make these assignments:

```
oleobj_automate = ole_1.Object
oleobj_automate = ole_2.Object
```

You cannot assign an OLEObject to the Object property of an OLE control because it is read-only. You cannot make this assignment:

```
ole_1.Object = oleobj_automate //Error!
```

Events for OLEObjects

You can implement events for an OLEObject by creating a user object that is a descendant of OLEObject. The SetAutomationPointer PowerScript function assigns an OLE automation pointer to the descendant so that it can use OLE automation.

Suppose oleobjectchild is a descendant of OLEObject that implements events such as the ExternalException and Error events. The following code creates an OLEObject and an instance of oleobjectchild, which is a user object that is a descendant of OLEObject, connects to Excel, then assigns the automation pointer to the oleobjectchild:

```
OLEObject ole1
oleobjectchild  oleChild

ole1 = CREATE OLEObject
ole1.ConnectToNewObject( "Excel.Application")

oleChild = CREATE oleobjectchild
oleChild.SetAutomationPointer( ole1 )
```

You can now use olechild for automation.

Automation scenario

The steps involved in automation can be included in a single script or be the actions of several controls in a window. If you want the user to participate in the automation, you might:

- Declare an OLE object as an instance variable of a window
- Instantiate the variable and connect to the server in the window’s Open event

- Send commands to the server in response to the user's choices and specifications in lists or edit boxes
- Disconnect and destroy the object in the window's Close event

If the automation does not involve the user, all the work can be done in a single script.

Example: generating form letters using OLE

This example takes names and addresses from a DataWindow object and letter body from a MultiLineEdit and creates and prints letters in Microsoft Word using VBA scripting.

❖ **To set up the form letter example:**

1 Create a Word document called *CONTACT.DOC* with four bookmarks and save the file in your PowerBuilder directory.

These are the bookmarks:

- name1 – for the name in the return address
- name2 – for the name in the salutation
- address1 – for the street, city, state, and zip in the return address
- body – for the body of the letter

The letter should have the following content:

```
Multimedia Promotions, Inc.
1234 Technology Drive
Westboro, Massachusetts
January 12, 2003

[bookmark name1]
[bookmark address1]

Dear [bookmark name2]:
[bookmark body]

Sincerely,
Harry Mogul
President
```

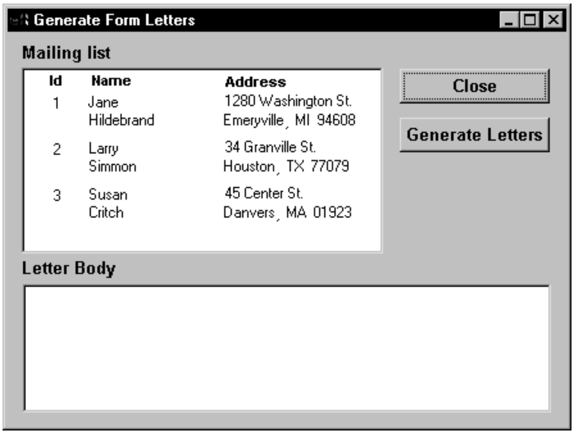
You could enhance the letter with a company and a signature logo. The important items are the names and placement of the bookmarks.

2 In PowerBuilder, define a DataWindow object called d_maillist that has the following columns:

id
first_name
last_name
street
city
state
zip

You can turn on Prompt for Criteria in the DataWindow object so the user can specify the customers who will receive the letters.

3 Define a window that includes a DataWindow control called dw_mail, a MultiLineEdit called mle_body, and a CommandButton or PictureButton:



Id	Name	Address
1	Jane Hildebrand	1280 Washington St. Emeryville, MI 94608
2	Larry Simmon	34 Granville St. Houston, TX 77079
3	Susan Critch	45 Center St. Danvers, MA 01923

4 Assign the DataWindow object d_maillist to the DataWindow control dw_mail.

5 Write a script for the window's Open event that connects to the database and retrieves data for the DataWindow object. The following code connects to a SQL Anywhere database. (When the window is part of a larger application, the connection is typically done by the application Open script.)

```
/***************************************************
Set up the Transaction object from the INI file
***************************************************/
SQLCA.DBMS=ProfileString("myapp.ini", &
   "Database", "DBMS", " ")
```

```
SQLCA.DbParm=ProfileString("myapp.ini", &
   "Database", "DbParm", " ")
/***************************************************
Connect to the database and test whether the
connect succeeded
**************************************************/
CONNECT USING SQLCA;
IF SQLCA.SQLCode <> 0 THEN
   MessageBox("Connect Failed", "Cannot connect" &
      + "to database. " + SQLCA.SQLErrText)
   RETURN
END IF
/***************************************************
Set the Transaction object for the DataWindow
control and retrieve data
**************************************************/
dw_mail.SetTransObject(SQLCA)
dw_mail.Retrieve()
```

6 Write the script for the Generate Letters button (the script is shown below).

The script does all the work, performing the following tasks:

- Creates the OLEObject variable
- Connects to the server (word.application)
- For each row in the DataWindow object, generates a letter

  To do so, it uses VBA statements to perform the tasks in Table 19-5.

Table 19-5: Script tasks

VBA statements	Task
open	Opens the document with the bookmarks
goto and typetext	Extracts the name and address information from a row in the DataWindow object and inserts it into the appropriate places in the letter
goto and typetext	Inserts the text the user types in mle_body into the letter
printout	Prints the letter
close	Closes the letter document without saving it

- Disconnects from the server
- Destroys the OLEObject variable

7 Write a script for the Close button. All it needs is one command:

```
Close(Parent)
```

Script for generating form letters

The following script generates and prints the form letters:

```
OLEObject contact_ltr
integer result, n
string ls_name, ls_addr
/***************************************************
Allocate memory for the OLEObject variable
***************************************************/
contact_ltr = CREATE oleObject
/***************************************************
Connect to the server and check for errors
***************************************************/
result = &
   contact_ltr.ConnectToNewObject("word.application")
IF result <> 0 THEN
   DESTROY contact_ltr
   MessageBox("OLE Error", &
      "Unable to connect to Microsoft Word. " &
      + "Code: " &
      + String(result))
      RETURN
END IF
/***************************************************
For each row in the DataWindow, send customer
data to Word and print a letter
***************************************************/
FOR n = 1 to dw_mail.RowCount()
/************************************************
   Open the document that has been prepared with
   bookmarks
************************************************/
   contact_ltr.documents.open("c:\pbdocs\contact.doc")
/************************************************
   Build a string of the first and last name and
   insert it into Word at the name1 and name2
   bookmarks
************************************************/
   ls_name = dw_mail.GetItemString(n, "first_name")&
   + " " + dw_mail.GetItemString(n, "last_name")
   contact_ltr.Selection.goto("name1")
   contact_ltr.Selection.typetext(ls_name)
   contact_ltr.Selection.goto("name2")
   contact_ltr.Selection.typetext(ls_name)
/************************************************
   Build a string of the address and insert it into
   Word at the address1 bookmark
************************************************/
```

```
   ls_addr = dw_mail.GetItemString(n, "street") &
      + "~r~n" &
      + dw_mail.GetItemString(n, "city") &
      + ", " &
      + dw_mail.GetItemString(n, "state") &
      + " " &
      + dw_mail.GetItemString(n, "zip")
   contact_ltr.Selection.goto("address1")
   contact_ltr.Selection.typetext(ls_addr)
/************************************************
   Insert the letter text at the body bookmark
***********************************************/
   contact_ltr.Selection.goto("body")
   contact_ltr.Selection.typetext(mle_body.Text)
/************************************************
   Print the letter
************************************************/
   contact_ltr.Application.printout()
/************************************************
   Close the document without saving
************************************************/
   contact_ltr.Documents.close
   contact_ltr.quit()
NEXT
/***************************************************
Disconnect from the server and release the memory for
the OLEObject variable
***************************************************/
contact_ltr.DisconnectObject()
DESTROY contact_ltr
```

Running the example

To run the example, write a script for the Application object that opens the window or use the Run/Preview button on the PowerBar.

When the application opens the window, the user can specify retrieval criteria to select the customers who will receive letters. After entering text in the MultiLineEdit for the letter body, the user can click on the Generate Letters button to print letters for the listed customers.

OLE objects in scripts

This chapter has described the three ways to use OLE in a window or user object. You have learned about:

- Inserting an object in an OLE control
- Placing an ActiveX control in an OLE custom control
- Declaring an OLEObject variable and connecting to an OLE object

In scripts, you can manipulate these objects by means of OLE **automation**, getting and setting properties, and calling functions that are defined by the OLE server. There are examples of automation commands in the preceding sections. This section provides more information about the automation interface in PowerBuilder.

The automation interface

In PowerBuilder, an OLEObject is your interface to an OLE server or ActiveX control. When you declare an OLEObject variable and connect to a server, you can use dot notation for that variable and send instructions to the server. The instruction might be a property whose value you want to get or set, or a function you want to call.

The general automation syntax for an OLEObject is:

oleobjectvar.*serverinstruction*

For OLE controls in a window, your interface to the server or ActiveX control is the control's Object property, which has a datatype of OLEObject.

The general automation syntax for an OLE control is:

olecontrol.**Object**.*serverinstruction*

Compiling scripts that include commands to the OLE server
When you compile scripts that apply methods to an OLEObject (including a control's Object property), PowerBuilder does not check the syntax of the rest of the command, because it does not know the server's command set. You must ensure that the syntax is correct to avoid errors during execution.

Make sure you give your applications a test run to ensure that your commands to the server application are correct.

What does the server support?

A server's command set includes properties and methods (functions and events).

OLE server applications publish the command set they support for automation. Check your server application's documentation for information.

For custom controls and programmable OLE objects, you can see a list of properties and methods in the PowerBuilder Browser. For more information about OLE information in the Browser, see "OLE information in the Browser" on page 363.

Setting properties

You access server properties for an OLE control through its Object property using the following syntax:

olecontrolname.**Object**.{ *serverqualifiers*.}*propertyname*

If the OLE object is complex, there could be nested objects or properties within the object that serve as qualifiers for the property name.

For example, the following commands for an Excel spreadsheet object activate the object and set the value property of several cells:

```
double value
ole_1.Activate(InPlace!)
ole_1.Object.cells[1,1].value = 55
ole_1.Object.cells[2,2].value = 66
ole_1.Object.cells[3,3].value = 77
ole_1.Object.cells[4,4].value = 88
```

For an Excel 95 spreadsheet, enclose the cells' row and column arguments in parentheses instead of square brackets. For example:

```
ole_1.Object.cells(1,1).value = 55
```

For properties of an OLEObject variable, the server qualifiers and property name follow the variable name:

oleobjectvar.{ *serverqualifiers*.}*propertyname*

The qualifiers you need to specify depend on how you connect to the object. For more information, see "Qualifying server commands" on page 351.

Calling functions

You can call server functions for an OLE control through its Object property using the following syntax:

olecontrolname.**Object**.{ *serverqualifiers*.}*functionname* ({ *arguments* })

If the OLE object is complex, there could be nested properties or objects within the object that serve as qualifiers for the function name.

Required parentheses
PowerScript considers all commands to the server either property settings or functions. For statements and functions to be distinguished from property settings, they must be followed by parentheses surrounding the parameters. If there are no parameters, specify empty parentheses.

Arguments and return values and their datatypes

PowerBuilder converts OLE data to and from compatible PowerBuilder datatypes. The datatypes of values you specify for arguments must be compatible with the datatypes expected by the server, but they do not need to be an exact match.

When the function returns a value, you can assign the value to a PowerBuilder variable of a compatible datatype.

Passing arguments by reference

If an OLE server expects an argument to be passed by reference so that it can pass a value back to your script, include the keyword REF just before the argument. This is similar to the use of REF in an external function declaration:

olecontrol.Object.*functionname* (REF *argname*)

In these generic examples, the server can change the values of *ls_string* and *li_return* because they are passed by reference:

```
string ls_string
integer li_return
ole_1.Object.testfunc(REF ls_string, REF li_return)
```

This example illustrates the same function call using an OLEObject variable.

```
OLEObject ole_obj
ole_obj = CREATE OLEObject
ole_obj.ConnectToNewObject("servername")
ole_obj.testfunc(REF ls_string, REF li_return)
```

Setting the timeout period
Calls from a PowerBuilder client to a server time out after five minutes. You can use the SetAutomationTimeout PowerScript function to change the default timeout period if you expect a specific OLE request to take longer.

Word and automation

Microsoft Word 6.0 and 7.0 support automation with a command set similar to the WordBasic macro language. The command set includes both statements and functions and uses named parameters. Later versions of Microsoft Word use Visual Basic for Applications (VBA), which consists of a hierarchy of objects that expose a specific set of methods and properties.

WordBasic statements WordBasic has both statements and functions. Some of them have the same name. WordBasic syntax differentiates between statements and functions calls, but PowerBuilder does not.

To specify that you want to call a statement, you can include AsStatement! (a value of the OLEFunctionCallType enumerated datatype) as an argument. Using AsStatement! is the only way to call WordBasic statements that have the same name as a function. Even when the statement name does not conflict with a function name, specifying AsStatement! is more efficient:

olecontrol.**Object**.application.wordbasic.*statementname*
(*argumentlist*, AsStatement!)

For example, the following code calls the AppMinimize statement:

```
ole_1.Object.application.wordbasic. &
   AppMinimize("",1,AsStatement!)
```

Named parameters PowerBuilder does not support named parameters that both WordBasic and Visual Basic use. In the parentheses, specify the parameter values without the parameter names.

For example, the following statements insert text at a bookmark in a Word 6.0 or 7.0 document:

```
ole_1.Activate(InPlace!)
Clipboard(mle_nameandaddress.Text)
ole_1.Object.application.wordbasic.&
   fileopen("c:\msoffice\winword\doc1.doc")
ole_1.Object.application.wordbasic.&
   editgoto("NameandAddress", AsStatement!)
ole_1.Object.application.wordbasic.&
   editpaste(1, AsStatement!)
```

The last two commands in a WordBasic macro would look like this, where Destination is the named parameter:

```
EditGoto.Destination = "NameandAddress"
EditPaste
```

In a PowerBuilder script, you would use this syntax to insert text in a Word 97 or later document:

```
ole_1.Object.Selection.TypeText("insert this text")
```

In the corresponding Visual Basic statement, the named parameter Text contains the string to be inserted:

```
Selection.TypeText Text:="insert this text"
```

Automation is not macro programming
You cannot send commands to the server application that declare variables or control the flow of execution (for example, IF THEN). Automation executes one command at a time independently of any other commands. Use PowerScript's conditional and looping statements to control program flow.

Example of Word automation To illustrate how to combine PowerScript with server commands, the following script counts the number of bookmarks in a Microsoft Word OLE object and displays their names:

```
integer i, count
string bookmarklist, curr_bookmark
ole_1.Activate(InPlace!)

count = ole_1.Object.Bookmarks.Count
bookmarklist = "Bookmarks = " + String(count) + "~n"

FOR i = 1 to count
   curr_bookmark = ole_1.Object.Bookmarks[i].Name
   bookmarklist = bookmarklist + curr_bookmark + "~n"
END FOR

MessageBox("BookMarks", bookmarklist)
```

Word automation tip
You can check that you are using the correct syntax for Word automation with the Word macro editor. Turn on macro recording in Word, perform the steps you want to automate manually, then turn off macro recording. You can then type Alt+F11 to open the macro editor and see the syntax that was built. Remember that PowerBuilder uses square brackets for array indexes.

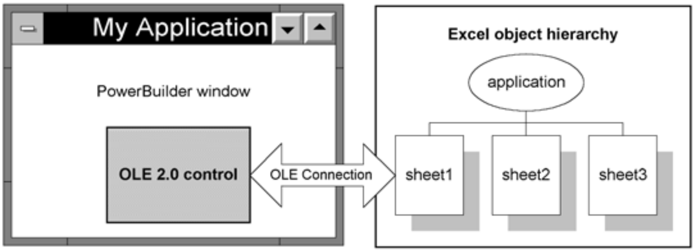
Example of Word 6.0 and 7.0 automation The following script counts the number of bookmarks in a Microsoft Word 6.0 or 7.0 OLE object and displays their names:

```
integer i, count
string bookmarklist, curr_bookmark
ole_1.Activate(InPlace!)

// Get the number of bookmarks
count = ole_1.Object. &
    application.wordbasic.countbookmarks
bookmarklist = "Bookmarks = " + String(count) + "~n"

// Get the name of each bookmark
FOR i = 1 to count
   curr_bookmark = ole_1.Object. &
   application.wordbasic.bookmarkname(i)
   bookmarklist = bookmarklist    + curr_bookmark +
"~n"
END FOR

MessageBox("BookMarks", bookmarklist)
```

Qualifying server commands

Whether to qualify the server command with the name of the application depends on the server and how the object is connected. Each server implements its own version of an object hierarchy, which needs to be reflected in the command syntax. For example, the Microsoft Excel object hierarchy is shown in Figure 19-2.

Figure 19-2: Microsoft Excel object hierarchy



When the server is Excel, the following commands appear to mean the same thing but can have different effects (for an Excel 95 spreadsheet, the cells' row and column arguments are in parentheses instead of square brackets):

```
ole_1.Object.application.cells[1,2].value = 55
```

```
ole_1.Object.cells[1,2].value = 55
```

The first statement changes a cell in the active document. It moves up Excel's object hierarchy to the Application object and back down to an open sheet. It does not matter whether it is the same one in the PowerBuilder control. If the user switches to Excel and activates a different sheet, the script changes that one instead. You should avoid this syntax.

The second statement affects only the document in the PowerBuilder control. However, it will cause a runtime error if the document has not been activated. It is the safer syntax to use, because there is no danger of affecting the wrong data.

Microsoft Word 6.0 and 7.0 implement the application hierarchy differently and require the qualifier application.wordbasic when you are manipulating an object in a control. (You must activate the object.) For example:

```
ole_1.Object.application.wordbasic.bookmarkname(i)
```

Later versions of Microsoft Word do not require a qualifier, but it is valid to specify one. You can use any of the following syntaxes:

```
ole_1.Object.Bookmarks.[i].Name
ole_1.Object.Bookmarks.item(i).Name

ole_1.Object.application.ActiveDocument. &
   Bookmarks.[i].Name
```

When you are working with PowerBuilder’s OLEObject, rather than an object in a control, you omit the application qualifiers in the commands because you have already specified them when you connected to the object. (For more about the OLEObject object type, see “Programmable OLE Objects” on page 336.)

Automation and the Any datatype

Because PowerBuilder knows nothing about the commands and functions of the server application, it also knows nothing about the datatypes of returned information when it compiles a program. Expressions that access properties and call functions have a datatype of Any. You can assign the expression to an Any variable, which avoids datatype conversion errors.

During execution, when data is assigned to the variable, it temporarily takes the datatype of the value. You can use the ClassName function to determine the datatype of the Any variable and make appropriate assignments. If you make an incompatible assignment with mismatched datatypes, you will get a runtime error.

Do not use the Any datatype unnecessarily
If you know the datatype of data returned by a server automation function, do not use the Any datatype. You can assign returned data directly to a variable of the correct type.

The following sample code retrieves a value from Excel and assigns it to the appropriate PowerBuilder variable, depending on the value’s datatype. (For an Excel 95 spreadsheet, the row and column arguments for cells are in parentheses instead of square brackets.)

```
string stringval
double dblval
date dateval
any anyval

anyval = myoleobject.application.cells[1,1].value
CHOOSE CASE ClassName(anyval)
   CASE "string"
      stringval = anyval
   CASE "double"
      dblval = anyval
   CASE "datetime"
      dateval = Date(anyval)
END CHOOSE
```

OLEObjects for efficiency

When your automation command refers to a deeply nested object with multiple server qualifiers, it takes time to negotiate the object's hierarchy and resolve the object reference. If you refer to the same part of the object hierarchy repeatedly, then for efficiency you can assign that part of the object reference to an OLEObject variable. The reference is resolved once and reused.

Instead of coding repeatedly for different properties:

```
ole_1.Object.application.wordbasic.propertyname
```

you can define an OLEObject variable to handle all the qualifiers:

```
OLEObject ole_wordbasic
ole_wordbasic = ole_1.Object.application.wordbasic
ole_wordbasic.propertyname1 = value
ole_wordbasic.propertyname2 = value
```

Example: resolving an object reference

This example uses an OLEObject variable to refer to a Microsoft Word object. Because it is referred to repeatedly in a FOR loop, the resolved OLEObject makes the code more efficient. The example destroys the OLEObject variable when it is done with it:

```
integer li_i, li_count
string ls_curr_bookmark
OLEObject ole_wb

ole_1.Activate(InPlace!)
ole_wb = ole_1.Object.application.wordbasic

// Get the number of bookmarks
li_count = ole_wb.countbookmarks
// Get the name of each bookmark
FOR li_i = 1 to count
   ls_curr_bookmark = ole_wb.bookmarkname(i)
   ... // code to save the bookmark name in a list
END FOR
```

Handling errors

Statements in scripts that refer to the OLE server's properties are not checked in the compiler because PowerBuilder does not know what syntax the server expects. Because the compiler cannot catch errors, runtime errors can occur when you specify property or function names and arguments the OLE server does not recognize.

Chain of error events

When an error occurs that is generated by a call to an OLE server, PowerBuilder follows this sequence of events:

1. If the error was generated by an ActiveX control that has defined a stock error event, the ocx_error event for the PowerBuilder OLE control is triggered.
2. Otherwise, the ExternalException event for the OLE object occurs.
3. If the ExternalException event has no script or its action argument is set to ExceptionFail! (the default), the Error event for the OLE object occurs.
4. If the Error event has no script or its action argument is set to ExceptionFail! (the default), any active exception handler for a RuntimeError or its descendants is invoked.
5. If no exception handler exists, or if the existing exception handlers do not handle the exception, the SystemError event for the Application object occurs.
6. If the SystemError has no script, an application runtime error occurs and the application is terminated.

You can handle the error in any of these events or in a script using a TRY-CATCH block. However, it is not a good idea to continue processing after the SystemError event occurs.

For more information about exception handling, see “Handling exceptions” on page 38.

Events for OLE errors

PowerBuilder OLE objects and controls all have two events for error handling:

- **ExternalException** Triggered when the OLE server or control throws an exception or fires an error event (if there is no ocx_error event). Information provided by the server can help diagnose the error.
- **Error** Triggered when the exception or error event is not handled. PowerBuilder error information is available in the script.

If the OLE control defines a stock error event, the PowerBuilder OLE control container has an additional event:

- **ocx_error** Triggered when the OLE server fires an error event. Information provided by the server can help diagnose the error.

The creator of an OLE control can generate the stock error event for the control using the Microsoft Foundation Classes (MFC) Class Wizard. The arguments for the ocx_error event in PowerBuilder map to the arguments defined for the stock error event.

Responding to the error

If the PowerBuilder OLE control has an ocx_error event script, you can get information about the error from the event's arguments and take appropriate action. One of the arguments of ocx_error is the boolean CancelDisplay. You can set CancelDisplay to TRUE to cancel the display of any MFC error message. You can also supply a different description for the error.

In either the ExternalException or Error event script, you set the Action argument to an ExceptionAction enumerated value. What you choose depends on what you know about the error and how well the application will handle missing information.

Table 19-6: ExceptionAction enumerated values

ExceptionAction value	Effect
ExceptionFail!	Fail as if the event had no script. Failing triggers the next error event in the order of event handling.
ExceptionIgnore!	Ignore the error and return as if no error occurred. **Caution** If you are getting a property value or expecting a return value from a function, a second error can occur during the assignment because of mismatched datatypes.
ExceptionRetry!	Send the command to the OLE server again (useful if the OLE server was not ready). **Caution** If you keep retrying and the failure is caused by an incorrect name or argument, you will set your program into an endless loop. You can set up a counter to limit the number of retries.
ExceptionSubstituteReturnValue!	Use the value specified in the ReturnValue argument instead of the value returned by the OLE server (if any) and ignore the error condition. You can set up an acceptable return value in an instance variable before you address the OLE server and assign it to the ReturnValue argument in the event script. The datatype of ReturnValue is Any, which accommodates all possible datatypes. With a valid substitute value, this choice is a safe one if you want to continue the application after the error occurs.

Example: ExternalException event

The ExternalException event, like the ocx_error event, provides error information from the OLE server that can be useful when you are debugging your application.

Suppose your window has two instance variables: one for specifying the exception action and another of type Any for storing a potential substitute value. Before accessing the OLE property, a script sets the instance variables to appropriate values:

```
ie_action = ExceptionSubstituteReturnValue!
ia_substitute = 0
li_currentsetting = ole_1.Object.Value
```

If the command fails, a script for the ExternalException event displays the Help topic named by the OLE server, if any. It substitutes the return value you prepared and returns. The assignment of the substitute value to *li_currentsetting* works correctly because their datatypes are compatible:

```
string ls_context

// Command line switch for WinHelp numeric context ID
ls_context = "-n " + String(helpcontext)
IF Len(HelpFile) > 0 THEN
   Run("winhelp.exe " + ls_context + " " + HelpFile)
END IF

Action = ExceptionSubstituteReturnValue!
ReturnValue = ia_substitute
```

Because the event script must serve for every automation command for the control, you would need to set the instance variables to appropriate values before each automation command.

Error event

The Error event provides information about the PowerBuilder context for the error. You can find out the PowerBuilder error number and message, as well as the object, script, and line number of the error. This information is useful when debugging your application.

The same principles discussed in the ExceptionAction value table for setting the Action and ReturnValue arguments apply to the Error event, as well as ExternalException.

For more information about the events for error handling, see the *PowerScript Reference*.

Creating hot links

Some OLE servers support property change notifications. This means that when a property is about to be changed and again after it has been changed, the server notifies the client, passing information about the change. These messages trigger the events PropertyRequestEdit and PropertyChanged.

PropertyRequestEdit event

When a property is about to change, PowerBuilder triggers the PropertyRequestEdit event. In that event's script you can:

- Find out the name of the property being changed by looking at the PropertyName argument.
- Obtain the old property value and save it

  The property still has its old value, so you can use the standard syntax to access the value.
- Cancel the change by changing the value of the CancelChange argument to TRUE

PropertyChanged event

When a property has changed, PowerBuilder triggers the PropertyChanged event. In that event's script, you can:

- Find out the name of the property being changed by looking at the PropertyName argument
- Obtain the new property value

  The value has already changed, so you *cannot* cancel the change.

Using the PropertyName argument

Because the PropertyName argument is a string, you *cannot* use it in dot notation to get the value of the property:

```
value = This.Object.PropertyName // Will not work
```

Instead, use CHOOSE CASE or IF statements for the property names that need special handling.

For example, in the PropertyChanged event, this code checks for three specific properties and gets their new value when they are the property that changed. The value is assigned to a variable of the appropriate datatype:

```
integer li_index, li_minvalue
long ll_color

CHOOSE CASE Lower(PropertyName)
   CASE "value"
   li_index = ole_1.Object.Value
   CASE "minvalue"
   li_minvalue = ole_1.Object.MinValue
```

```
        CASE "backgroundcolor"
        ll_color = ole_1.Object.BackgroundColor
        CASE ELSE
        ... // Other processing
    END CHOOSE
```

If a larger change occurred

In some cases the value of the PropertyName argument is an empty string (""). This means a more general change has occurred—for example, a change that affects several properties.

If notification is not supported

If the OLE server does not support property change notification, then the PropertyRequestEdit and PropertyChanged events are never triggered, and scripts you write for them will not run. Check your OLE server documentation to see if notification is supported.

If notifications are not supported and your application needs to know about a new property value, you might write your own function that checks the property periodically.

For more information about the PropertyRequestEdit and PropertyChanged events, see the *PowerScript Reference*.

Setting the language for OLE objects and controls

When you write automation commands, you generally use commands that match the locale for your computer. If your locale and your users' locale will differ, you can specify the language you have used for automation with the SetAutomationLocale function.

You can call SetAutomationLocale for OLE controls, custom controls, and OLEObjects, and you can specify a different locale for each automation object in your application.

For example, if you are developing your application in Germany and will deploy it all over Europe, you can specify the automation language is German. Use this syntax for an OLE control called ole_1:

```
ole_1.Object.SetAutomationLocale(LanguageGerman!)
```

Use this syntax for an OLEObject called oleobj_report:

```
oleobj_report.SetAutomationlocale(LanguageGerman!)
```

The users of your application must have the German automation interface for the OLE server application.

What languages do your users' computers support?
When your users install an OLE server application (particularly an OLE application from Microsoft), they get an automation interface in their native language and in English. It might not be appropriate for you to write automation commands in your native language if your users have a different language.

For more information, see the SetAutomationLocale function in the *PowerScript Reference*.

Low-level access to the OLE object

If you need low-level access to OLE through a C or C++ DLL that you call from PowerBuilder, you can use these functions:

- GetNativePointer (for OLEControl and OLECustomControl)
- GetAutomationNativePointer (for OLEObject)

When you have finished, you must use these functions to free the pointer:

- ReleaseNativePointer (for OLEControl and OLECustomControl)
- ReleaseAutomationNativePointer (for OLEObject)

For more information, see the *PowerScript Reference*.

OLE objects in DataWindow objects

The preceding sections discuss the automation interface to OLE controls and OLE objects. You can also use scripts to change settings for an OLE object embedded in a DataWindow object, and you can address properties of the external OLE object.

This section describes how to use the Object property in dot notation to set DataWindow properties and issue automation commands for OLE objects in DataWindow objects.

Naming the OLE object

To use dot notation for the OLE object, give the object a name. You specify the name on the General page in the object's property sheet.

Setting properties

You set properties of the OLE container object just as you do for any object in the DataWindow object. The Object property of the control is an interface to the objects within the DataWindow object.

For example, this statement sets the Pointer property of the object ole_word:

```
dw_1.Object.ole_word.Pointer = "Cross!"
```

It is important to remember that the compiler does not check syntax after the Object property. Incorrect property references cause runtime errors.

For more information about setting properties, handling errors, and the list of properties for the OLE DWObject, see the *DataWindow Reference*.

OLE objects and the Modify function
You cannot create an OLE object in a DataWindow object dynamically using the CREATE keyword of the Modify function. The binary data for the OLE object is not compatible with Modify syntax.

Functions and properties

There are four functions you can call for the OLE DWObject. They have the same effect as for the OLE control. They are:

- Activate
- Copy
- DoVerb
- UpdateLinksDialog

To call the functions, you use the Object property of the DataWindow control, just as you do for DataWindow object properties:

```
dw_1.Object.ole_word.Activate(InPlace!)
```

Four properties that apply to OLE controls in a window also apply to the OLE DWObject.

Table 19-7: Properties that apply to OLE controls and DWObject

Property	datatype	Description
ClassLongName	String	(Read-only) The long name for the server application associated with the OLE DWObject.
ClassShortName	String	(Read-only) The short name for the server application associated with the OLE DWObject.

Property	datatype	Description
LinkItem	String	(Read-only) The entire link name of the item to which the object is linked. For example, if the object is linked to *C:\FILENAME.XLS!A1:B2*, then LinkItem would contain *C:\FILENAME.XLS!A1:B2*.
ObjectData	Blob	If the object is embedded, the object itself is stored as a blob in the ObjectData property. If the object is linked, this property contains the link information and the cached image (for display).

Automation

You can send commands to the OLE server using dot notation. The syntax involves two Object properties:

- **The Object property of the DataWindow control.** Gives you access to DataWindow objects, including the OLE container DWObject
- **The Object property of the OLE DWObject.** Gives you access to the automation object

The syntax is:

dwcontrol.Object.*oledwobject*.Object.{ *serverqualifiers*. }*serverinstruction*

For example, this statement uses the WordBasic Insert function to add a report title to the beginning of the table of data in the Word document:

```
dw_1.Object.ole_word.Object.application.wordbasic.&
   Insert("Report Title " + String(Today()))
```

OLE columns in an application

OLE columns in a DataWindow object enable you to store, retrieve, and modify blob data in a database. To use an OLE column in an application, place a DataWindow control in a window and associate it with the DataWindow object.

For users of SQL Server

If you are using a SQL Server database, you must turn off transaction processing to use OLE. In the Transaction object used by the DataWindow control, set AutoCommit to TRUE.

For how to create an OLE column in a DataWindow object, see the PowerBuilder *User's Guide*.

Activating an OLE server application

Users can interact with the blob exactly as you did in preview in the DataWindow painter: they can double-click a blob to invoke the server application, then view and edit the blob. You can also use the OLEActivate function in a script to invoke the server application. Calling OLEActivate simulates double-clicking a specified blob.

The OLEActivate function has this syntax:

dwcontrol.**OLEActivate** (*row*, *columnnameornumber*, *verb*)

Specifying the verb

When using OLEActivate, you need to know the action to pass to the OLE server application. (Windows calls these actions verbs.) Typically, you want to edit the document, which for most servers means you specify 0 as the verb.

To obtain the verbs supported by a particular OLE server application, use the advanced interface of the Windows Registry Editor utility (run REGEDT32 /V).

For information about Registry Editor, see the Windows online Help file *REGEDT32.HLP*.

Example

For example, you might want to use OLEActivate in a Clicked script for a button to allow users to use OLE without their having to know they can double-click the blob's icon.

The following statement invokes the OLE server application for the OLE column in the current row of the DataWindow control dw_1 (assuming that the second column in the DataWindow object is an OLE column):

```
dw_1.OLEActivate(dw_1.GetRow(), 2, 0)
```

For more information

For more information about using OLE in a DataWindow object, see the PowerBuilder *User's Guide*.

OLE information in the Browser

The system stores information about the OLE server applications and OLE custom controls installed on your computer in the registry.

PowerBuilder reads the registry and displays the registration information for all registered OLE servers and custom controls.

❖ **To view the OLE information:**

1 Click the Browser button on the PowerBar.

2 Click the OLE tab in the Browser.

There are three categories of OLE object, as shown in Table 19-8.

Table 19-8: OLE object categories

OLE object category	Description
Insertable objects	OLE servers that can link or embed objects in OLE containers. OLE servers that support insertable objects must have a visual component.
Custom controls	ActiveX controls that can be included in an OLE container. ActiveX controls can also be insertable objects. If so, they will appear on both lists.
Programmable objects	OLE servers to which you can send automation instructions. A programmable object might not have a visual aspect, which means it supports only automation and cannot support insertable objects.

When you expand each of these categories, you see the individual OLE servers that are installed. Each OLE server can also be expanded. The information provided depends on the category.

Class information

All the categories provide class information about the OLE server. You see a list of registry keys. Some of the keys are meaningful in their own right and some have values. The values, or simply the presence or absence of keys, tell you how to find the OLE server and what it supports.

Table 19-9 lists some typical keys and what they mean.

Table 19-9: OLE registry keys

Registry key	Value
GUID	The global unique identifier for the OLE server.
TypeLib - GUID	The global unique identifier for the type library for an ActiveX control.
ProgID	A string that identifies the OLE server or ActiveX control. It usually includes a version number.
VersionIndependentProgID	A string that identifies the OLE server or ActiveX control, but does not include a version number.
InprocServer32	The name of the file for the 32-bit version of an ActiveX control.
ToolboxBitmap32	The name of a bitmap file for the 32-bit ActiveX control that can be used to represent the ActiveX control in toolbars or toolboxes of a development environment.
DefaultIcon	The name of an icon file or executable containing an icon to be used for an insertable icon that is being displayed as an icon.

Registry key	Value
Version	The version number of the OLE server or ActiveX control.
Insertable	No value – specifies that the entry is an OLE server that supports insertable object.
Control	No value – specifies that the entry is an ActiveX control.
Verb	No value – specifies that the entry accepts verbs as commands.

In addition to registry information, the Browser displays the properties and methods of ActiveX controls and programmable objects. To provide the information, PowerBuilder uses the registry information to query the ActiveX control for its properties and methods. The information includes arguments and datatypes.

Browser as script-writing tool

Take advantage of the Browser when writing scripts. You can find property and function names and paste them into your scripts. The Browser provides the full syntax for accessing that property.

❖ **To paste OLE information into a script:**

1. Open the Browser.
2. Click the OLE tab.
3. Expand the list to find what you want. For example, find the ActiveX control you want and expand the list further to find a property.
4. Highlight the property and select Copy from the pop-up menu.
5. Position the insertion point in the Script view and select Paste from the pop-up menu.

   The Browser inserts syntax like this into your script:

   ```
   OLECustomControl.Object.NeedlePosition
   ```

   After you change OLECustomControl to the actual name of your control, your script correctly accesses the NeedlePosition property.

What the Browser pastes into your script depends on what you have selected. If you select an object (a level above its properties in the hierarchy), PowerBuilder pastes the object’s ProgID. You can use the ProgID in the ConnectToNewObject function.

For more information about automation and the registry, see Chapter 20, “PowerBuilder Runtime Automation Server.”

Advanced ways to manipulate OLE objects

In addition to OLE objects in controls and objects for automation, PowerBuilder provides an interface to the underpinnings of OLE data storage.

OLE data is stored in objects called **streams**, which live in objects called **storages**. Streams and storages are analogous to the files and directories of a file system. By opening, reading, writing, saving, and deleting streams and storages, you can create, combine, and delete your OLE objects. PowerBuilder provides access to storages and streams with the OLEStorage and OLEStream object types.

When you define OLE controls and OLEObject variables, you have full access to the functionality of server applications and automation, which already provide you with much of OLE's power. You might never need to use PowerBuilder's storage and stream objects unless you want to construct complex combinations of stored data.

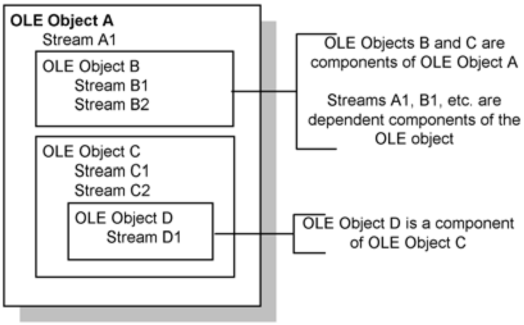
Storage files from other applications

This section discusses OLE storage files that a PowerBuilder application has built. Other PowerBuilder applications will be able to open the objects in a storage file built by PowerBuilder. Although Excel, Word, and other server applications store their native data in OLE storages, these files have their own special formats, and it is not advisable to open them directly as storage files. Instead, you should always insert them in a control (InsertFile) or connect to them for automation (ConnectToObject).

Structure of an OLE storage

An OLE storage is a repository of OLE data. A storage is like the directory structure on a disk. It can be an OLE object and can contain other OLE objects, each contained within the storage, or within a substorage within the storage. The substorages can be separate OLE objects—unrelated pieces like the files in a directory—or they can form a larger OLE object, such as a document that includes pictures as shown in Figure 19-3.

Figure 19-3: OLE storage structure


OLE Object A
Stream A1
OLE Object B
Stream B1
Stream B2
OLE Object C
Stream C1
Stream C2
OLE Object D
Stream D1
OLE Objects B and C are components of OLE Object A
Streams A1, B1, etc. are dependent components of the OLE object
OLE Object D is a component of OLE Object C

A storage or substorage that contains an OLE object has identifying information that tags it as belonging to a particular server application. Below that level, the individual parts should be manipulated only by that server application. You can open a storage that is a server's object to extract an object within the storage, but you should not change the storage.

A storage that is an OLE object has presentation information for the object. OLE does not need to start the server in order to display the object, because a rendering is part of the storage.

A storage might not contain an OLE object—it might exist simply to contain other storages. In this case, you cannot open the storage in a control (because there would be no object to insert).

Object types for storages and streams

PowerBuilder has two object types that are the equivalent of the storages and streams stored in OLE files. They are:

- OLEStorage
- OLEStream

These objects are class user objects, like a Transaction or Message object. You declare a variable, instantiate it, and open the storage. When you are through with the storage, you close it and destroy the variable, releasing the OLE server and the memory allocated for the variable.

Opening a storage associates an OLEStorage variable with a file on disk, which can be a temporary file for the current session or an existing file that already contains an OLE object. If the file does not exist, PowerBuilder creates it.

You can put OLE objects in a storage with the SaveAs function. You can establish a connection between an OLE control in a window and a storage by calling the Open function for the OLE control.

A stream is not an OLE object and cannot be opened in a control. However, streams allow you to put your own information in a storage file. You can open a stream within a storage or substorage and read and write data to the stream, just as you might to a file.

Performance tip
Storages provide an efficient means of displaying OLE data. When you insert a file created by a server application into a control, OLE has to start the server application to display the object. When you open an object in an OLE storage, there is no overhead for starting the server—OLE uses the stored presentation information to display the object. There is no need to start the server if the user never activates the object.

Opening and saving storages

PowerBuilder provides several functions for managing storages. The most important are Open, Save, and SaveAs.

Using the Open function

When you want to access OLE data in a file, call the Open function. Depending on the structure of the storage file, you might need to call Open more than once.

This code opens the root storage in the file into the control. For this syntax of Open, the root storage must be an OLE object, rather than a container that only holds other storages. (Always check the return code to see if an OLE function succeeded.)

```
result = ole_1.Open("MYFILE.OLE")
```

If you want to open a substorage in the file into the control, you have to call Open twice: once to open the file into an OLEStorage variable, and a second time to open the substorage into the control. *stg_data* is an OLEStorage variable that has been declared and instantiated using CREATE:

```
result = stg_data.Open("MYFILE.OLE")
result = ole_1.Open(stg_data, "mysubstorage")
```

Using the Save function

If the user activates the object in the control and edits it, then the server saves changes to the data in memory and sends a DataChange event to your PowerBuilder application. Then your application needs to call Save to make the changes in the storage file:

```
result = ole_1.Save()
IF result = 0 THEN result = stg_data.Save()
```

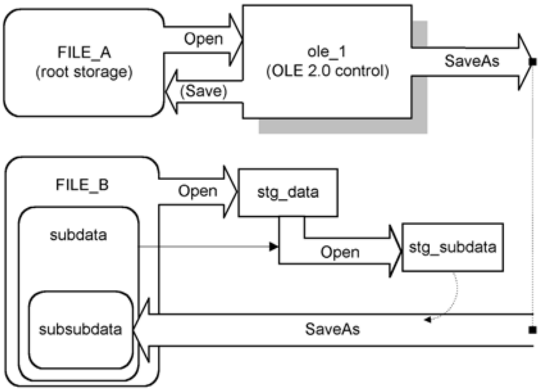
Using the SaveAs function

You can save an object in a control to another storage variable or file with the SaveAs function. The following code opens a storage file into a control, then opens another storage file, opens a substorage within that file, and saves the original object in the control as a substorage nested at a third level:

```
OLEStorage stg_data, stg_subdata
stg_data = CREATE OLEStorage
stg_subdata = CREATE OLEStorage
ole_1.Open("FILE_A.OLE")
stg_data.Open("FILE_B.OLE")
stg_subdata.Open("subdata", stgReadWrite!, &
   stgExclusive!, stg_data)
ole_1.SaveAs(stg_subdata, "subsubdata")
```

The diagram illustrates how to open the nested storages so that you can perform the SaveAs. If any of the files or storages do not exist, Open and SaveAs create them. Note that if you call Save for the control before you call SaveAs, the control's object is saved in *FILE_A*. After calling SaveAs, subsequent calls to Save save the object in subsubdata in *FILE_B*.

Figure 19-4: Nested OLE storages



The following example shows a simpler way to create a sublevel without creating a storage at the third level. You do not need to nest storages at the third level, nor do you need to open the substorage to save to it:

```
OLEStorage stg_data, stg_subdata
```

```
stg_data = CREATE OLEStorage
stg_subdata = CREATE OLEStorage
ole_1.Open("FILE_A.OLE")
stg_data.Open("FILE_B.OLE")
ole_1.SaveAs(stg_data, "subdata")
```

Getting information about storage members

When a storage is open, you can use one of the Member functions to get information about the substorages and streams in that storage and change them.

Table 19-10: OLE storage Member functions

Function	Result
MemberExists	Checks to see if the specified member exists in a storage. Members can be either storages or streams. Names of members must be unique—you cannot have a storage and a stream with the same name. A member can exist but be empty.
MemberDelete	Deletes a member from a storage.
MemberRename	Renames a member in a storage.

This code checks whether the storage subdata exists in *stg_data* before it opens it. (The code assumes that *stg_data* and *stg_subdata* have been declared and instantiated.)

```
boolean lb_exists
result = stg_data.MemberExists("subdata", lb_exists)
IF result = 0 AND lb_exists THEN
   result = stg_subdata.Open(stg_data, "subdata")
END IF
```

To use MemberExists with the storage member IOle10Native, use the following construction:

```
ole_storage.memberexists(char(1) + 'Ole10Native', &
   lb_boolean)
```

The `char(1)` is required because the “I” in IOle10Native is not an I, as you see if you look at the storage with a utility such as Microsoft's DocFile Viewer.

You need to use a similar construction to open the stream. For example:

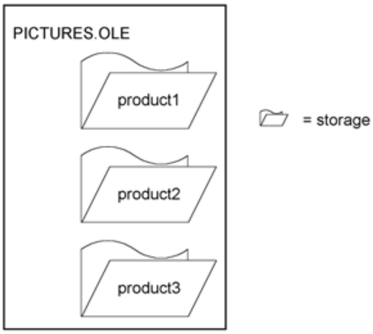
```
ole_stream.open(ole_storage, char(1) + 'Ole10Native', &
   StgReadWrite!, StgExclusive!)
```

Example: building a storage

Suppose you have several drawings of products and you want to display the appropriate image for each product record in a DataWindow object. The database record has an identifier for its drawing. In an application, you could call InsertFile using the identifier as the file name. However, calling the server application to display the picture is relatively slow.

Instead you could create a storage file that holds all the drawings, as shown in the diagram. Your application could open the appropriate substorage when you want to display an image.

Figure 19-5: OLE storage file



The advantage of using a storage file like this one (as opposed to inserting files from the server application into the control) is both speed and the convenience of having all the pictures in a single file. Opening the pictures from a storage file is fast, because a single file is open and the server application does not need to start up to display each picture.

OLE objects in the storage
Although this example illustrates a storage file that holds drawings only, the storages in a file do not have to belong to the same server application. Your storage file can include objects from any OLE server application, according to your application's needs.

This example is a utility application for building the storage file. The utility application is a single window that includes a DataWindow object and an OLE control.

The DataWindow object, called dw_prodid, has a single column of product identifiers. You should set up the database table so that the identifiers correspond to the file names of the product drawings. The OLE control, called ole_product, displays the drawings.

List of scripts for the example

The example has three main scripts:

- The window's Open event script instantiates the storage variable, opens the storage file, and retrieves data for the DataWindow object. (Note that the application's Open event connects to the database.)
- The RowFocusChanged event of the DataWindow object opens the drawing and saves it in the storage file.
- The window's Close event script saves the storage file and destroys the variable.

Add controls to the window

First, add the dw_prodid and ole_product controls to the window.

Application Open event script

In the application's Open event, connect to the database and open the window.

Instance variable

Declare an OLEStorage variable as an instance variable of the window:

```
OLEStorage stg_prod_pic
```

Window Open event script

The following code in the window's Open event instantiates an OLEStorage variable and opens the file *PICTURES.OLE* in that variable:

```
integer result
stg_prod_pic = CREATE OLEStorage
result = stg_prod_pic.Open("PICTURES.OLE")
dw_prod.SetTransObject(SQLCA)
dw_prod.Retrieve()
```

Retrieve triggers the RowFocusChanged event

It is important that the code for creating the storage variable and opening the storage file comes before Retrieve. Retrieve triggers the RowFocusChanged event, and the RowFocusChanged event refers to the OLEStorage variable, so the storage must be open before you call Retrieve.

RowFocusChanged event script

The InsertFile function displays the drawing in the OLE control. This code in the RowFocusChanged event gets an identifier from the prod_id column in a DataWindow object and uses that to build the drawing's file name before calling InsertFile. The code then saves the displayed drawing in the storage:

```
integer result
string prodid
//Get the product identifier from the DataWindow.
prodid = this.Object.prod_id[currentrow]

// Use the id to build the file name. Insert the
// file's object in the control.
```

```
result = ole_product.InsertFile( &
   GetCurrentDirectory() + "\" + prodid + ".gif")

// Save the OLE object to the storage. Use the
// same identifier to name the storage.
result = ole_product.SaveAs( stg_prod_pic, prodid)
```

Close event script

This code in the window's Close event saves the storage, releases the OLE storage from the server, and releases the memory used by the OLEStorage variable:

```
integer result
result = stg_prod_pic.Save()
DESTROY stg_prod_pic
```

Check the return values
Be sure to check the return values when calling OLE functions. Otherwise, your application will not know if the operation succeeded. The sample code returns if a function fails, but you can display a diagnostic message instead.

Running the utility application

After you have set up the database table with the identifiers of the product pictures and created a drawing for each product identifier, run the application. As you scroll through the DataWindow object, the application opens each file and saves the OLE object in the storage.

Using the storage file

To use the images in an application, you can include the prod_id column in a DataWindow object and use the identifier to open the storage within the *PICTURES.OLE* file. The following code displays the drawing for the current row in the OLE control ole_product (typically, this code would be divided between several events, as it was in the sample utility application above):

```
OLEStorage stg_prod_pic
//Instantiate the storage variable and open the file
stg_prod_pic = CREATE OLEStorage
result = stg_prod_pic.Open("PICTURES.OLE")

// Get the storage name from the DataWindow
// This assumes it has been added to the DataWindow's
// rowfocuschanging event
prodid = this.Object.prod_id[newrow]


//Open the picture into the control
result = ole_product.Open( stg_prod_pic, prodid )
```

The application would also include code to close the open storages and destroy the storage variable.

Opening streams

Streams contain the raw data of an OLE object. You would not want to alter a stream created by a server application. However, you can add your own streams to storage files. These streams can store information about the storages. You can write streams that provide labels for each storage or write a stream that lists the members of the storage.

To access a stream in an OLE storage file, you define a stream variable and instantiate it. Then you open a stream from a storage that has already been opened. Opening a stream establishes a connection between the stream variable and the stream data within a storage.

The following code declares and creates OLEStorage and OLEStream variables, opens the storage, and then opens the stream:

```
integer result
OLEStorage stg_pic
OLEStream stm_pic_label
/***************************************************
Allocate memory for the storage and stream variables
***************************************************/
stg_pic = CREATE OLEStorage
stm_pic_label = CREATE OLEStream
/***************************************************
Open the storage and check the return value
***************************************************/
result = stg_prod_pic.Open("picfile.ole")
IF result <> 0 THEN RETURN
/***************************************************
Open the stream and check the return value
***************************************************/
result = stm_pic_label.Open(stg_prod_pic, &
   "pic_label", stgReadWrite!)
IF result <> 0 THEN RETURN
```

PowerBuilder has several stream functions for opening and closing a stream and for reading and writing information to and from the stream.

Table 19-11: Stream functions

Function	Result
Open	Opens a stream into the specified OLEStream variable. You must have already opened the storage that contains the stream.
Length	Obtains the length of the stream in bytes.
Seek	Positions the read/write pointer within the stream. The next read or write operation takes place at the pointer.

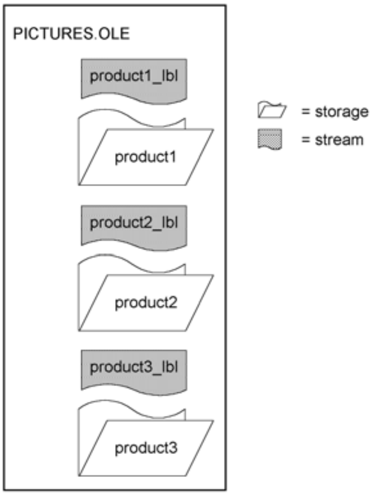
Function	Result
Read	Reads data from the stream beginning at the read/write pointer.
Write	Writes data to the stream beginning at the read/write pointer. If the pointer is not at the end, Write overwrites existing data. If the data being written is longer than the current length of the stream, the stream's length is extended.
Close	Closes the stream, breaking the connection between it and the OLEStream variable.

Example: writing and reading streams

This example displays a picture of a product in the OLE control ole_product when the DataWindow object dw_product displays that product's inventory data. It uses the file constructed with the utility application described in the earlier example (see "Example: building a storage" on page 371). The pictures are stored in an OLE storage file, and the name of each picture's storage is also the product identifier in a database table. This example adds label information for each picture, stored in streams whose names are the product ID plus the suffix _lbl.

Figure 19-6 shows the structure of the file.

Figure 19-6: OLE storage file structure



The example has three scripts:

- The window's Open event script opens the storage file and retrieves data for the DataWindow object. (Note that the application's Open event connects to the database.)

- The RowFocusChanged event of the DataWindow object displays the picture. It also opens a stream with a label for the picture and displays that label in a StaticText. The name of the stream is the product identifier plus the suffix _lbl.

  If the label is empty (its length is zero), the script writes a label. To keep things simple, the data being written is the same as the stream name. (Of course, you would probably write the labels when you build the file and read them when you display it. For the sake of illustration, reading and writing the stream are both shown here.)
- The window's Close event script saves the storage file and destroys the variable.

The OLEStorage variable *stg_prod_pic* is an instance variable of the window:

```
OLEStorage stg_prod_pic
```

The script for the window's Open event is:

```
integer result
stg_prod_pic = CREATE OLEStorage
result = stg_prod_pic.Open( is_ole_file)
```

The script for the RowFocusChanged event of dw_prod is:

```
integer result
string prodid, labelid, ls_data
long ll_stmlength
OLEStream stm_pic_label
/***************************************************
Create the OLEStream variable.
***************************************************/
stm_pic_label = CREATE OLEStream
/***************************************************
Get the product id from the DataWindow.
***************************************************/
this.Object.prod_id[currentrow]
/***************************************************
Open the picture in the storage file into the
control. The name of the storage is the product id.
***************************************************/
result = ole_prod.Open(stg_prod_pic, prodid)
IF result <> 0 THEN RETURN
/***************************************************
Construct the name of the product label stream and
open the stream.
***************************************************/
```

```
labelid = prodid + "_lbl"
result = stm_pic_label.Open( stg_prod_pic, &
   labelid, stgReadWrite! )
IF result <> 0 THEN RETURN
/****************************************************
Get the length of the stream. If there is data
(length > 0), read it. If not, write a label.
****************************************************/
result = stm_pic_label.Length(ll_stmlength)
IF ll_stmlength > 0 THEN
   result = stm_pic_label.Read(ls_data)
   IF result <> 0 THEN RETURN
   // Display the stream data in st_label
   st_label.Text = ls_data
ELSE
   result = stm_pic_label.Write( labelid )
   IF result < 0 THEN RETURN
   // Display the written data in st_label
   st_label.Text = labelid
END IF
/*****************************************************
Close the stream and release the variable's memory.
****************************************************/
result = stm_pic_label.Close()
DESTROY stm_pic_label
```

The script for the window’s Close event is:

```
integer result
result = stg_prod_pic.Save()
DESTROY stg_prod_pic
```

Strategies for using storages

Storing data in a storage is not like storing data in a database. A storage file does not enforce any particular data organization; you can organize each storage any way you want. You can design a hierarchical system with nested storages, or you can simply put several substorages at the root level of a storage file to keep them together for easy deployment and backup. The storages in a single file can be from the different OLE server applications.

If your DBMS does not support a blob datatype or if your database administrator does not want large blob objects in a database log, you can use storages as an alternative way of storing OLE data.

It is up to you to keep track of the structure of a storage. You can write a stream at the root level that lists the member names of the storages and streams in a storage file. You can also write streams that contain labels or database keys as a way of documenting the storage.

CHAPTER 20

PowerBuilder Runtime Automation Server

About this chapter

This chapter describes how to access PowerBuilder class user objects by means of automation. A PowerBuilder object is the OLE server; PowerBuilder or any other OLE-enabled development tool can be the client application, accessing the methods and properties of the PowerBuilder object.

Contents

Using the runtime automation server

Use PowerBuilder COM servers

This chapter focuses on the use of the PowerBuilder runtime automation server. PowerBuilder COM/COM+ server generation is the preferred technique for building COM-compliant servers. The PowerBuilder automation server technology might be discontinued in a future release.

For information about generating PowerBuilder COM objects from custom class user objects, see Chapter 27, "Building a COM or COM+ Component."

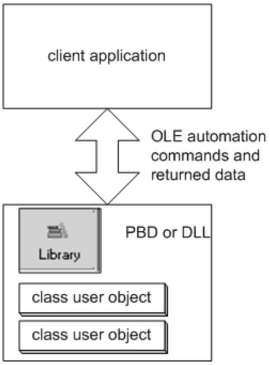
Chapter 19, "Using OLE in an Application," explains how PowerBuilder provides containers for OLE objects and custom controls and how you can use automation to program those objects. The functionality for those objects is provided by server applications. The programming tells a server application how to manipulate its data.

What the automation server is

The PowerBuilder automation server is an OLE server for *programmable objects*—rather than insertable, visible objects. It provides access to class user objects (nonvisual user objects) that you have defined in a PowerBuilder library. You can initiate a server session, create one or more objects, and send commands to those objects using automation syntax.

The class user object can create instances of other objects, and the server can pass references to these objects to the client.

Figure 20-1: Server passes class user object references to client



Client applications

Any client application that supports automation and programmable objects can access the PowerBuilder automation server. You can create your client application in any COM-compliant tool such as PowerBuilder, Visual C++, or Visual Basic.

Runtime overhead

Each time you connect to a server, you invoke an instance of the PowerBuilder runtime environment. However, this is not a huge penalty—each runtime shares the definition of the system classes in the client's runtime session.

User class definitions are not shared. Therefore, if you create the same large objects in each of several runtime sessions, memory usage for an object is multiplied by the number of sessions in which you create it. You could make your application more memory-efficient by reorganizing the application and creating the objects in a single session.

Uses for runtime automation

Any user object that encapsulates functionality and provides information can be a useful automation server. The information you want to access must be stored in public instance variables or available as return values or reference arguments of functions.

Examples of an automation server include:

- A user object that creates a DataStore and includes functions that report statistics on the retrieved data
- A user object that defines business rules, such as functions that validate data

Three methods

There are three ways to access PowerBuilder user objects. You can access:

- A PowerBuilder class user object that you have defined and registered
- PowerBuilder itself as the server, after which you can create any object whose definition is in the libraries you have specified
- A named server that allows you to use a name appropriate for your business while giving you access to the PowerBuilder server

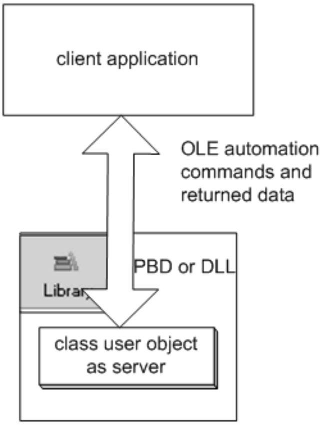
A user object as automation server

You can define a class user object and register it in the registry. When you use your client's functions for accessing an external object, you have access to the user object's properties and functions. (Automation accesses the object's instance variables as properties.)

The advantages of a registered user object as an automation server include:

- Browsable information about the object in the registry, making it easy for others to program the object
- Simpler client code for accessing the object
- Restriction of client access to classes you publish in the registry, return from function calls, or declare as instance variables of the user object

Figure 20-2: Interaction of class user object as server with client


client application
OLE automation commands and returned data
PBD or DLL
Library
class user object as server

For you to access an independent user object, it must be installed as an independent entity in the registry. You connect to it by means of an OLEObject variable and use automation to access it. The object invokes a PowerBuilder runtime session to support it.

When you access the object in your runtime session, you create a single instance of the object, and PowerBuilder invokes a runtime session to support it. For each registered object you create, you incur the overhead of additional runtime sessions.

The steps you take to register and use a user object are described in “Using a user object as an automation server” on page 384.

PowerBuilder as automation server

When you install PowerBuilder, an entry is added to the registry for PowerBuilder.Application, a general-purpose PowerBuilder automation server. You can create instances of any number of class user objects and access their properties and methods. (Automation accesses the object’s instance variables as properties.)

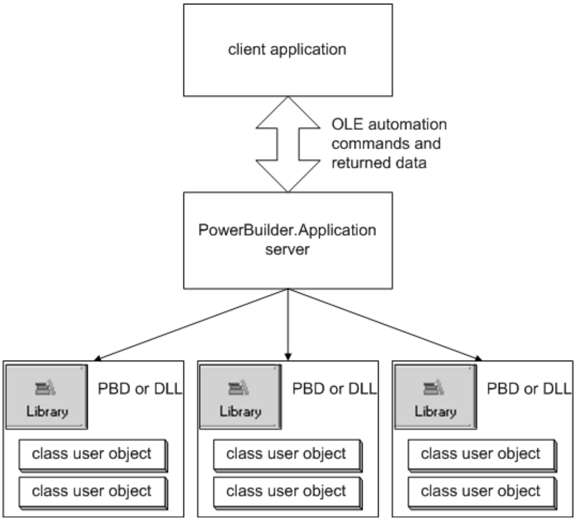
The advantages of using PowerBuilder.Application include:

- Access to more than one object without invoking the overhead of more than one PowerBuilder runtime environment.
- Direct access to all system and private classes available in the library list. These classes can be instantiated using the CreateObject function of PowerBuilder.Application.

When you connect to PowerBuilder.Application, you specify the libraries that you will access. You can instantiate any number of user objects whose definitions reside in those libraries, as well as system classes.

Each object you create in the client exists as an independent OLEObject in the client, and you can address each one using automation. If the client passes a server object reference back to another server object in the same runtime session, PowerBuilder recognizes the PowerBuilder datatype of the object. This allows the two objects to interact in the server session, rather than being limited to automation commands from the client.

Figure 20-3: Objects interacting in a server session



The steps you take to set up and use PowerBuilder.Application are described in “Using PowerBuilder as an automation server” on page 388.

Named automation server

For business reasons, you might want to avoid references to PowerBuilder.Application, but still have access to the additional functionality it provides—for example, the efficiency of instantiating more than one object in a server session.

You can create an entry in the registry that serves as a second pointer to PowerBuilder.Application, allowing you to give the server a name appropriate for your business.

The steps involved in setting up a named server are described in “Creating and using a named server” on page 392.

Using a user object as an automation server

Accessing a registered user object as an automation server involves these steps:

1. Create the object you will access.
2. Build the object’s runtime library.
3. Register the object.
4. Write code in the client that connects to and uses the object.

Creating a class user object to be a server

How you define user objects for an automation server depends mainly on your application design. The object must be a custom class user object. You define instance variables and functions for the object. The object can declare and instantiate other objects for its own use.

No references to visual objects
The class user object you use as an automation server cannot contain any references to visual objects such as message boxes or windows.

Objects on the client

You can pass your object references to other objects in the client application. Those references are of type OLEObject, and your object can use automation syntax to access properties and methods of the object. (Automation accesses the object’s instance variables as properties.)

Testing the user object

Before you try accessing your user object as a server, test its design and functions in a simpler environment. You can build an application for testing the user object locally.

Building the object's runtime library

After you have defined your object, use the Library painter to build a dynamic library (*PBD*) or compiled library (*DLL*). The reasons for choosing either type (Pcode or compiled machine code) are the same as for building any PowerBuilder application.

For more information, see "Compiler basics" on page 750.

You might want to use the Library painter to reorganize libraries so that your object and any other objects it instantiates are in a library by themselves. Otherwise, your library will be bigger than it needs to be.

Here is a quick overview of what you need to do to build your library in the Library painter:

1. In the Library painter, select the library name.
2. Select Library>Build Runtime Library.
3. Check or clear the Machine Code check box to correspond to your decision about *PBD*s versus *DLL*s.

   Other options in the dialog box are not essential to this process. For information about them, click the Help button or see the PowerBuilder *User's Guide*.
4. Click OK to build the library.
5. Move the library to the directory you want it to be registered in.

Registering the object

To use your object, you have to register it in the registry. You can also create a type library that provides information to registry browser applications about your object's properties and functions.

For more information, see "More about user objects and the registry" on page 394.

The Automation Server project wizard makes registering and creating type libraries easy.

❖ **To create registry information and register your object:**

1. Select the Automation Server wizard from the Project tab in the New dialog box.

2 Complete all the pages in the wizard.

Use Table 20-1 to help you.

Table 20-1: Automation Server wizard pages

Page	What to specify
Select component	Select the object you want to use as an automation server. You can select only one object. For when to specify PowerBuilder.Application, see “Creating and using a named server” on page 392.
Specify Program Identifier	Specify an identifier for the object such as Mycompany.Myapp. Do not supply a version number. PowerBuilder constructs a version-dependent entry using the version number you specify on another screen. The identifier can contain up to 39 characters, must not contain any punctuation apart from the period between vendor and application, and must not start with a digit.
Specify Registry File and Object GUID	If the GUID text box is empty, click Generate to generate a new globally unique identifier (GUID). The new GUID will be the class identifier (CLSID) for your object. If you specified an existing programmatic identifier, the GUID will be filled in. You can create a new GUID if you do not want to reuse the existing one. For information about reusing GUIDs, see “Multiple versions and updates” on page 402. The registry update file is a text file containing information for updating the registry. Typically it has the same name as the library, and the extension *REG*.
Create Type Library File	You need to create a type library only if you want OLE browsers to display information about the properties and methods of your object.
Specify Build Options	Check the machine code *DLL* check box if you built a *DLL* instead of a *PBD* file.

3 Select File>Open from the menu bar and select the project created by the wizard, or double-click the Build Project item on the ToDo list.

4 Select Design>Build Project from the menu bar in the Project painter to generate the registry and type library files.

5 Run the registry file to add information to the registry.

For more information about the registry and writing your own registration tool, see “Creating registry information” on page 397.

Writing client code that accesses the user object

The client code for accessing a registered user object is simpler than the code for accessing PowerBuilder.Application. The library list and machine code properties are already determined and stored in the registry. All you need to do is connect to the object and use automation to access its properties and methods.

PowerBuilder as client

To establish a server automation session with your user object, you need code for these steps.

1 Declare an OLEObject variable for your user object:

```
OLEObject ole_analyze
```

2 Connect to your object using its programmatic identifier (ProgID) and check that the connection was established. A status of 0 indicates success:

```
ole_analyze = CREATE OLEObject
li_status = ole_analyze.ConnectToNewObject &
   ("MyCompany.Analyze")
IF li_status < 0 THEN
   MessageBox("No Server", &
      "Can't connect to MyCompany.Analyze.")
   RETURN
END IF
```

3 Access functions or properties of the object using automation syntax:

```
ld_avg = ole_analyze.uof_average()
ole_analyze.Projection = TRUE
li_status = ole_analyze.uof_RTFreport(REF ls_rpt)
```

If you want to handle errors in the OLEObject ExternalException and Error events, use a user object inherited from OLEObject instead of declaring an OLEObject variable:

1 Open the User Object painter and create a standard class user object inherited from OLEObject.

2 Write scripts for the Error and ExternalException events.

3 Use the name of the new class instead of OLEObject in the declaration:

```
uo_oleobject ole_analyze
```

Visual Basic as client

Similar code in Visual Basic connects to your registered object.

1 Declare an object variable for your user object:

```
Dim ole_analyze As Object
```

2. Connect to your object using its programmatic identifier (ProgID) and check that the connection was established:

   ```
   Set ole_analyze = CreateObject("MyCompany.Analyze")
   If ole_analyze Is Nothing Then
      REM Handle the error
   End If
   ```

3. Access functions or properties of the object using automation syntax:

   ```
   ld_avg = ole_analyze.uof_average()
   ole_analyze.Projection = TRUE
   li_status = ole_analyze.uof_RTFreport(REF ls_rpt)
   ```

Using PowerBuilder as an automation server

Using PowerBuilder.Application as an automation server involves these steps:

1. Define the objects you will access.
2. Build the runtime libraries for those objects.
3. Write code in the client that connects to PowerBuilder, creates the objects, and accesses their methods and properties.

Creating the user objects you will access

How you define user objects for an automation server depends mainly on your application design. The object must be a custom class user object. You define instance variables and functions for the object. The object can declare and instantiate other objects for its own use.

Objects created using automation

When the client instantiates multiple objects in the same server runtime session, you can pass references to those objects, enabling the objects to work together.

You can make a PowerBuilder object you create from the client aware of another by passing it that reference. When the objects exist in the same session, PowerBuilder accepts the OLE object reference and also recognizes the underlying PowerBuilder object datatype. This reference gives access to the first object’s properties and methods just as in any PowerBuilder application.

Keeping this technique in mind, you can define functions for the user object that accept object references passed from the client, assign them to the correct variable type, and treat the references as instantiated PowerBuilder objects, which they are.

Building runtime libraries

After you have defined your objects, use the Library painter to build PowerBuilder dynamic libraries (*PBD*s) or compiled libraries (*DLL*s). All the libraries accessed in the same PowerBuilder.Application session must be the same type. The reasons for choosing either type, Pcode or compiled machine code, are the same as for building any PowerBuilder application.

Here is a quick overview of what you need to do to build each library in the Library painter.

1. In the Library painter, select the library name.
2. Select Library>Build Runtime Library.
3. Check or clear the Machine Code check box to correspond to your decision about *PBD*s versus *DLL*s.

   Other options in the dialog box are not essential to this process. For information about them, click the Help button or see the PowerBuilder *User's Guide*.
4. Click OK to build the library.
5. Repeat steps 1 to 4 for each library specified in the LibraryList property of PowerBuilder.Application.
6. Put the resulting *PBD*s or *DLL*s in the desired directory. The client application will specify the paths so that the server can locate them.

For more information about building an application in the Project painter, see the PowerBuilder *User's Guide*.

Writing client code that accesses PowerBuilder and user objects

A client application that wants to establish a PowerBuilder.Application session needs code to:

- Connect to the server
- Set properties for the server
- Instantiate objects
- Access those objects

All the steps of the process should be accompanied by error checking because there are any number of reasons why server sessions can fail.

The following steps with code examples illustrate how to do it. The first set of steps shows client code for PowerBuilder. A Visual Basic example follows.

PowerBuilder as a client

To establish a server automation session with PowerBuilder.Application and access objects, you need code for each of these steps:

1 Declare one OLEObject variable for PowerBuilder.Application. Declare additional OLEObject variables for each object you will create.

```
OLEObject ole_pba, ole_analyze
```

If you want to handle errors in the OLEObject ExternalException and Error events, use a user object inherited from OLEObject instead.

2 Start the automation server and check that the connection was established. A status of 0 indicates success.

```
ole_pba = CREATE OLEObject
li_status = ole_pba.ConnectToNewObject &
   ("PowerBuilder.Application")
IF li_status < 0 THEN
   MessageBox("No Server", &
      "Can't connect to PowerBuilder.Application.")
   RETURN
END IF
```

3 Set the properties of PowerBuilder.Application, establishing the libraries you will access. You cannot change these property values after you create objects.

```
ole_pba.LibraryList = &
   "c:\pbobjs\myobj\serv1.dll;c:\pbobjs\myobj\serv2
.dll"
ole_pba.MachineCode = TRUE
```

4 Create the first object you want to use and check for success. Specify the object's name as defined in the library:

```
ole_analyze = ole_pba.CreateObject("uo_analyze")
IF IsNull(ole_analyze) THEN
   MessageBox("No Object", &
      "Can't create object uo_analyze.")
   RETURN
END IF
```

5 Access functions or properties of the object using automation syntax. (These properties and methods are hypothetical.)

```
ld_avg = ole_analyze.uof_average()
ole_analyze.Projection = TRUE
li_status = ole_analyze.uof_RTFReport(REF ls_rpt)
```

6 Disconnect from PowerBuilder.Application and destroy the objects when you are done. (Exiting your application also accomplishes this.)

```
DESTROY ole_analyze
ole_pba.DisconnectObject()
DESTROY ole_pba
```

Visual Basic version

This example shows typical Visual Basic client code for establishing a server automation session with PowerBuilder.Application. The steps are similar to those for PowerBuilder above.

1 Declare an object variable for the PowerBuilder.Application. Declare additional object variables for each object you will create.

```
Dim ole_pba As Object
Dim ole_analyze As Object
```

2 Start the automation server and check that the connection was established. A status of 0 indicates success.

```
Set ole_pba = CreateObject_
   ("PowerBuilder.Application")
If ole_pba Is Nothing Then
   REM Handle the error
End If
```

3 Set the properties of PowerBuilder.Application, establishing the libraries you will access. You cannot change these property values after you create objects.

```
ole_pba.LibraryList = _
   "c:\pb\myobj\serv1.dll;c:\pb\myobj\serv2.dll"
ole_pba.MachineCode = TRUE
```

4 Create the first object you want to use and check for success. You specify the object's name as defined in the library:

```
Set ole_analyze = ole_pba.CreateObject _
   ("uo_analyze")
If ole_analyze Is Nothing Then
   REM Handle the error
End If
```

5 Access functions or properties of the object using automation syntax. (These properties and methods are hypothetical.)

```
ld_avg = ole_analyze.uof_average()
ole_analyze.Projection = TRUE
li_status = ole_analyze.uof_RTFreport(REF ls_rpt)
```

6 Destroy the objects. (Exiting the application also accomplishes this.)

```
Set ole_analyze = Nothing
Set ole_pba = Nothing
```

For complete information about PowerBuilder.Application functions and properties, see "Runtime automation server reference" on page 402.

Creating and using a named server

Using your own named server involves these steps:

1 Define the objects you will access.

2 Build the runtime libraries for those objects.

3 Register your server in the registry.

4 Write code in the client that connects to your server, creates objects, and accesses their methods and properties.

Creating the user objects you will access

Defining user objects for use in a named server is the same as for PowerBuilder.Application.

For information, see "Creating the user objects you will access" on page 388.

Building runtime libraries

Building runtime libraries for use in a named server is the same as for PowerBuilder.Application.

For information, see "Building runtime libraries" on page 389.

Registering the server

You can use the Automation Server project wizard to register your server, as described in “Registering the object” on page 385.

To register your server rather than a single user object, select PowerBuilder.Application as the component. Proceed as described in “Registering the object” on page 385 and install the registry update file in the registry. You do not need to create a type library file for a named server.

Writing client code that accesses the server and user objects

A client application that wants to establish a session with your named server needs code to:

- Connect to the server
- Instantiate objects
- Access those objects

The following steps with code examples illustrate how to do it. The examples are client code for PowerBuilder. All the steps of the process should be accompanied by error checking because there are any number of reasons why server sessions can fail.

1 Declare one OLEObject variable for your server. Declare additional OLEObject variables for each object you will create:

```
OLEObject ole_server, ole_analyze
```

If you want to handle errors in the OLEObject ExternalException and Error events, use a user object inherited from OLEObject instead.

2 Start the automation server and check that the connection was established. A status of 0 indicates success:

```
ole_server = CREATE OLEObject
li_status = ole_server.ConnectToNewObject &
   ("MyCompany.MyServer")
IF li_status < 0 THEN
   MessageBox("No Server", &
      "Can't connect to the server.")
   RETURN
END IF
```

3 Create the first object you want to use and check for success. Specify the object’s name as defined in the library:

```
ole_analyze = &
   ole_server.CreateObject("uo_analyze")
```

```
IF IsNull(ole_analyze) THEN
   MessageBox("No Object", &
      "Cannot create object uo_analyze.")
   RETURN
END IF
```

4 Access functions or properties of the object using automation syntax. (These properties and methods are hypothetical.)

```
ld_avg = ole_analyze.uof_average()
ole_analyze.Projection = TRUE
li_status = ole_analyze.uof_RTFReport(REF ls_rpt)
```

5 Disconnect from the server and destroy the objects when you are done. (Exiting your application also accomplishes this.)

```
DESTROY ole_analyze
ole_server.DisconnectObject()
DESTROY ole_server
```

More about user objects and the registry

The registry stores information required by a program for it to access a registered object. The information is stored in the registration database. To update the registry, the Registry Editor reads registry update files (*REG*) and inserts the information in the database.

What are all the identifiers for?

GUIDs and CLSIDs

A globally unique identifier (GUID) is a 128-bit integer that is virtually guaranteed to be unique. You can generate your own using the PowerBuilder.Application GenerateGUID function. The formula is based on the time of day, the date, and a unique number contained in your network card. If you do not have a network card, you can request GUIDs from Microsoft in sets of 256.

GUIDs are used as class identifiers (CLSID) to uniquely identify each object and type library. The CLSID enables a program to be sure of the interface that it is interacting with.

ProgIDs

Programmatic identifiers (ProgID) are unique only in the local environment. They may contain up to 39 characters, must not contain any punctuation (except for the period), and must not start with a digit. They have two forms:

- **Version-independent** Does not include a version number

  applicationname.*objectname*

  For example, MyApp.Application or MyApp.AnalysisUserObject

- **Version-dependent** Includes a version number

  applicationname.objectname.versionnumber

  For example, MyApp.Application.1 or MyApp.AnalysisUserObject.1

You should establish consistent naming conventions for your applications and objects so that others can see the relationships among your objects.

Other programmers use the ProgID to connect to objects that you register. If they specify the version-independent ID, the registry redirects them to the current version.

When you use the PowerBuilder.Application function GenerateRegFile, you supply a version-independent ProgID and PowerBuilder constructs the version-dependent entry using the version number you specify.

Where information about your object is stored

Information about a registered object is stored in the registry in three or four places. There is a section for a:

- CLSID
- Version-dependent ProgID
- Version-independent ProgID
- Type Library (optional)

All this cross-referencing ensures that an object's information can be found no matter what the program starts with.

Keys and their subtrees

The PowerBuilder.Application functions update the tree *HKEY_CLASSES_ROOT*. The other keys in the registry reflect this information for a particular context.

Subkeys

Information for each entry is stored in subkeys, which are a level below the entry subkey. Some subkeys have values, such as a file name. Others have meaning just because they are there, such as NotInsertable.

Each entry and its associated subkeys is described below.

ProgID

The list of registered ProgIDs appears in the registry under this key:

```
My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\CLASSES
```

Both version-dependent and version-independent ProgIDs appear. The ProgID is a cross-reference to the object's CLSID.

Table 20-2: ProgID subkeys

Subkey	Value
\CLASSES\ProgID	Description of object
\CLASSES\ProgID\CLSID	The object's CLSID
\CLASSES\ProgID\CurVer (for the version-independent entry only)	The version-dependent ProgID
\CLASSES\ProgID\NotInsertable	Empty string

CLSID

Under the *CLASSES* key is a subkey *CLSID* that lists all the registered CLSIDs. Each CLSID records all the information about the registered object:

```
My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\CLASSES\CLSID
```

Table 20-3: CLSID subkey contents

Subkey	Value
\CLASSES\CLSID\{guid}	The GUID for the object.
\CLASSES\CLSID\{guid}\InProcServer32	The path and file name for the PowerBuilder runtime *DLL*.
\CLASSES\CLSID\{guid}\NotInsertable	""
\CLASSES\CLSID\{guid}\PowerBuilder	Not set. Subkeys store information about the object's environment (see below).
\CLASSES\CLSID\{guid}\Programmable	""
\CLASSES\CLSID\{guid}\ProgID	The ProgID you specified with the current version number.
\CLASSES\CLSID\{guid}\TypeLib	The CLSID for the type library.
\CLASSES\CLSID\{guid}\VersionIndependentProgID	The ProgID you specified without a version number.

Table 20-4: PowerBuilder subkey contents

PowerBuilder subkey under \CLASSES\CLSID	Value
{guid}\PowerBuilder\BinaryType	The type of code in the library, either compiled machine code (MCODE) or Pcode
{guid}\PowerBuilder\LibraryList	The name of the library that contains the object
{guid}\PowerBuilder\ClassName	The object's name in the library

TypeLib

Under the *CLASSES* key is another subkey for *TypeLib*.

```
My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\CLASSES
   TypeLib
```

Table 20-5: TypeLib subkey contents

Subkey	Value
\CLASSES\TypeLib\{guid}	The GUID for the type library.
\CLASSES\TypeLib\{guid}\VersionNum	The version number of the type library. A tree of subkeys is below the version number, allowing entries for type libraries in several languages. *VersionNum\LanguageID\Win32* specifies the path and name of type library file for 32-bit Windows operating systems.
\CLASSES\TypeLib\{guid}\FLAGS	0
\CLASSES\TypeLib\{guid}\HELPDIR	The directory where a Help file is located.

Creating registry information

A client application uses the registry to find the information required to access your user object. The registry includes information about where to find the files for the PowerBuilder runtime environment and the libraries. It can also contain a pointer to a type library file that documents the properties and methods of the object. When the type library is registered, it lets the user examine the object in an OLE browser. (Instance variables of the object appear as properties in the OLE browser.)

PowerBuilder.Application provides facilities for generating the necessary registry information. To add your object to the registry, you need a registry update file, which contains information to be added to the system's registration database. The type library is optional—you need it if you want others to be able to inspect your object's properties and methods in an OLE browser.

To use these functions, you must run a client application that connects to PowerBuilder.Application. (You can run the client application in the development environment.)

Functions for generating registry information include:

- **GenerateGUID** Gets a valid globally unique identifier to serve as the CLSID of your object and its type library
- **GenerateRegFile** Uses the properties of PowerBuilder.Application and other values to generate a registry update file that the Registry Editor (*REGEDT32.EXE*) can use
- **GenerateTypeLib** Uses the information in the object to generate a type library and updates the registry file with information about the type library

You can use these functions yourself, as shown in the example below, or you can use an installation application that provides tools for creating registry information, such as InstallShield.

Deploying and the registry

The registry file that you create on your computer can be used to register your object on your own computer. The path information for files (the object library and the type library) is specific to your machine and is probably not applicable to other machines. To deploy your object, you need to modify the registry update files according to the directories the user specifies for installation, or call functions for modifying the registry itself.

See "Deploying an application that uses the automation server" on page 401.

Sample script for generating registry information

This script takes information from several SingleLineEdit controls and a CheckBox and builds a registry update file and a type library. The script hard-codes version information, but you can use edit boxes to obtain version information too.

The SingleLineEdits and CheckBox used in the following example are shown in Table 20-6.

Table 20-6: SingleLineEdits and CheckBox examples

Object for input	Data provided
sle_object	The name of the object in the PowerBuilder library
sle_progid	A programmatic identifier you supply
sle_desc	A description of your object, which is displayed in the registry
sle_regfile	The name of the registry update file you want to generate
sle_typelibfile	The name of the type library file you want to generate
sle_library	The name of the PowerBuilder library containing the object
cbx_machinecode	Whether the library uses compiled code or Pcode

This program generates a registry update file and type library.

```
oleObject ole_pb
string ls_reg_guid, ls_tlb_guid
long ll_result, ll_res2

// Connect to the PowerBuilder.Application server
ole_pb = CREATE oleObject
ll_result = ole_pb.ConnectToNewObject &
   ("PowerBuilder.Application")
IF ll_result < 0 THEN
   MessageBox( "Can't Connect", &
      "Error connection to PowerBuilder.Application")
   RETURN
END IF

// Set properties for the server session, which
// will become the values for the registered object
ole_pb.LibraryList = sle_library.Text
ole_pb.MachineCode = cbx_machinecode.Checked

// Get GUIDs for the object and type library
ll_result = ole_pb.GenerateGUID( REF ls_reg_guid)
ll_res2 = ole_pb.GenerateGUID( REF ls_tlb_guid)


IF ll_result < 0 THEN
   MessageBox( "Can't Get GUID", &
      "Generating GUID for Reg file failed.")
   RETURN
ELSEIF ll_res2 < 0 THEN
   MessageBox( "Can't Get GUID", &
      "Generating GUID for TypeLib file failed.")
   RETURN
END IF
```

```
// Use info from user to generate registry update file
// Arguments for GenerateRegFile:
//    valid GUID
//    Name of uo in PB library
//    ProgID as in registry
//    Major version, Minor version
//    Description
//    Name of reg file
ll_result = ole_pb.GenerateRegFile( &
   ls_reg_guid, &
   sle_object.Text, &
   sle_progid.Text, &
   1, 0, &
   sle_desc.Text, &
   sle_RegFile.Text )


IF ll_result < 0 THEN
   MessageBox("Can't generate RegFile", &
      "result code is " + String(ll_result) )
   RETURN
END IF

// Use information that matches the registry update
// file to generate the type library
// Arguments for GenerateTypeLib:
//
//       GUID in Reg file
//       Name of uo in PB library
//       Prog ID as in Reg file
//       Locale, Major version, Minor version
//       Description
//       Help context, Help file
//       GUID for TypeLib
//       Name of typelib file
//       Name of reg file
ll_result = ole_pb.GenerateTypeLib( &
   ls_reg_guid, &
   sle_object.Text, &
   sle_progid.Text, &
   0, 1, 0, &
   sle_desc.Text, &
   0, "", &
   ls_tlb_guid, &
   sle_typelibfile.text, &
   sle_regfile.text)


IF ll_result < 0 THEN
   MessageBox("Can't generate TypeLib File", &
      "result code is " + String(ll_result) )
   RETURN
END IF
```

Deploying an application that uses the automation server

When deploying your objects, you need to coordinate registry entries with the locations of all the files.

PowerBuilder runtime files

For both PowerBuilder.Application and a registered user object as a server, you need to deploy the PowerBuilder runtime environment on each user's machine.

For information about PowerBuilder deployment and required files, see Part 9, "Deployment Techniques."

Instead of the PowerBuilder executable, the registry looks for the PowerBuilder virtual machine *DLL*, *PBVM110.DLL*, to start the runtime environment. When you deploy, the path for this file is recorded in the registry. Users should not move the file to another directory—the registry entry would be made invalid.

Object library and type library

The registry records the location of the library containing your object and the location of the type library.

Therefore, when you generate a registry update file on your own computer, path information reflects the file location on your machine. When you deploy, you can:

- Customize the registry update file (it is an editable text file)
- Make changes to the registry after it has been updated

  You do this programmatically using functions in PowerBuilder or the Windows SDK, or manually with the Registry Editor.

Multiple versions and updates

When you distribute a new version of an object, you can reuse the GUID that was the CLSID for the previous version, or you can assign a new GUID. Your choice should depend on the degree of compatibility between the versions.

Table 20-7: CLSID options

Option for the CLSID	Conditions
Reuse the GUID	The interface for the object is the same and existing applications can access the same properties and functions that were in the old version
Assign a new GUID	The interface has changed and existing applications will fail if they access the new version

If you assign a new GUID so that the old object remains available to existing applications, you need a new ProgID too. If the existing application refers to the version-independent ProgID and you use the same ProgID for the new version, the old application will connect to the new object anyway.

When you design your server object, you need to think about its future development and design current applications accordingly.

Runtime automation server reference

This section describes the PowerBuilder.Application runtime automation server and its properties and functions:

- PowerBuilder.Application server object
- CreateObject function
- GenerateGUID function
- GenerateRegFile function
- GenerateTypeLib function
- Exception codes
- Sample registry update file

PowerBuilder.Application server object

Description

PowerBuilder.Application is an automation server. The OLE client that starts the automation server can be PowerBuilder or some other client that supports automation and automation object types.

Accessing objects by means of the server You can access PowerBuilder objects using automation by:

1. Connecting an OLEObject (or the equivalent object type in other client applications) to PowerBuilder.Application, which starts PowerBuilder as a server application.
2. Setting properties of PowerBuilder.Application to specify the PowerBuilder libraries you want to access.
3. Calling functions that create class user objects (nonvisual) in those libraries and assigning those objects to additional OLEObject variables.
4. Accessing the properties and functions of the user objects using automation syntax.

Programmable object in the registry PowerBuilder.Application is not a class in the PowerBuilder system object hierarchy. It is a programmable object registered in the Windows registry. To see it in the Browser, click the OLE tab and expand the Programmable Objects category.

Properties

Table 20-8: Properties of PowerBuilder.Application

Property	Datatype	Description
LibraryList	String	A list of file names separated by semicolons that are the *DLL*s or *PBD*s containing the objects you will access in your PowerBuilder.Application session. All the libraries should have the same executable type, either compiled machine code or Pcode. Setting LibraryList is effective only before you create any object instances. After the first object is instantiated, changes are ignored.
MachineCode	Boolean	Specifies whether the objects you will instantiate are generated with machine code or Pcode. Values are: • TRUE – (Default) The library was compiled with machine code. The default file name extension is *DLL*. • FALSE – The library was compiled with Pcode. The default file name extension is *PBD*. Setting MachineCode is effective only before you create any object instances. After the first object is instantiated, changes are ignored.

Functions

Table 20-9: Functions of PowerBuilder.Application

Function	Datatype returned	Description
CreateObject	OLEObject	Creates an instance of a class user object in an OLE server session. An object reference is returned to the client session so that the client can access its properties and functions by means of automation syntax.
GenerateGUID	Long	Generates a globally unique identifier and puts the string representation of the identifier in a string variable passed by reference. A network card is required for this function to succeed.

Function	Datatype returned	Description
GenerateRegFile	Long	Generates a file that contains instructions for updating the registry with information about a PowerBuilder object that you want to deploy as an independent automation server.
GenerateTypeLib	Long	Generates a type library file with browsing information about a PowerBuilder object that you want to deploy as an independent automation server.

CreateObject function

Description

Creates an instance of a PowerBuilder class in a PowerBuilder.Application OLE server session.

Applies to

PowerBuilder.Application (automation server)

Syntax

{ *automationobject*.} CreateObject (*classname*)

Argument	Description
automationobject	When PowerBuilder is the OLE client, the name of the OLEObject instantiated with the PowerBuilder.Application automation server. For other clients, use syntax appropriate for calling a function belonging to an automation object.
classname	A string specifying the name of the class you want to create. The class must be defined in one of the libraries specified in the PowerBuilder.Application LibraryList property.

Return value

OLEObject. Returns a reference to the instantiated object, which is valid for automation. If the object could not be created, CreateObject returns NULL.

Usage

If the OLE client is Visual Basic, you can test for the keyword nothing to see if CreateObject succeeded.

If the object's executable type does not correspond to the value of the MachineCode property, then CreateObject returns NULL. All the objects created for one PowerBuilder.Application session must have the same executable type (either Pcode or compiled machine code). When you create more than one object in a single PowerBuilder.Application session, those objects can be passed as function arguments or returned as results.

You do not need to use the CREATE statement for the OLEObject variable before calling the CreateObject function.

Examples

This example is a PowerBuilder script that starts the PowerBuilder.Application server and creates an object that is contained in *MYLIBRARY.DLL*. If the object is created successfully, the script calls the function uf_calc, which returns a Long as a status code and passes back the result of the calculation in the variable *ld_result*:

```
OLEObject PBObject, PBNVObject
long ll_status
double ld_result

PBObject = CREATE OLEObject
ll_status = PBObject.ConnectToNewObject &
   ("PowerBuilder.Application")
IF ll_status = 0 THEN
   // Handle the error
ELSE
   PBObject.LibraryList = "c:\myappl\mylibrary.dll"
   PBObject.MachineCode = TRUE

   PBNVObject = CREATE OLEObject

   PBNVObject = &
      PBObject.CreateObject("nvo_myobject")
   IF IsNull(PBNVObject) THEN
   // Handle the error
   ELSE
   ll_status = PBNVObject.uf_calc &
      (12, 14, REF result)
   END IF
   DESTROY PBNVObject
   PBObject.DisconnectObject( )
END IF

DESTROY PBObject
```

This example is a Visual Basic script that does the same tasks as the PowerBuilder script above:

```
Dim PBObject as object
Dim PBNVObject as object
Dim status as long
Dim result as double
```

```
Set PBObject = _
   CreateObject("PowerBuilder.Application")
If PBObject is nothing then
   REM handle the error
Else
   PBObject.LibraryList = “c:\myappl\mylibrary.dll”
   Set PBNVObject = _
   PBObject.CreateObject(“nvo_myobject”)
   If PBNVObject is nothing then
   REM handle the error
   Else
   status = PBNVObject.uf_calc(12, 14, REF result)
   Set PBNVObject = nothing
   End if

   Set PBObject = nothing
End if
```

See also

ConnectToNewObject

GenerateGUID function

Description

Creates a globally unique identifier (GUID) that can be used as a class ID (CLSID) when you register an object and its type library in the Windows registry. Both the object and its type library have a GUID.

Applies to

PowerBuilder.Application (automation server)

Syntax

{ *automationobject*.} GenerateGUID (REF *guidvariable*)

Argument	Description
automationobject	When PowerBuilder is the OLE client, the name of the OLEObject instantiated with the PowerBuilder.Application automation server. For other clients, use syntax appropriate for calling a function belonging to an automation object.
REF *guidvariable*	A string variable, passed by reference, that will hold the string representation of the GUID generated by the function. Because GenerateGUID is called as an automation function whose prototype is not checked by the compiler, you must specify REF to pass *guidvariable* by reference.

Return value

Long. Returns a status code indicating whether an error occurred. Values are:

0 A GUID was generated successfully.

-1 Unable to load the *DLL* for calling GenerateGUID.

-2 No network card was found, so a GUID could not be generated.

-3 Creating a GUID failed.

-9 Unknown error.

Usage

GenerateGUID requires:

- *RPCRT4.DLL*, in which it calls the function UuidCreate. The *DLL* must be present on your path.
- A network card. The function uses the identification information for your network card to generate a unique value.

If you do not meet these conditions, you can contact Microsoft to obtain GUIDs.

A GUID is a long hexadecimal number that can be represented as a string in the format (8-4-4-4-12 digits):

```
{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}
```

The GUID is used in the Windows registry to identify each program entity. Both the object being registered and its type library have a GUID.

You cannot perform automation against a PowerBuilder object until the program object is defined in the registry. GenerateGUID is a tool to help you create the necessary registry information. When you generate a GUID, you pass the GUID to the functions GenerateRegFile and GenerateTypeLib to create registry update files. You use those update files on each user's machine to install your object in its registry.

Each time you create a new GUID for your object and update the registry, a new registry entry is created. If you use the same GUID in your registry update file, the old registry entry is replaced with the new one.

When you deploy an update to your object, you can generate new update files with the same GUID. Existing applications can successfully access the newly updated object as long as its interface (its properties and function signatures) remains compatible. However, if you changed the interface for your object, you should use a new GUID for the update. Existing applications can continue to use the old object and new applications can use the new one, taking advantage of changed and new functionality.

For how to use a PowerBuilder wizard to register objects, see “Registering the object” on page 385.

Examples

This example establishes a PowerBuilder.Application server session and generates a GUID:

```
oleObject PBObject
string ls_GUID
long ll_result

PBObject = CREATE oleObject

result = PBObject.ConnectToNewObject &
   ("PowerBuilder.Application")
IF result < 0 THEN
   // handle the error
ELSE
   ll_result = PBObject.GenerateGUID(REF ls_GUID)
END IF
```

See also

GenerateRegFile
GenerateTypeLib

GenerateRegFile function

Description

Creates a file with registry update instructions for deploying a PowerBuilder object as an automation server.

Applies to

PowerBuilder.Application (automation server)

Syntax

{ *automationobject*.} GenerateRegFile (*guid*, *classname*, *progid*, *majorversion*, *minorversion*, *description*, *outputfilename*)

Argument	Description
automationobject	When PowerBuilder is the OLE client, the name of the OLEObject instantiated with the PowerBuilder.Application automation server. For other clients, use syntax appropriate for calling a function belonging to an automation object.

Argument	Description
guid	A string whose value is the globally unique identifier (GUID) you want to use for the object You can specify a GUID that you: • Generated with a call to GenerateGUID • Generated earlier and are reusing (because this object replaces an earlier version) • Received from Microsoft
classname	A string whose value is the name of the class user object that will be the automation server. The object must be in one of the libraries specified in the LibraryList property.
progid	A string whose value is the programmatic identifier that people will use to connect to your server.
majorversion	An integer whose value you want to use for the major version number of your server object.
minorversion	An integer whose value you want to use for the minor version number of your server object.
description	A string whose value is a description of your server object. The description will be displayed in the Registry Editor and other OLE browsers.
outputfilename	A string whose value is the name of the file the GenerateRegFile will create. The default file name extension recognized by the registry is *REG*.

Return value

Long. Returns a status code indicating whether an error occurred. Values are:

0 The registry update file was generated successfully.

-1 Memory allocation error.

-2 No output file name was provided.

-3 Unable to open the output file.

-9 Unknown error.

Usage

Before you call GenerateRegFile, you must start a PowerBuilder.Application session and set the LibraryList and MachineCode properties to the value that the server object will use. You also need a GUID to serve as the object's class identifier (CLSID). For information about the format of a GUID, see the GenerateGUID function on page 407.

After you create a registry update file, you can also generate a type library file for the object, which provides browsing information for your server object. See the GenerateTypeLib function on page 412.

The default extension for a registry update file is *REG*. You can install the object in the registry by double-clicking on the update file. Information is installed in the registry about the object, including the location of its *DLL* and the location of its type library file, if any. For how to use a PowerBuilder tool to register objects, see “Registering the object” on page 385.

Examples

This example calls GenerateRegFile. (Do *not* use this GUID in your own applications.)

```
long ll_result
ll_result = GenerateRegFile( &
   "{12345678-1234-1234-1234-123456789012}", &
   "uo_salary_formulas",   "MyCompany.SalaryFormulas",
&
   1, 0, &
   "PowerBuilder functions for calculating salaries", &
   "c:\pbds\bizrules.reg")
```

This example establishes a PowerBuilder.Application server session and sets all the values necessary for generating a registry update file.

```
oleObject PBObject
string ls_GUID
long ll_result

PBObj = CREATE oleObject

result = &
  PBObj.ConnectToNewObject("PowerBuilder.Application")
IF result < 0 THEN
   // Handle the error
ELSE
   PBObject.LibraryList = "c:\myappl\mylibrary.pbd"
   PBObject.MachineCode = FALSE

   ll_result = PBObject.GenerateGUID(REF ls_GUID)
   IF ll_result < 0 THEN


   // Handle the error
   ELSE
   ll_result = PBObject.GenerateRegFile(   ls_GUID, &
      "uo_myserverobject",   "MyCompany.Object",
&
      1, 0,    "My object's description", &
      "c:\myappl\object.reg")
```

```
            IF ll_result < 0 THEN
               // Handle the error
            ELSE
               // Generate Type Library
            END IF
            END IF
         END IF
```

See also

GenerateGUID
GenerateTypeLib

GenerateTypeLib function

Description

Creates a file with type library information for a PowerBuilder object you are deploying as an automation server.

Applies to

PowerBuilder.Application (automation server)

Syntax

{ *automationobject*.} GenerateTypeLib (*classguid*, *classname*, *progid*, *localeid*, *majorversion*, *minorversion*, *description*, *helpcontext*, *helpfile*, *typelibguid*, *typelibfilename*, *registryfilename*)

Argument	Description
automationobject	When PowerBuilder is the OLE client, the name of the OLEObject instantiated with the PowerBuilder.Application automation server. For other clients, use syntax appropriate for calling a function belonging to an automation object.
classguid	A String whose value is the globally unique identifier (GUID) that you used for the object when you generated its registry file
classname	A String whose value is the name of the object in the PowerBuilder library for which type information will be generated. *classname* should be the same name you specified when you generated the registry file.
progid	A String whose value is the programmatic identifier that people will use to connect to your server. *progid* should be the same value you specified when you generated the registry file.

Argument	Description
localeid	A Long whose value is the locale identifier (LCID) for the language of your object. Values for specific languages are defined in Microsoft documentation. You can also specify a value of the PowerBuilder enumerated datatype LanguageID.
majorversion	An Integer whose value you specified for the major version number of your server object.
minorversion	An Integer whose value you specified for the minor version number of your server object.
description	A String whose value you specified as the description of your server object.
helpcontext	A Long whose value is the context ID for a Help topic in *helpfile*. Specify 0 to display the Help file's contents page.
helpfile	A String whose value is the name of a Help file containing Help topics for your server object.
typelibguid	A String whose value will be the globally unique identifier (GUID) for the TypeLib information in the registry.
typelibfilename	A String whose value is the name of the file GenerateTypeLib will create. The default file name extension recognized by the registry is *TLB*.
registryfilename	A String whose value is the name of the registry update file generated for this object with GenerateRegFile. If you just want to generate a type library, specify an empty string for *registryfilename*.

Return value

Long. Returns a status code indicating whether an error occurred. Values are:

0 The registry update file was generated successfully.

2 No output file name was provided.

3 Cannot create the type library.

4 Invalid name or error setting name.

5 Invalid locale identifier or error setting the locale identifier.

6 Error converting the TypeLib GUID to a CLSID.

7 Invalid GUID or error setting GUID.

7 Error setting description.

9 Error setting version.

10 Error setting Help context.

11 Error setting Help file name.

14 Class not found in the libraries specified in LibraryList.

18 Error converting ClassName GUID to CLSID.

30 Error loading standard type library (OLE is not installed correctly).

31 Error getting IUnknown type information (OLE is not installed correctly).

32 Error getting IDispatch type information (OLE is not installed correctly).

36 Error opening the registry update file associated with this type library.

Usage

The type library is optional. You need it only if you want OLE browsers to display information about the properties and methods of your object.

If you want to register your type library, then a valid registry update file must exist for the object before you create the Type Library file. The process of creating the Type Library adds information to the registry update file.

To generate the registry update file, see the GenerateRegFile function on page 409.

The values you specify for the arguments of GenerateTypeLib must match the values in the associated registry update file that has already been created. Otherwise, you will create a TypeLib entry in the registry that is not associated with any object.

Before you call GenerateTypeLib, you must start a PowerBuilder.Application session and set the LibraryList and MachineCode properties to the values for the server object. You also need two GUIDs, the GUID used for the object's CLSID (specified when you created the registry update file) and a GUID for the type library itself. For information about the format of a GUID, see the GenerateGUID function on page 407.

After you generate the registry update and type library files, you can register the object in the registry on your system or on other systems.

When you register your object in the registry, it stores information about the locations of the library file and the type library file. The path you specified for the LibraryList property specifies its location or must be in the path. The type library file should be in its intended directory when you register the object.

Examples

This example calls GenerateTypeLib. (Do *not* use this GUID in your own applications.)

```
long ll_result
ll_result = GenerateTypeLib( &
   "{12345678-1234-1234-1234-123456789012}", &
   "uo_salary_formulas", "MyCompany.SalaryFormulas", &
   0, 1, 0, &
   "PowerBuilder functions for calculating salaries", &
   0, "c:\pbds\bizrules.hlp", &
   "{12345679-1234-1234-1234-123456789012}", &
   "c:\pbds\bizrules.tlb",   "c:\pbds\bizrules.reg")
```

This example establishes a PowerBuilder.Application server session and sets all the values necessary for generating a registry update file:

```
oleObject PBObject
string ls_GUID, ls_typelibGUID
long ll_result

PBObject = CREATE oleObject

result = PBObject.ConnectToNewObject &
   ("PowerBuilder.Application")
IF result < 0 THEN
   // Handle the error
   RETURN -1
END IF

PBObject.LibraryList = "c:\myappl\mylibrary.pbd"
PBObject.MachineCode = FALSE

// GUID for object's CLSID
ll_result = PBObject.GenerateGUID(REF ls_GUID)
IF ll_result < 0 THEN
   // Handle the error
   RETURN -1
END IF

// GUID for object's Type Library
ll_result = PBObject.GenerateGUID(REF ls_typelibGUID)
IF ll_result < 0 THEN
   // Handle the error
   RETURN -1
END IF

// Generate registry update file
ll_result = PBObject.GenerateRegFile( ls_GUID, &
   "uo_myserverobject",   "MyCompany.Object", &
   1, 0,    "My object's description", &
   "c:\myappl\object.reg")
IF ll_result < 0 THEN
   / Handle the error
   RETURN -1
END IF

// Generate Type Library
ll_result = PBObject.GenerateTypeLib( ls_GUID, &
   uo_myserverobject",   "MyCompany.Object",      0,
1, 0, &
   "My object's description",   0, $
   "c:\myappl\myhelp.hlp", ls_typelibGUID, &
   "c:\myappl\object.tlb",   "c:\myappl\object.reg")
IF ll_result < 0 THEN
   // Handle the error
   RETURN -1
END IF
```

See also

GenerateGUID
GenerateRegFile

Exception codes

Automation clients accessing a PowerBuilder server may generate OLE exceptions under certain circumstances. When user actions cause an exception, PowerBuilder populates the EXCEPINFO structure passed by the caller with an exception code identifying the type of exception. For a PowerBuilder client, the exception information is passed as arguments to the ExternalException event.

Client applications might receive these exception codes from the PowerBuilder automation server:

Table 20-10: PowerBuilder automation server exception codes

Code	Meaning
1001	Memory allocation failure.
1002	Requested Object is not in the library list.
1003	Object create failed.
1004	Binary format does not match version or is out of date.
1005	Property accessed as an array is not an array.
1006	Unexpected error executing script.
1007	No method was found matching the requested name and argument types.
1008	Unable to convert variant argument type.

Sample registry update file

This sample registry file has registration information for a class user object named uo_salarydata.

```
REGEDIT4
;;;;;;;;;;;;;;;;
;
; Registry entries for MyCompany.SalaryData
;
; CLSID = {E022EF01-6789-11CF-92BF-00805F9236E9}
;
; PowerBuilder Generated Registry File
;;;;;;;;;;;;;;;;
; Version independent entries:
[HKEY_CLASSES_ROOT\MyCompany.SalaryData]
@="DataStore with functions returning salary statistics"
[HKEY_CLASSES_ROOT\MyCompany.SalaryData\CLSID]
@="{E022EF01-6789-11CF-92BF-00805F9236E9}"
[HKEY_CLASSES_ROOT\MyCompany.SalaryData\CurVer]
@="MyCompany.SalaryData.1"
[HKEY_CLASSES_ROOT\MyCompany.SalaryData\NotInsertable"
@=""

; Version specific entries:
[HKEY_CLASSES_ROOT\AppID\{E022EF01-6789-11CF-92BF-00805F9236E9}]
@="DataStore with functions returning salary statistics"
[HKEY_CLASSES_ROOT\MyCompany.SalaryData.1]
@="DataStore with functions returning salary statistics"
[HKEY_CLASSES_ROOT\MyCompany.SalaryData.1\CLSID]
@="{E022EF01-6789-11CF-92BF-00805F9236E9}"
[HKEY_CLASSES_ROOT\MyCompany.SalaryData.1\NotInsertable]
@=""

; CLSID entries:
[HKEY_CLASSES_ROOT\CLSID\{E022EF01-6789-11CF-92BF-
00805F9236E9}]
@="DataStore with functions returning salary statistics"
"AppID"="{E022EF01-6789-11CF-92BF-00805F9236E9}"
[HKEY_CLASSES_ROOT\CLSID\{E022EF01-6789-11CF-92BF-0805F9236E9}\ProgID]
@="MyCompany.SalaryData.1"
[HKEY_CLASSES_ROOT\CLSID\{E022EF01-6789-11CF-92BF-
00805F9236E9}\VersionIndependentProgID]
@="MyCompany.SalaryData"
[HKEY_CLASSES_ROOT\CLSID\{E022EF01-6789-11CF-92BF-
00805F9236E9}\InProcServer32]
@="pbVM110.dll"
```

```
"ThreadingModel"="Apartment"
[HKEY_CLASSES_ROOT\CLSID\{E022EF01-6789-11CF-
92BF 00805F9236E9}\NotInsertable]
@=""
[HKEY_CLASSES_ROOT\CLSID\{E022EF01-6789-11CF-92BF-00805F9236E9}\Programmable]
@=""
[HKEY_CLASSES_ROOT\CLSID\{E022EF01-6789-11CF-92BF-
00805F9236E9}\PowerBuilder\ClassName]
@="uo_salarydata"
[HKEY_CLASSES_ROOT\CLSID\{E022EF01-6789-11CF-92BF-
00805F9236E9}\PowerBuilder\LibraryList]
@="D:\\pbserver.pbd"
[HKEY_CLASSES_ROOT\CLSID\{E022EF01-6789-11CF-92BF-
00805F9236E9}\PowerBuilder\BinaryType]
@="PCODE"
; Type Library Registration entries:
[HKEY_CLASSES_ROOT\CLSID\{E022EF01-6789-11CF-92BF-00805F9236E9}\TypeLib]
@="{E022EF02-6789-11CF-92BF-00805F9236E9}"
[HKEY_CLASSES_ROOT\TypeLib\{E022EF02-6789-11CF-92BF-00805F9236E9}\1.0]
@="Type Library for DataStore with functions returning salary statistics"
[HKEY_CLASSES_ROOT\TypeLib\{E022EF02-6789-11CF-92BF-
00805F9236E9}\1.0\9\Win32]
@="D:\\pbserver.tlb"
```

CHAPTER 21 Building a Mail-Enabled Application

About this chapter

This chapter describes how to use the messaging application program interface (MAPI) with PowerBuilder applications to send and receive electronic mail.

Contents

About MAPI

PowerBuilder supports MAPI (messaging application program interface), so you can enable your applications to send and receive messages using any MAPI-compliant electronic mail system.

For example, your PowerBuilder applications can:

- Send mail with the results of an analysis performed in the application
- Send mail when a particular action is taken by the user
- Send mail requesting information
- Receive mail containing information needed by the application's user

How MAPI support is implemented

To support MAPI, PowerBuilder provides the items listed in Table 21-1.

Table 21-1: PowerBuilder MAPI support

Item	Name
A mail-related system object	MailSession
Mail-related structures	MailFileDescription MailMessage MailRecipient

Item	Name
Object-level functions for the MailSession object	MailAddress MailDeleteMessage MailGetMessages MailHandle MailLogoff MailLogon MailReadMessage MailRecipientDetails MailResolveRecipient MailSaveMessage MailSend
Enumerated datatypes	MailFileType MailLogonOption MailReadOption MailRecipientType MailReturnCode

Using MAPI

To use MAPI, you create a MailSession object, then use the MailSession functions to manage it.

For example:

```
MailSession PBmail
PBmail = CREATE MailSession

PBmail.MailLogon(...)
... // Manage the session: send messages,
... // receive messages, and so on.
PBmail.MailLogoff()

DESTROY PBmail
```

You can use the Browser to get details about the attributes and functions of the MailSession system object, the attributes of the mail-related structures, and the valid values of the mail-related enumerated datatypes.

For information about using the Browser, see the *PowerBuilder User’s Guide*. For complete information about the MailSession functions, see the *PowerScript Reference*. For complete information about MAPI, see the documentation for your MAPI-compliant mail application.

CHAPTER 22

Using External Functions and Other Processing Extensions

About this chapter

This chapter describes how to use external functions and other processing extensions in PowerBuilder.

Contents

Using external functions

External functions are functions that are written in languages other than PowerScript and stored in dynamic libraries. External functions are stored in dynamic link libraries (*DLL*s).

You can use external functions written in any language that supports the standard calling sequence for 32-bit platforms.

If you are calling functions in libraries that you have written yourself, remember that you need to export the functions. Depending on your compiler, you can do this in the function prototype or in a linker definition (*DEF*) file.

Use _stdcall convention

C and C++ compilers typically support several calling conventions, including _cdecl (the default calling convention for C programs), _stdcall (the standard convention for Windows API calls), _fastcall, and thiscall. PowerBuilder, like many other Windows development tools, requires external functions to be exported using the WINAPI (_stdcall) format. Attempting to use a different calling convention can cause an application crash.

When you create your own C or C++ DLLs containing functions to be used in PowerBuilder, make sure that they use the standard convention for Windows API calls. For example, if you are using a DEF file to export function definitions, you can declare the function like this:

```
LONG WINAPI myFunc()
{
...
};
```

Using PBNI
You can also call external functions in PowerBuilder extensions. PowerBuilder extensions are built using the PowerBuilder Native Interface (PBNI). For more information about building PowerBuilder extensions, see the *PowerBuilder Native Interface Programmer's Guide and Reference*. For more information about using PowerBuilder extensions, see the *PowerBuilder Extension Reference*.

Declaring external functions

Before you can use an external function in a script, you must declare it.

Two types

You can declare two types of external functions:

- **Global external functions**, which are available anywhere in the application
- **Local external functions**, which are defined for a particular type of window, menu, or user object

  These functions are part of the object's definition and can always be used in scripts for the object itself. You can also choose to make these functions accessible to other scripts as well.

Datatypes for external function arguments

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

For a comparison of datatypes in external functions and datatypes in PowerBuilder, see the section on declaring and calling external functions in the *PowerScript Reference*.

❖ **To declare an external function:**

1 If you are declaring a local external function, open the object for which you want to declare it.

2 In the Script view, select Declare in the first drop-down list and either Global External Functions or Local External Functions from the second list.

3 Enter the function declaration in the Script view.

For the syntax to use, see the *PowerScript Reference* or the examples below.

4 Save the object.

PowerBuilder compiles the declaration. If there are syntax errors, an error window opens, and you must correct the errors before PowerBuilder can save the declaration.

Modifying existing functions
You can also modify existing external function declarations in the Script view.

Sample declarations

Suppose you have created a C dynamic library, *SIMPLE.DLL*, that contains a function called SimpleFunc that accepts two parameters: a character string and a structure. The following statement declares the function in PowerBuilder, passing the arguments by reference:

```
FUNCTION int SimpleFunc(REF string lastname, &
   REF my_str pbstr) LIBRARY "simple.dll"
```

By default, PowerBuilder handles string arguments and return values as if they have Unicode encoding. If SimpleFunc passes ANSI strings as arguments, you must use this syntax to declare it:

```
FUNCTION int SimpleFunc(REF string lastname, &
   REF my_str pbstr) LIBRARY "simple.dll" &
   ALIAS FOR "SimpleFunc;ansi"
```

Declaring Windows API functions

The Windows API includes over a thousand functions that you can call from PowerBuilder. The following examples show sample declarations for functions in the 32-bit Windows API libraries *KERNEL32.DLL*, *GDI32.DLL*, and *USER32.DLL*.

Windows API calls
Some 32-bit function names end with A (for ANSI) or W (for wide). Use wide function names in PowerBuilder.

For a complete list of Windows API functions, see the Microsoft Windows SDK documentation. For examples of PowerBuilder declaration syntax and scripts, search for Windows API calls in the Technical Documents section of the Sybase Web site at http://www.sybase.com/support/techdocs/.

The following statements declare a function that gets the handle of any window that is called by name, and a function that releases the open object handle:

```
FUNCTION ulong FindWindowW(ulong classname, &
      string windowname) LIBRARY "User32.dll"
FUNCTION boolean CloseHandle(ulong w_handle) &
      LIBRARY "Kernel32.dll"
```

The following statement declares a function that draws a pie chart based on the coordinates received:

```
FUNCTION boolean Pie(ulong hwnd,long x1,long y1, &
      long x2,long y2,long x3,long y3,long x4, &
      long y4) LIBRARY "Gdi32.dll"
```

The following statement declares an external C function named IsZoomed:

```
FUNCTION boolean IsZoomed(Ulong handle)  &
      LIBRARY "User32.DLL"
```

A script that uses IsZoomed is included as an example in “Using utility functions to manage information” on page 428.

For more information about these functions, see the Microsoft documentation in the MSDN Library at http://msdn.microsoft.com/library/default.asp.

Passing arguments

In PowerBuilder, you can define external functions that expect arguments to be passed by reference or by value. When you pass an argument by reference, the external function receives a pointer to the argument and can change the contents of the argument and return the changed contents to PowerBuilder. When you pass the argument by value, the external function receives a copy of the argument and can change the contents of the copy of the argument. The changes affect only the local copy; the contents of the original argument are unchanged.

The syntax for an argument that is passed by reference is:

REF *datatype arg*

The syntax for an argument that is passed by value is:

datatype arg

Passing numeric datatypes

The following statement declares the external function TEMP in PowerBuilder. This function returns an integer and expects an integer argument to be passed by reference:

```
FUNCTION int TEMP(ref int degree) LIBRARY
"LibName.DLL"
```

The same statement in C would be:

```
int _stdcall TEMP(int * degree)
```

Since the argument is passed by reference, the function can change the contents of the argument, and changes made to the argument within the function will directly affect the value of the original variable in PowerBuilder. For example, the C statement `*degree = 75` would change the argument named degree to 75 and return 75 to PowerBuilder.

The following statement declares the external function TEMP2 in PowerBuilder. This function returns an Integer and expects an Integer argument to be passed by value:

```
FUNCTION int TEMP2(int degree) LIBRARY "LibName.DLL"
```

The same statement in C would be:

```
int _stdcall TEMP2(int degree)
```

Since the argument is passed by value, the function can change the contents of the argument. All changes are made to the local copy of the argument; the variable in PowerBuilder is not affected.

Passing strings

PowerBuilder assumes all string arguments and returned values use Unicode encoding. If a function uses strings with ANSI encoding, you need to add an ALIAS FOR clause to the function declaration and add a semicolon followed by the ansi keyword. For example:

```
FUNCTION string NAME(string CODE) LIBRARY
"LibName.DLL" ALIAS FOR "NAME;ansi"
```

Passing by value The following statement declares the external C function NAME in PowerBuilder. This function expects a String argument with Unicode encoding to be passed by value:

```
FUNCTION string NAME(string CODE) LIBRARY
"LibName.DLL"
```

The same statement in C would point to a buffer containing the String:

```
char * _stdcall NAME(char  * CODE)
```

Since the String is passed by value, the C function can change the contents of its local copy of *CODE*, but the original variable in PowerBuilder is not affected.

Passing by reference PowerBuilder has access only to its own memory. Therefore, an external function cannot return to PowerBuilder a pointer to a string. (It cannot return a memory address.)

When you pass a string to an external function, either by value or by reference, PowerBuilder passes a pointer to the string. If you pass by value, any changes the function makes to the string are not accessible to PowerBuilder. If you pass by reference, they are.

The following statement declares the external C function NAME2 in PowerBuilder. This function returns a String and expects a String argument to be passed by reference:

```
FUNCTION string NAME2(ref string CODE)  &
      LIBRARY "LibName.DLL"
```

In C, the statement would be the same as when the argument is passed by value, shown above:

```
char * _stdcall NAME2(char * CODE)
```

The String argument is passed by reference, and the C function can change the contents of the argument and the original variable in PowerBuilder. For example, `Strcpy(CODE,STUMP)` would change the contents of *CODE* to *STUMP* and change the variable in the calling PowerBuilder script to the contents of variable *STUMP*.

If the function NAME2 in the preceding example takes a user ID and replaces it with the user's name, the PowerScript string variable *CODE* must be long enough to hold the returned value. To ensure that this is true, declare the String and then use the Space function to fill the String with blanks equal to the maximum number of characters you expect the function to return.

If the maximum number of characters allowed for a user's name is 40 and the ID is always five characters, you would fill the String *CODE* with 35 blanks before calling the external function:

```
String CODE
CODE = ID + Space(35)
. . .
NAME2(CODE)
```

For information about the Space function, see the *PowerScript Reference*.

Passing chars to C functions Char variables passed to external C functions are converted to the C char type before passing. Arrays of Char variables are converted to the equivalent C array of char variables.

An array of Char variables embedded in a structure produces an embedded array in the C structure. This is different from an embedded String, which results in an embedded pointer to a string in the C structure.

Recommendation
Whenever possible, pass String variables back to PowerBuilder as a return value from the function.

Calling external functions on UNIX

In PowerBuilder custom class user objects that you plan to deploy as EAServer components on a UNIX platform, you can call external functions in shared libraries compiled on the operating system where the server is running. You cannot call external functions in libraries that make Windows API calls or rely on graphical processing.

You use the standard PowerScript syntax to declare functions that reside in UNIX shared libraries. For example, this statement declares the function getcwd in the standard C library on Solaris:

```
FUNCTION string getcwd(REF string buff, &
      unsigned int size) LIBRARY "/usr/lib/libc.so"
```

You call the function from a script in your application in the way you call any other function. In this example, the space function allocates enough space to hold the directory name returned by getcwd:

```
string ls_return, ls_directory
ls_directory = space(100)
```

```
. . .
ls_return = getcwd(ls_directory, 100)
```

Updating the load library path
On UNIX, when your component calls an external function, it must be able to find the shared library in which that function resides. To ensure this, you must update the library path environment variable to include the directory where that shared library is stored.

Using utility functions to manage information

The utility functions provide a way to obtain and pass Windows information to external functions and can be used as arguments in the PowerScript Send function. Table 22-1 describes the PowerScript utility functions.

Five utility functions

Table 22-1: Utility functions

Function	Return value	Purpose
Handle	UnsignedInt	Returns the handle to a specified object.
IntHigh	UnsignedInt	Returns the high word of the specified Long value. IntHigh is used to decode Windows values returned by external functions or the LongParm attribute of the Message object.
IntLow	UnsignedInt	Returns the low word of the specified Long value. IntLow is used to decode Windows values returned by external functions or the LongParm attribute of the Message object.
Long	Long	Combines the low word and high word into a Long. The Long function is used to pass values to external functions.
LongLong	LongLong	Combines the low word and high word into a LongLong. The LongLong function is used to pass values to external functions.

Examples

This script uses the external function IsZoomed to test whether the current window is maximized. It uses the Handle function to pass a window handle to IsZoomed. It then displays the result in a SingleLineEdit named sle_output:

```
boolean Maxed
```

```
Maxed = IsZoomed(Handle(parent))
if Maxed then sle_output.Text = "Is maxed"
if not Maxed then sle_output.Text = "Is normal"
```

This script passes the handle of a window object to the external function FlashWindow to change the title bar of a window to inactive and then active:

```
// Declare loop counter and handle to window object
int   nLoop
uint  hWnd
// Get the handle to the PowerBuilder window.
hWnd = handle(This)
// Make the title bar inactive.
FlashWindow (hWnd, TRUE)
//Wait ...
For nLoop = 1 to 300
Next
// Return the title bar to its active color.
FlashWindow (hWnd, FALSE)
```

Sending Windows messages

To send Windows messages to a window that you created in PowerBuilder or to an external window (such as a window you created using an external function), use the Post or Send function. To trigger a PowerBuilder event, use the EVENT syntax or the TriggerEvent or PostEvent function.

Using Post and Send

You usually use the Post and Send functions to trigger Windows events that are not PowerBuilder-defined events. You can include these functions in a script for the window in which the event will be triggered or in any script in the application.

Post is asynchronous: the message is posted to the message queue for the window or control. Send is synchronous: the window or control receives the message immediately.

As of PowerBuilder 6.0, all events posted by PowerBuilder are processed by a separate queue from the Windows system queue. PowerBuilder posted messages are processed before Windows posted messages.

Obtaining the window's handle
To obtain the handle of the window, use the Handle function. To combine two integers to form the Long value of the message, use the Long function. Handle and Long are utility functions, which are discussed later in this chapter.

Triggering PowerBuilder events

To trigger a PowerBuilder event, you can use the techniques listed in Table 22-2.

Table 22-2: Triggering PowerBuilder events

Technique	Description
TriggerEvent function	A synchronous function that triggers the event immediately in the window or control
PostEvent function	An asynchronous function: the event is posted to the event queue for the window or control
Event call syntax	A method of calling events directly for a control using dot notation

All three methods bypass the messaging queue and are easier to code than the Send and Post functions.

Example All three statements shown below click the CommandButton cb_OK and are in scripts for the window that contains cb_OK.

The Send function uses the Handle utility function to obtain the handle of the window that contains cb_OK, then uses the Long function to combine the handle of cb_OK with 0 (BN_CLICK) to form a Long that identifies the object and the event:

```
Send(Handle(Parent),273,0,Long(Handle(cb_OK),0))
cb_OK.TriggerEvent(Clicked!)
cb_OK.EVENT Clicked()
```

The TriggerEvent function identifies the object in which the event will be triggered and then uses the enumerated datatype Clicked! to specify the clicked event.

The dot notation uses the EVENT keyword to trigger the Clicked event. TRIGGER is the default when you call an event. If you were posting the clicked event, you would use the POST keyword:

```
Cb_OK.EVENT POST Clicked()
```

The Message object

The Message object is a predefined PowerBuilder global object (like the default Transaction object SQLCA and the Error object) that is used in scripts to process Microsoft Windows events that are not PowerBuilder-defined events.

When a Microsoft Windows event occurs that is not a PowerBuilder-defined event, PowerBuilder populates the Message object with information about the event.

Other uses of the Message object

The Message object is also used:

- To communicate parameters between windows when you open and close them

  For more information, see the descriptions of OpenWithParm, OpenSheetWithParm, and CloseWithReturn in the *PowerScript Reference*.

- To pass information to an event if optional parameters were used in TriggerEvent or PostEvent

  For more information, see the *PowerScript Reference*.

Customizing the Message object

You can customize the global Message object used in your application by defining a standard class user object inherited from the built-in Message object. In the user object, you can add additional properties (instance variables) and functions. You then populate the user-defined properties and call the functions as needed in your application.

For more information about defining standard class user objects, see the *PowerBuilder User’s Guide*.

Message object properties

The first four properties of the Message object correspond to the first four properties of the Microsoft Windows message structure.

Table 22-3: Message object properties

Property	Datatype	Use
Handle	Integer	The handle of the window or control.
Number	Integer	The number that identifies the event (this number comes from Windows).

Property	Datatype	Use
WordParm	UnsignedInt	The word parameter for the event (this parameter comes from Windows). The parameter's value and meaning are determined by the event.
LongParm	Long	The long parameter for the event (this number comes from Windows). The parameter's value and meaning are determined by the event.
DoubleParm	Double	A numeric or numeric variable.
StringParm	String	A string or string variable.
PowerObjectParm	PowerObject	Any PowerBuilder object type including structures.
Processed	Boolean	A boolean value set in the script for the user-defined event: • TRUE—The script processed the event. Do not call the default window Proc (DefWindowProc) after the event has been processed. • FALSE—(Default) Call DefWindowProc after the event has been processed.
ReturnValue	Long	The value you want returned to Windows when Message.Processed is TRUE. When Message.Processed is FALSE, this attribute is ignored.

Use the values in the Message object in the event script that caused the Message object to be populated. For example, suppose the FileExists event contains the following script. OpenWithParm displays a response window that asks the user if it is OK to overwrite the file. The return value from FileExists determines whether the file is saved:

```
OpenWithParm( w_question, &
   "The specified file already exists. " + &
   "Do you want to overwrite it?" )
IF Message.StringParm = "Yes" THEN
   RETURN 0  // File is saved
ELSE
   RETURN -1 // Saving is canceled
END IF
```

For information on Microsoft message numbers and parameters, see the Microsoft Software Developer's Kit (SDK) documentation.

Context information

The PowerBuilder context feature allows applications to access certain host (non-PowerBuilder) services. This is a PowerBuilder implementation of functionality similar to the COM QueryInterface. PowerBuilder provides access to the following host services:

- Context information service
- Context keyword service
- CORBACurrent service
- Error logging service
- Internet service
- Secure Sockets Layer service
- Transaction server service

PowerBuilder creates service objects appropriate for the current execution context (native PowerBuilder, PowerBuilder window plug-in, PowerBuilder window ActiveX, transaction server). This allows your application to take full advantage of the execution environment. For example, in a PowerBuilder window plug-in, your application can access parameters specified in the Embed element. Additionally, when running the PowerBuilder window ActiveX under Internet Explorer, your application can access the ActiveX automation server, which allows your program to access and control the Internet browser.

The context feature uses seven PowerBuilder service objects: ContextInformation, ContextKeyword, CORBACurrent, ErrorLogging, Inet, SSLServiceProvider, and TransactionServer; it also uses the InternetResult object. (The context feature is sometimes called the Context object, but it is *not* a PowerBuilder system object.)

For more information about these objects, see *Objects and Controls* or the PowerBuilder Browser.

Enabling a service

Before you use a service, you instantiate it by calling the GetContextService function. When you call this function, PowerBuilder returns a reference to the instantiated service. Use this reference in dot notation when calling the service's functions.

❖ **To enable a service:**

1 Establish an instance variable of the appropriate type:

```
ContextInformation icxinfo_base
```

```
ContextKeyword     icxk_base
CORBACurrent       corbcurr_base
ErrorLogging       erl_base
Inet               iinet_base
SSLServiceProvider sslsp_base
TransactionServer  ts_base
```

2 Instantiate the instance variable by calling the GetContextService function:

```
this.GetContextService("ContextInformation",  &
   icxinfo_base)
this.GetContextService("ContextKeyword", icxk_base)
// Use Keyword instead of ContextKeyword on EAServer
this.GetContextService("Keyword", icxk_base)
this.GetContextService("CORBACurrent", &
   corbcurr_base)
this.GetContextService("ErrorLogging", erl_base)
this.GetContextService("Internet", iinet_base)
this.GetContextService("SSLServiceProvider", &
   sslsp_base)
this.GetContextService("TransactionServer",ts_base)
```

Using a CREATE statement

You can instantiate a service object with a PowerScript CREATE statement. However, this always creates an object for the default context (native PowerBuilder execution environment), regardless of where the application is running.

Context information service

You use the context information service to obtain information about an application’s execution context. The service provides current version information, as well as whether the application is running in the PowerBuilder execution environment and whether as a PowerBuilder window plug-in or a PowerBuilder window ActiveX. Using this information, you can modify display characteristics and application behavior. For example, you might hide a Close command button when running as a plug-in or ActiveX.

Additionally, when running in the PowerBuilder window ActiveX under Internet Explorer, the context information service can return a reference to an ActiveX automation server object. Your application can use this reference to call functions that control the Web browser.

Accessing context information

Using the context information service, you can access the information in Table 22-4.

Table 22-4: Context information

Item	Use this function	Comment
Full context name	GetName	Value returned depends on the context: • Default: PowerBuilder Runtime • Window plug-in: PowerBuilder window Plugin • Window ActiveX: PowerBuilder window ActiveX
Abbreviated context name	GetShortName	Value returned depends on the context: • Default: PBRUN • Window plug-in: PBWinPlugin • Window ActiveX: PBRTX
Company name	GetCompanyName	Returns Sybase, Inc.
Version	GetVersionName	Returns the full version number (for example, 11.0.0.1)
Major version	GetMajorVersion	Returns the major version number (for example, 11.0)
Minor version	GetMinorVersion	Returns the minor version number (for example, 0)
Fix version	GetFixesVersion	Returns the fix version number (for example, 1)

Using the ClassName function

You can also use the ClassName function to determine the context of the object. The return value of the ClassName function differs by context. For example, for the Window plug-in the return value is plugincontextinformation, and for the Window ActiveX it is rtxcontextinformation.

You can use this information for many purposes, including:

- **Modifying application appearance based on execution context** For example, you might hide a Close button when running in the PowerBuilder window plug-in and PowerBuilder window ActiveX. In a plug-in or ActiveX, closing the window results in a blank space in the HTML page.
- **Verifying that the context supports the current version** For example, if your application requires features or fixes from Version 11.0.0.1, you can use the context information service to check the version in the current execution context.

❖ To access context information:

1 Declare an instance or global variable of type ContextInformation:

```
ContextInformation icxinfo_base
```

2 Create the context information service by calling the GetContextService function:

```
this.GetContextService("ContextInformation", &
   icxinfo_base)
```

3 Call context information service functions as necessary.

This example calls the GetShortName function to determine the current context and the GetVersionName function to determine the current version:

```
String  ls_name
String  ls_version
Constant String ls_currver = "7.0.01"
icxinfo_base.GetShortName(ls_name)
IF ls_name <> "PBRun" THEN
   cb_close.visible = FALSE
END IF
icxinfo_base.GetVersionName(ls_version)
IF ls_version <> ls_currver THEN
   MessageBox("Error",  &
      "Must be at Version " + ls_currver)
END IF
```

Accessing the ActiveX automation server

If you are running the PowerBuilder window ActiveX in Internet Explorer Version 3.0 or later, your application can call the context information service's GetHostObject function to obtain a reference to an ActiveX automation server object (the hosting object). Specifically, if you pass an uninstantiated OLEObject variable to GetHostObject, it returns a reference to the IWebBrowserApp automation server.

Your application can call and access IWebBrowserApp methods and properties, which allow you to access and control certain aspects of browser behavior, including:

- Go back
- Go forward
- Go home
- Refresh
- Navigate to a specified Web page
- Exit the browser

For more information on the IWebBrowserApp interface, its methods, and its properties, see the Internet Explorer documentation or the Microsoft Web site.

❖ **To access the ActiveX automation server:**

1 Declare instance or global variables of type ContextInformation and OLEObject:

```
ContextInformation   icxinfo_base
OLEObject   iole_browser
```

2 Create the context information service by calling the GetContextService function:

```
GetContextService("ContextInformation",  &
   icxinfo_base)
```

3 Establish a reference to the ActiveX automation server by calling the GetHostObject function:

```
Integer li_rtrn
li_rtrn = icxinfo_base.GetHostObject(iole_browser)
IF li_rtrn = 1 THEN
   sle_host.Text = "GetHostObject succeeded"
ELSE
   sle_host.Text = "GetHostObject failed"
END IF
```

4 Call IWebBrowserApp functions as necessary. This example calls the Navigate function to open the default Web browser displaying the Sybase home page:

```
IF IsValid(iole_browser) THEN
   iole_browser.Navigate  &
      ("http://www.sybase.com", 0, 0, 0)
END IF
```

Context keyword service

Use the context keyword service to access environment information for the current context. In the default environment, this service returns host workstation environment variables. In the PowerBuilder window plug-in, this service allows you to access parameters specified in the plug-in’s Embed element. When running within EAServer, you can use the keyword service to get a specific component property value (on all EAServer platforms except AIX, you must use Keyword as the string parameter in the GetContextService function).

For information about using the context keyword service in EAServer, see “Accessing component properties” on page 491.

Accessing environment variables

When running in the PowerBuilder execution environment (the default context), you use this service to return environment variables.

❖ **To access environment variables:**

1. Declare an instance or global variable of type ContextKeyword. Also declare an unbounded array of type String to contain returned values:

```
ContextKeyword  icxk_base
String   is_values[]
```

2. Create the context information service by calling the GetContextService function:

```
this.GetContextService("Keyword", icxk_base)
```

3. Call the GetContextKeywords function to access the environment variable you want. This example calls the GetContextKeywords function to determine the current application *Path*:

```
icxk_base.GetContextKeywords("Path", is_values)
```

4. Extract values from the returned array as necessary. When accessing environment variables, the array should always have a single element:

```
MessageBox("Path", "Path is: " + is_values[1])
```

Accessing Embed element parameters

The Embed element can contain additional, user-specified, parameters. Additionally, the Embed element allows more than one value for each parameter.

When running in the PowerBuilder window plug-in context, you use this service to access parameters specified in the Embed element. If the specified parameter is not found, the service tries to match the specified parameter with an environment variable.

❖ **To access Embed element parameters:**

1. Declare an instance or global variable of type ContextKeyword. Also declare an unbounded array of type String to contain returned values:

```
ContextKeyword  icxk_base
String   is_values[ ]
```

2. Create the context information service by calling the GetContextService function:

```
GetContextService("Keyword", icxk_base)
```

3 Call the GetContextKeywords function. This example calls the GetContextKeywords function to access values for the user-specified parameter, VALID:

```
icxk_base.GetContextKeywords("VALID", is_values)
```

4 Extract values from the returned array as necessary. This example displays parameters in a list:

```
Integer   li_count
FOR li_count = 1 to UpperBound(is_values)
      lb_parms.AddItem(is_values[li_count])
NEXT
```

CORBACurrent service

Client applications and EAServer components marked as OTS style can create, control, and obtain information about EAServer transactions using functions of the CORBACurrent context service object. The CORBACurrent object provides most of the methods defined for the CORBA Current interface.

For more information, see "Client- and component-demarcated transactions" on page 528.

Error logging service

To record errors generated by PowerBuilder objects running in a transaction server to a log file, create an instance of the ErrorLogging service object and invoke its log method. For example:

```
ErrorLogging erlinfo_base
this.GetContextService("ErrorLogging",  &
   erlinfo_base)
erlinfo_base.log("Write this string to log")
```

The errors are recorded in the EAServer log if the component is running in EAServer, and in the Windows system application log if the component is running in COM+.

Internet service

Use the Internet service to:

- Display a Web page in the default browser (HyperLinkToURL function, which starts the default browser with the specified URL)
- Access the HTML for a specified page (GetURL function, which performs an HTTP Get)
- Send data to a CGI, ISAPI, or NSAPI program (PostURL function, which performs an HTTP Post)

Hyperlinking to a URL

You call the Internet service's HyperLinkToURL function to start the default browser with a specified URL.

❖ **To hyperlink to a URL:**

1 Declare an instance or global variable of type Inet:

```
Inet  iinet_base
```

2 Create the Internet service by calling the GetContextService function:

```
THIS.GetContextService("Inet", iinet_base)
```

3 Call the HyperLinkToURL function, passing the URL of the page to display when the browser starts:

```
iinet_base.HyperlinkToURL  &
   ("http://www.sybase.com")
```

Getting a URL

You call the Internet service's GetURL function to perform an HTTP Get, returning raw HTML for a specified URL. This function returns the raw HTML using the InternetResult object.

❖ **To perform an HTTP Get:**

1 Declare an instance or global variable of type Inet. Also declare an instance or global variable using the descendent InternetResult object as the datatype (n_ir_msgbox in this example):

```
Inet  iinet_base
n_ir_msgbox  iir_msgbox
```

2 Create the Internet service by calling the GetContextService function:

```
THIS.GetContextService("Internet", iinet_base)
```

3 Create an instance of the descendent InternetResult object:

```
iir_msgbox = CREATE n_ir_msgbox
```

4 Call the GetURL function, passing the URL of the page to be returned and a reference to the instance of the descendent InternetResult object:

```
iinet_base.GetURL  &
   ("http://www.sybase.com", iir_msgbox)
```

When the GetURL function completes, it calls the InternetData function defined in the descendent InternetResult object, passing the HTML for the specified URL.

Posting to a URL

You call the Internet service's PostURL function to perform an HTTP Post, sending data to a CGI, ISAPI, or NSAPI program. This function returns the raw HTML using the InternetResult object.

❖ **To perform an HTTP Post:**

1 Declare an instance or global variable of type Inet. Also declare an instance or global variable using the descendent InternetResult object as the datatype (n_ir_msgbox in this example):

```
Inet  iinet_base
n_ir_msgbox  iir_msgbox
```

2 Create the Internet service by calling the GetContextService function:

```
THIS.GetContextService("Internet", iinet_base)
```

3 Create an instance of the descendent InternetResult object:

```
iir_msgbox = CREATE n_ir_msgbox
```

4 Establish the arguments to the PostURL function:

```
Blob    lblb_args
String  ls_headers
String  ls_url
Long    ll_length
ls_url = "http://coltrane.sybase.com/"
ls_url += "cgi-bin/pbcgi80.exe/"
ls_url += "myapp/n_cst_html/f_test?"
lblb_args = Blob("")
ll_length = Len(lblb_args)
ls_headers = "Content-Length: "  &
   + String(ll_length) + "~n~n"
```

5 Call the PostURL function, passing the URL of the routine to be executed, the arguments, the header, an optional server port specification, and a reference to the instance of the descendent InternetResult object:

```
iinet_base.PostURL  &
   (ls_url, lblb_args, ls_headers, 8080, iir_msgbox)
```

When the PostURL function completes, it calls the InternetData function defined in the descendent InternetResult object, passing the HTML returned by the specified routine.

Using the InternetResult object

The GetURL and PostURL functions both receive data in an InternetResult object. This object acts as a buffer, receiving and caching the asynchronous data as it is returned by means of the Internet. When all data is received, the InternetResult object calls its InternetData function, which you override to process the data as appropriate.

Implement in descendants of InternetResult

You implement this feature by creating standard class user objects of type InternetResult. In each of these descendent user objects, define an InternetData function to process the passed HTML as appropriate.

❖ To implement a descendent InternetResult object:

1. Create a standard class user object of type InternetResult.
2. Declare a new user object function as follows:
   - **Name** InternetData
   - **Access** Public
   - **Returns** Integer
   - **Argument name** Data, passed by value
   - **Argument datatype** Blob
3. Add code to the InternetData function that processes the returned HTML as appropriate. This example simply displays the HTML in a MessageBox:

```
MessageBox("Returned HTML", &
   String(data, EncodingANSI!))
Return 1
```

Secure Sockets Layer service

PowerBuilder allows you to establish Secure Sockets Layer (SSL) connections to EAServer. The SSL protocol allows connections to be secured using public-key encryption and authentication algorithms that are based on digital certificates. SSL is a "wrapper" protocol: packets for another protocol are secured by embedding them inside SSL packets. For example, HTTPS is HTTP secured by embedding each HTTP packet within an SSL packet. Likewise, IIOPS is IIOP embedded within SSL.

You use an instance of the SSLServiceProvider object to establish the connection from the client to the server. For more information, see the *PowerScript Reference* and Chapter 26, “Using SSL in PowerBuilder clients.”

Transaction server service

Use the transaction server service to access information about the context of an object running in a transaction server such as EAServer or COM+. You can use the TransactionServer object to influence transaction behavior programmatically, and to access the methods of another component on the transaction server.

For more information, see Chapter 24, “Building an EAServer Component” and Chapter 27, “Building a COM or COM+ Component.”

PART 6

Developing Distributed Applications

This part describes tools and techniques for building distributed applications with PowerBuilder.

PowerBuilder Enterprise edition only
The tools for building multitier applications available with PowerBuilder are supported in the Enterprise edition only.

CHAPTER 23

Distributed Application Development with PowerBuilder

About this chapter

This chapter gives an overview of distributed application development with PowerBuilder.

PowerBuilder Enterprise edition only
Distributed application development is supported in the Enterprise edition only.

Contents

Distributed application architecture

Distributed application development, also called multitier development, offers a natural way to separate the user interface components of an application from the business logic that the application requires. By centralizing business logic on a middle-tier server, you can reduce the workload on the client and control access to sensitive information.

In a distributed application, the client and server work together to perform tasks for the business user. The client handles all interactions with the user while the middle-tier server provides background services to the client. Typically, the middle-tier server performs most of the processing and database access. To invoke the services of the server, the client calls a method (or function) associated with a component (or object) that resides on the server.

Partitioned applications

Client-side logic for enterprise applications must be as small and efficient as possible to conserve network bandwidth. To accomplish this goal, applications are partitioned into three parts: presentation, business logic, and database access. The database resides on the bottom tier of the enterprise system to maintain and secure the organization's information assets. The business logic resides in the middle tier or server. The presentation is on the user's desktop, or top tier, or is dynamically downloaded to the user's desktop.

The server is then responsible for executing and securing the vast majority of a corporation's business logic. This makes it a critical component in the network-centric architecture. The client communicates with the server, calling middle-tier components that perform business logic.

Web application architecture

A Web application is a variation of the distributed architecture where the client is hosted in a Web browser. PowerBuilder provides several technologies for building Web applications, including .NET Web Forms deployment, JSP targets, and the Web DataWindow, all of which provide a thin client solution. The architecture of your application varies depending on which technologies you decide to use.

For more information, see Chapter 30, “Web Application Development with PowerBuilder.”

Server support

PowerBuilder developers can build clients that invoke the services of Sybase EAServer, COM+, and other application servers, and build components (or objects) that execute business logic inside each of these servers.

PowerBuilder also provides support for building clients for Enterprise JavaBeans components (EJBs) running on any J2EE-compliant server.

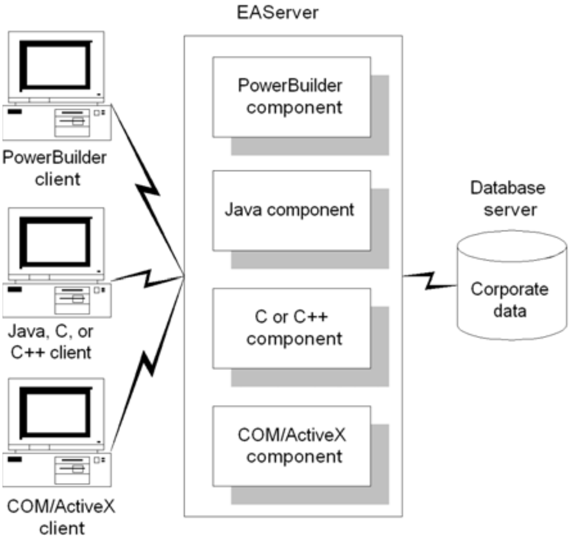
EAServer

PowerBuilder and EAServer are fully integrated. A PowerBuilder application can act as a client to any EAServer component. In addition, EAServer can contain PowerBuilder custom class user (nonvisual) objects that execute as middle-tier components.

EAServer hosts the PowerBuilder virtual machine natively. This means that EAServer can communicate directly with PowerBuilder nonvisual user objects, and vice versa. EAServer components developed in PowerBuilder can take full advantage of the ease of use and flexibility of PowerScript and the richness of PowerBuilder’s system objects.

Components developed in PowerBuilder can exploit features such as transactions, interoperability, and instance pooling. As shown in Figure 23-1, any type of client can access any type of component running in EAServer, regardless of the language used to develop the component.

Figure 23-1: Clients and components in EAServer

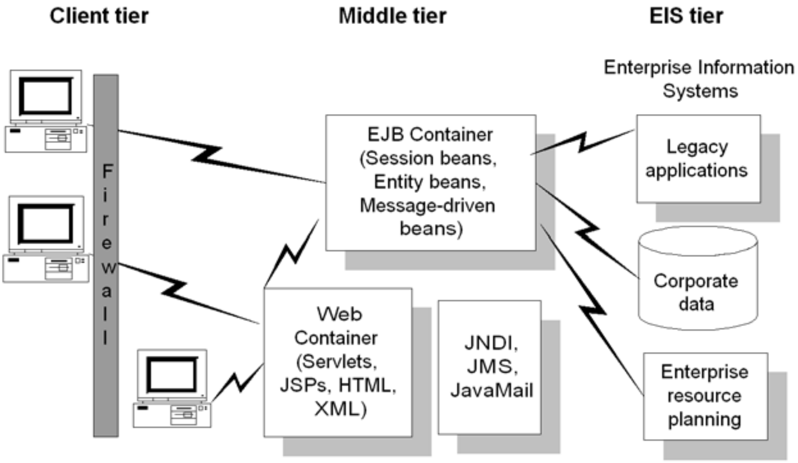


For more information, see Chapter 24, “Building an EAServer Component” and Chapter 25, “Building an EAServer Client.”

J2EE servers

J2EE, the Java 2 Platform, Enterprise Edition, is the official Java framework for enterprise application development. A J2EE application is composed of separate components that are installed on different computers in a multitiered system. Figure 23-2 shows three tiers in this system: the client tier, middle tier, and Enterprise Information Systems (EIS) tier. The middle tier is sometimes considered to be made up of two separate tiers: the Web tier and the business tier.

Figure 23-2: J2EE client, middle, and EIS tiers


Client tier
Middle tier
EIS tier
Enterprise Information Systems
F i r e w a l l
EJB Container (Session beans, Entity beans, Message-driven beans)
Web Container (Servlets, JSPs, HTML, XML)
JNDI, JMS, JavaMail
Legacy applications
Corporate data
Enterprise resource planning

Client components, such as application clients and applets, run on computers in the client tier. Web components, such as Java servlets and JavaServer Pages (JSP) components, run on J2EE servers in the Web tier. Enterprise JavaBeans (EJB) components are business components and run on J2EE servers in the business tier. The EIS tier is made up of servers running relational database management systems, enterprise resource planning applications, mainframe transaction processing, and other legacy information systems.

In PowerBuilder, you can build client applications that use the services of EJB components running on any J2EE-compliant server. For more information, see Chapter 29, “Building an EJB client.”

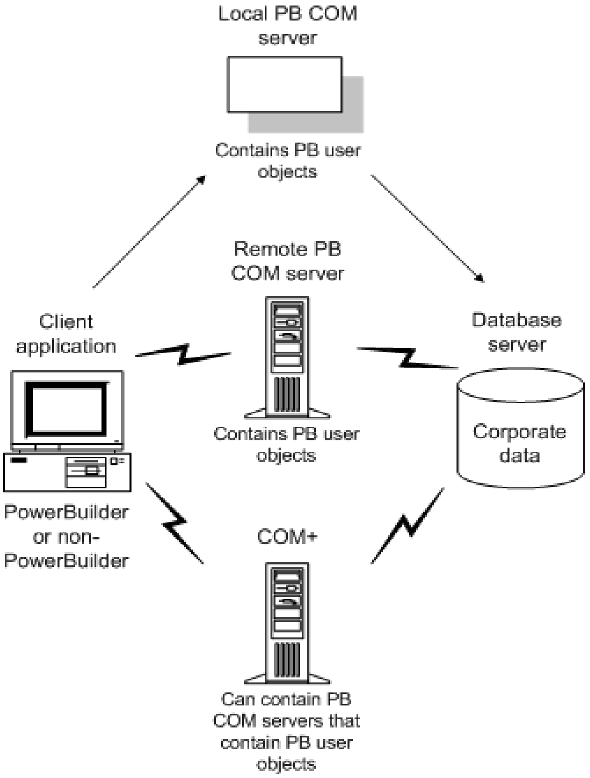
Plug-in for J2EE servers

You can also deploy custom class user objects to third-party application servers on which the PowerBuilder Application Server Plug-in is installed. The Plug-in is a Sybase product that supports several Web servers. Wizards that help you generate application server components that you can deploy to these servers and proxies that you can use to build client applications are built into PowerBuilder, but you must purchase the Plug-in product separately. The wizards and techniques are very similar to those used for building EAServer components and clients. For more information, see the documentation for the PowerBuilder Application Server Plug-in on the Product Manuals Web site at http://www.sybase.com/support/manuals/.

COM+

A PowerBuilder application can act as a client to a COM server. The server can be built using PowerBuilder or any other COM-compliant application development tool and it can run locally, on a remote computer as an in-process server, or in COM+, as shown in Figure 23-3.

Figure 23-3: PowerBuilder clients for COM components



You can develop a custom class user object containing business logic in PowerBuilder and then package the object as a COM object. A PowerBuilder COM server can include one or more PowerBuilder custom class user objects. You code the user objects in the User Object painter and then build the server in the Project painter. You can also deploy the COM server directly to a local COM+ server or create a COM+ package from the Project painter.

For more information, see Chapter 27, “Building a COM or COM+ Component” and Chapter 28, “Building a COM or COM+ Client.”

CHAPTER 24 Building an EAServer Component

About this chapter

This chapter explains how to use PowerBuilder to build an EAServer component.

Contents

About building an EAServer component

PowerBuilder provides tools for developing custom class (nonvisual) user objects and deploying them as EAServer components. You can deploy these components to an EAServer host running on Windows, UNIX, and Linux operating systems. See “Deploying a component to EAServer” on page 507.

Limitations on UNIX
If you plan to deploy components to a UNIX or Linux server, you should be aware that the PowerBuilder runtime libraries on these platforms do not support graphical operations or calls to the Windows application programming interface.

About using the wizards

PowerBuilder provides several wizards to facilitate the development and deployment of EAServer components. All of the wizards build an EAServer project:

- **Target wizard** Creates a new target, a new application object, a new custom class user object, and a new EAServer project
- **Object wizard** Creates a new custom class user object in an existing target and a new EAServer project
- **Project wizard** Creates an EAServer project in which you select one or more existing custom class user objects

About the development process

Steps for building EAServer components

To build and deploy an EAServer component from a custom class user object, complete the following steps:

1. Use the EAServer Component Target Wizard to create a new user object in a new target. Alternatively, if you are working in an existing target, you can use the EAServer Component Object wizard to create the object. These wizards also allow you to enter information exposing the new user object as a Web service.
2. Add functions, events, and instance variables to the generated user object in the User Object painter.
3. Test and debug the object.
4. Deploy the object to EAServer.

To test or deploy an EAServer component that you developed in PowerBuilder, create a project object and build the project. You can create a project object from the Target, Object, or Project wizard.

To deploy a component, open the project in the Project painter, optionally modify the project settings, and build the project. When you do this, the EAServer component generator deploys the component interface and the PowerBuilder implementation of that interface to the target server.

For testing purposes, you can use live editing to build the project automatically from the User Object painter. This removes the need to build the project from the Project painter. When live editing is enabled in the User Object painter, PowerBuilder builds the project for an EAServer component each time you save the user object. For more information on live editing, see “Testing and debugging the component” on page 496.

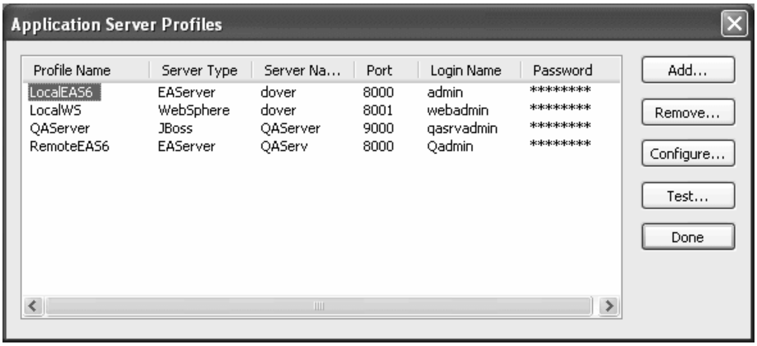
To-Do List

When you create a new user object by using the EAServer Target or Object wizard, you can optionally create a To-Do List. If you check the Generate To-Do List box on the last page of the wizard, the wizard adds tasks to the To-Do List to remind you to complete all phases of development.

Creating an application server profile

An application server profile is a named set of parameters stored in your system registry that defines a connection to a particular EAServer or third-party application server host. Before you use a wizard to create a component, you should create a profile for the server where the component will be deployed.

The Application Server Profiles dialog box lists your defined profiles. You create, edit, delete, and test EAServer profiles from this dialog box.



Profile Name	Server Type	Server Na...	Port	Login Name	Password
LocalEAS6	EAServer	dover	8000	admin	********
LocalWS	WebSphere	dover	8001	webadmin	********
QAServer	JBoss	QAServer	9000	qasrvadmin	********
RemoteEAS6	EAServer	QAServ	8000	Qadmin	********

In the EAServer and Application Server Component and Proxy wizards, there is a Manage Profiles button on the Choose EAServer Profile or Choose Application Server Profile page. Clicking this button opens the Application Server Profiles dialog box so that you can add a new profile or modify an existing profile without leaving the wizard.

The Profile Name in the Edit Application Server Profile dialog box cannot be edited. This is because the name is stored in the project object along with the other properties of the profile. If the profile name cannot be found in the registry when the project is deployed, the description in the project object is used.

❖ **To create an application server profile:**

1. Click the Application Server Profile button in the PowerBar.

   The Application Server Profiles dialog box displays, listing your configured profiles.

2. Select Add.

   The Edit Application Server Profile dialog box displays.

3. Type the profile name, server name, port number, login name, and password (if required).

4. (Optional) Select Test to verify the connection.

5. Click OK to save your changes and close the dialog box.

   The Application Server Profiles dialog box displays, with the new profile name listed. The Application Server profile values are saved in the registry in *HKEY_CURRENT_USER/Software/Sybase/PowerBuilder/110/JaguarServerProfiles*.

Working with shared and service components

When you create an EAServer component in PowerBuilder, the wizard offers you a choice of creating a standard, shared, or service component.

About shared components

How EAServer manages program variable space

The EAServer architecture is component-oriented. Each component maintains its own state. When a single client instantiates several PowerBuilder objects on the server, EAServer maintains program variable space for the objects separately. Each PowerBuilder user object running in EAServer has its own copy of the global and shared variables. The PowerBuilder objects share no common state. They can communicate only through methods, EAServer shared components, server files, and databases.

To allow clients to share state information, EAServer provides support for shared components. Shared components allow multiple clients to share the same component instance.

EAServer Manager and the Management Console
In EAServer 5.x, you use EAServer Manager to manage properties in EAServer. In EAServer 6.x, you use the EAServer Manager plug-in the Sybase Management Console.

Marking a component as shared in a PowerBuilder wizard is equivalent to marking it as shared on the General page of the component's properties in the Management Console or the Instances page in EAServer Manager. Only a single instance of the component can be instantiated in EAServer.

Clients (and other server components) access a shared component as if it were any other kind of component.

Benefits of using EAServer shared components

Shared components allow you to:

- Provide convenient access to common data that would otherwise need to be retrieved separately by each client connection
- Reduce the number of database accesses, allowing the database server to be available for other processing

EAServer shared components versus PowerBuilder shared objects

EAServer shared components offer many of the same benefits as PowerBuilder shared objects. PowerBuilder components that you deploy to EAServer can act as clients to EAServer shared components, as well as to PowerBuilder shared objects. EAServer shared components can also be accessed by components and clients that are not implemented in PowerBuilder.

However, EAServer does not treat PowerBuilder shared objects as if they were EAServer shared components. Therefore, the functions used to manipulate PowerBuilder shared objects (SharedObjectRegister, SharedObjectGet, and so forth) do not work with EAServer shared components. If you try to call one of these functions inside a PowerBuilder component running in EAServer, the request will fail.

About service components

A service component performs background processing for EAServer clients and other EAServer components. EAServer loads service components at server start-up time.

When you mark a component as a service component in one of the PowerBuilder wizards, PowerBuilder installs the component as a service in EAServer at deployment time.

In EAServer 5.x, this is equivalent to adding the component to the list of services for the server by modifying its com.sybase.EAServer.server.services property.

In EAServer 6.x, properties are specified in Ant configuration files. For example, the service components that are installed with EAServer are configured in the default-service-components file in the *EAServer6\bin* directory.

For information about creating and configuring service components in EAServer 6.x, see the *Automated Configuration Guide* on the Sybase Product Manuals Web site at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.dc00548_0600/html/easautoconfig/X1201010.htm.

Shared or not shared

When you create a service component, the wizard marks the component as shared. If you want more than one instance of a service component, you can change that setting in the Project painter. Select the number of instances you want in the Create Instances spin control on the Components page of the EAServer Component Generator property sheet in the Project painter. Notice that when you change the number of instances to a number greater than one, the Concurrency and Automatic Demarcation/Deactivation are checked. This is to ensure that the component does not encounter threading issues in EAServer. For more information, see "Threading issues and component types" next.

Functions for service components

The PowerBuilder wizards include three additional functions for a service component. These functions are defined in the CTSServices::GenericServices interface and allow you to control the behavior of background processes associated with the service:

- **Start** EAServer calls the Start function after the service component has been loaded. You can add logic to this function to perform start-up initialization for the service.

- **Run** EAServer calls the Run function after the first invocation of the Start function returns. The Run function allows you to perform repetitive tasks as a background process. The Run function must periodically suspend its own execution by calling the **JagSleep** C function. The JagSleep function frees the CPU to perform other tasks. To use the JagSleep function, declare an external function for JagSleep in PowerBuilder. Here is the syntax to use for the function declaration:

```
subroutine JagSleep (Long seconds) LIBRARY
   "libjdispatch.dll"
```

- **Stop** This function allows you to stop the execution of the background process coded in the Run function. You can implement a service-manager client for your service that calls Stop, Start, and Run so that the service can be restarted without restarting EAServer. The script for the Stop function can also clean up any resources that were allocated in the Start function.

Threading issues and component types

Each instance of a PowerBuilder component executes in its own session, and each session can support only one thread of execution. Therefore a single PowerBuilder component instance cannot simultaneously execute multiple client requests. However, multiple instances of the same component can each execute a separate client request. When you create a component, PowerBuilder sets default values for several properties that affect the way threads are handled in EAServer.

Using the Thread Manager
You can also use the EAServer Thread Manager to develop more robust services. See “Using the EAServer Thread Manager” on page 462.

Concurrency property

The Concurrency property determines whether multiple instances of a component can be created to handle multiple client requests. Checking the Concurrency check box in the wizard or the Project painter sets the com.sybase.jaguar.component.thread.safe property to TRUE.

Standard components For standard components, you can improve performance by allowing multiple instances of a component to handle client requests. The default setting for the concurrency property for standard components is checked, but you can change the setting if you want only one instance of a given component.

Shared components For shared components, only one instance of the component should be active at any time and therefore only one thread can be executed. The Concurrency check box is disabled and not checked for shared components.

Service components Although service components are usually treated as shared components, you can choose to create more than one instance of a service component to improve performance and scalability. There are three options on the Components page that interact for service components: Concurrency, Automatic Demarcation/Deactivation, and Create Instances (the Create Instances option can be changed only for service components).

When you change the Create Instances option to 2 or more, the Concurrency check box and the Automatic Demarcation/Deactivation check box become checked. Multiple instances of the service component can be created if necessary, and are deactivated after each method call. If you clear the Automatic Demarcation/Deactivation check box, so that you need to explicitly deactivate component instances, the Create Instances check box is reset to 1 and the Concurrency check box is cleared.

bind.thread, sharing, and tx_vote properties

There are three other component properties that affect the handling of threads in EAServer: sharing, tx_vote, and bind.thread.

Bind Object property not used
An additional property, bind.object, enables client threads to execute in a single instance but also supports creation of multiple instances. This property *cannot* be used for PowerBuilder components and is always set to FALSE.

When the *bind.thread* property is set to TRUE, a method on a component instance must execute on the same thread that created the instance. This property must be set to TRUE if you are using live editing to build your component. It should be set to FALSE for components that are deployed to UNIX servers to improve scalability.

The *sharing* property identifies whether or not the component is shared. It is set to FALSE when you select Standard in the wizard and to TRUE when you select Shared or Service. The only way this property can be changed in PowerBuilder is by changing the Create Instances setting on the Components tab page for service components in the Project painter. If either the sharing or thread.safe property is set to TRUE, the other must be set to FALSE.

A component that can remain active between consecutive method invocations is called a stateful component. A component that is deactivated after each method call and that supports instance pooling is said to be a stateless component. Typically, an application built with stateless components offers the greatest scalability. The tx_vote property determines whether the component is deactivated after every method call. It is set to FALSE (stateless) if you check the Automatic demarcation/deactivation check box in the wizard or select more than one instance on the Component page in the Project painter; otherwise it is set to TRUE (stateful). You can have only one instance of a stateful service object.

Table 24-1 summarizes the default setting for each type of component and shows which can be changed.

Table 24-1: Thread-handling properties

Component	bind.thread	sharing	thread.safe	tx_vote
Standard	FALSE, mutable	FALSE, immutable	TRUE, mutable	FALSE, mutable
Shared	FALSE, mutable	TRUE, immutable	FALSE, immutable	FALSE, mutable
Service (single instance)	FALSE, mutable	TRUE, immutable	FALSE, mutable	FALSE, mutable
Service (multiple instances)	FALSE, mutable	FALSE, immutable	TRUE, mutable	FALSE, mutable If changed to TRUE, number of instances is set to 1, sharing to TRUE, and thread.safe to FALSE.

If you deploy a service component for which bind.thread, thread.safe, and sharing are set to TRUE, EAServer disables the thread.safe property automatically at runtime.

Using the EAServer Thread Manager

The Thread Manager is a built-in EAServer component that allows you to run EAServer component instances in threads that execute independently of client method invocations. You can use threads spawned by the Thread Manager to perform any processing that must occur asynchronously with respect to user interaction.

For example, you might have a component method that begins a lengthy file indexing operation. The method could call the Thread Manager to start the processing in a new thread, then return immediately.

Since each instance of a PowerBuilder component executes in its own session, and each session can support only one thread of execution, you cannot develop a service that can be stopped or refreshed without using the Thread Manager. In the service's start or run method, spawn threads that do the service's processing. In the service's stop method, call the Thread Manager stop method to halt the threads.

For a more complete description of the Thread Manager, see the EAServer documentation.

Providing support for instance pooling

Benefits of instance pooling

EAServer components can optionally support instance pooling. **Instance pooling** allows EAServer clients to reuse component instances. By eliminating the resource drain caused by repeated allocation of component instances, instance pooling improves the overall performance of EAServer.

Specifying pooling options in the wizards

When you create an EAServer component using one of the PowerBuilder wizards, you have the option to specify one of the pooling options for the component shown in Table 24-2.

Table 24-2: EAServer component pooling options

Pooling option	Description
Supported	The component is *always* pooled after each client use. When this option is selected, the CanBePooled event is not triggered for the component. This option has the effect of setting the component's pooling property to TRUE. If the Automatic Demarcation/Deactivation setting for the component is enabled, instances are pooled after each method invocation. If the setting is disabled, instances are pooled when the component calls the SetComplete (or SetAbort) method of the TransactionServer context object.
Not supported	By default, the component is not pooled after each client use. However, you can override the default behavior by scripting the CanBePooled event. In the CanBePooled event, you can specify programmatically whether a particular component instance should be pooled. If you script the CanBePooled event, this event is triggered after each client use. This option has the effect of setting the component's pooling property to FALSE.

Controlling the state of a pooled instance

When you create an EAServer component that supports instance pooling, that component may need to reset its state after each client has finished using the pooled instance.

To allow you to control the state of a component, EAServer triggers one or more of the events shown in Table 24-3 during the lifecycle of the component.

Table 24-3: Component state events

Event	PBM code
Activate	PBM_COMPONENT_ACTIVATE
CanBePooled	PBM_COMPONENT_CANBEPOOLED
Deactivate	PBM_COMPONENT_DEACTIVATE

When the component's pooling option is set to Supported (the pooling property is set to TRUE), you may need to script the Activate and Deactivate events to reset the state of the pooled component. This is necessary if the component maintains state in an instance, shared, or global variable.

When the component's pooling option is set to Not Supported (the pooling property is set to FALSE), you can optionally script the CanBePooled event to specify whether a particular component instance should be pooled. If you script the CanBePooled event, you may also need to script the Activate and Deactivate events to reset the state of the pooled component. If you do not script the CanBePooled event, the component instance is not pooled.

The EAServer Component Target and Object wizards automatically include the Activate and Deactivate events to a custom class user object that will be deployed as an EAServer component. If you want to script the CanBePooled event, you need to add this event yourself. If you do this, be sure to map the event to the correct PBM code.

Constructor and Destructor are fired once

When instance pooling is in effect, the Constructor and Destructor events are fired only once for the component. The Constructor and Destructor events are not fired each time a new client uses the component instance. Therefore, to reset the state of a component instance that is pooled, add logic to the Activate and Deactivate events, not the Constructor and Destructor events.

Instance pool timeout

Instance pooling can decrease client response time, but can also increase memory usage in the server. You can specify how long, in seconds, an instance can remain idle in the pool. The default is 600 (ten minutes). To free resources used by idle component instances, the server may remove instances that remain idle past this time limit.

You can set environment variables to configure the way memory is managed in PowerBuilder and EAServer. For more information, see “Configuring memory management” on page 47 and the technical document EAServer/PowerBuilder Memory Tuning and Troubleshooting at http://www.sybase.com/detail?id=1027319.

The lifecycle of a component

To understand how instance pooling works, you need to understand the lifecycle of a component instance. This is what happens during the component lifecycle:

1. The component is typically instantiated on the first method invocation. When this occurs on a component developed in PowerBuilder, EAServer creates a new PowerBuilder session for the component to run in.
2. The PowerBuilder session creates the instance of the PowerBuilder nonvisual object that represents the EAServer component. Creating the object causes the **Constructor event** to be fired.
3. After the object has been instantiated, EAServer triggers the **Activate event** on the nonvisual object to notify the object that it is about to be used by a new client. At this point, the component must ensure that its state is ready for execution.
4. EAServer then executes the method called by the client on the component.

5 When the component indicates that its work is complete, EAServer triggers the **Deactivate event** to allow the component to clean up its state. If the Automatic Demarcation/Deactivation setting for the component is enabled, the Deactivate event is triggered automatically after each method invocation. If the setting is disabled, the Deactivate event is triggered when the component calls the SetComplete (or SetAbort) method of the TransactionServer context object.

6 If you have selected the Not Supported pooling option (or set the component's pooling property to FALSE) and also scripted the **CanBePooled event**, EAServer triggers this event to ask the component whether it is able to be pooled at this time. The CanBePooled event allows the component instance to selectively enable or refuse pooling.

The return value of the CanBePooled event determines whether the component instance is pooled. A return value of 1 enables pooling; a return value of 0 disables pooling. If the CanBePooled event has not been scripted, then by default the instance is not pooled.

What happens when the pooling property is enabled
When you select the Supported pooling option (or set the component's pooling property to TRUE), component instances are *always* pooled and the CanBePooled event is never triggered.

7 If an instance is not pooled after deactivation, EAServer triggers the **Destructor event**. Then it destroys the PowerBuilder object and terminates the runtime session.

Providing support for transactions

Benefits of EAServer's transaction support

EAServer components that you develop in PowerBuilder can participate in EAServer transactions. An **EAServer transaction** is a transaction whose boundaries and outcome are determined by EAServer. You can mark components to indicate that they will provide transaction support. When a component provides transaction support, EAServer ensures that the component's database operations execute as part of a transaction.

Multiple EAServer components can participate in a single EAServer transaction; EAServer ensures that database changes performed by the participating components are all committed or rolled back. By defining components to use EAServer transactions, you can ensure that all work performed by components that participate in a transaction occurs as intended.

Indicating how the component will support transactions

Each EAServer component has a transaction attribute that indicates how the component participates in EAServer transactions. When you develop an EAServer component in PowerBuilder, you can specify the transaction attribute in the wizards. Table 24-4 lists the options.

Table 24-4: Transaction attribute options

Transaction type	Description
Not supported	The component never executes as part of a transaction. If the component is activated by another component that is executing within a transaction, the new instance's work is performed outside the existing transaction.
Supports Transaction	The component can execute in the context of an EAServer transaction, but a transaction is not required to execute the component's methods. If the component is instantiated directly by a client, EAServer does not begin a transaction. If component A is instantiated by component B and component B is executing within a transaction, component A executes in the same transaction.
Requires Transaction	The component always executes in a transaction. When the component is instantiated directly by a client, a new transaction begins. If component A is activated by component B and B is executing within a transaction, A executes within the same transaction; if B is not executing in a transaction, A executes in a new transaction.
Requires New Transaction	Whenever the component is instantiated, a new transaction begins. If component A is activated by component B, and B is executing within a transaction, then A begins a new transaction that is unaffected by the outcome of B's transaction; if B is not executing in a transaction, A executes in a new transaction.
Mandatory	Methods can be invoked only by a client that has an outstanding transaction. Calling this component when there is no outstanding transaction generates a runtime error.

Transaction type	Description
OTS Style	The component can manage transactions. It can inherit a client's transaction. If called without a transaction, the component can explicitly begin, commit, and roll back transactions using an instance of the CORBACurrent context service object.
Never	Methods cannot be invoked when there is an outstanding transaction. Calling this component when there is an outstanding transaction generates a runtime error.

Using the transaction service context object

Component methods can call EAServer's transaction state primitives to influence whether EAServer commits or aborts the current transaction. To give you access to EAServer's transaction state primitives, PowerBuilder provides a transaction service context object called TransactionServer.

If you plan to use the TransactionServer context object, you should set the UseContextObject DBParm parameter to Yes.

For transactional components, setting UseContextObject to Yes tells PowerBuilder that you will be using the methods of the TransactionServer object rather than COMMIT and ROLLBACK to indicate whether the component has completed its work for the current transaction. If your scripts contain COMMIT and ROLLBACK statements, they will generate a runtime error. Setting UseContextObject to No causes COMMIT and ROLLBACK statements to call the EAServer transaction service's CommitWork and AbortWork methods. You should only use this setting if you want to retain legacy code and you do not want to use the TransactionServer object.

For components that do not need to be in a transaction, the UseContextObject setting is ignored and PowerBuilder drivers handle COMMIT and ROLLBACK statements.

To use the transaction context service, declare a variable of type TransactionServer and call the GetContextService function to create an instance of the service.

Example In the Activate (or Constructor) event for a component, you can call GetContextService to instantiate the TransactionServer service:

```
// Instance variable:
// TransactionServer ts

Integer li_rc
li_rc = this.GetContextService("TransactionServer", &
   ts)
```

```
IF li_rc <> 1 THEN
   // handle the error
END IF
```

In one of the component methods, you can then update the database and call SetComplete if the update succeeds or SetAbort if it fails:

```
//Instance variable:
//DataStore ids_datastore
long ll_rv
...
...
ll_rv = ids_datastore.Update()
IF ll_rv = 1 THEN
   ts.SetComplete()
ELSE
   ts.SetAbort()
END IF
```

The TransactionServer interface provides the methods in Table 24-5 to allow you to access EAServer's transaction primitives.

Table 24-5: TransactionServer methods

Method	Description
DisableCommit	Indicates that the current transaction cannot be committed because the component's work has not been completed. The instance remains active after the current method returns.
EnableCommit	Indicates that the component should not be deactivated after the current method invocation; allows the current transaction to be committed if the component instance is deactivated.
IsInTransaction	Determines whether the current method is executing in a transaction.
IsTransactionAborted	Determines whether the current transaction has been aborted.
SetAbort	Indicates that the component cannot complete its work for the current transaction and that the transaction should be rolled back. The component instance will be deactivated when the method returns.
SetComplete	Indicates that the component has completed its work in the current transaction and that, as far as it is concerned, the transaction can be committed and the component instance can be deactivated.

Automatic Demarcation/ Deactivation

If you want a component to be automatically deactivated after each method invocation, you can enable Automatic Demarcation/Deactivation for the component. This sets the component's tx_vote property to FALSE. When Automatic Demarcation/Deactivation is enabled, you do not need to make explicit calls to SetComplete to cause deactivation because SetComplete is assumed by default. To roll back the transaction, you can call SetAbort.

If you do not want the component to be automatically deactivated after each method invocation, disable the Automatic Demarcation/Deactivation setting for the component. This sets the component's tx_vote property to TRUE. When you disable Automatic Demarcation/Deactivation, EAServer waits for notification before completing transactions; therefore, be sure to deactivate programmatically by making an explicit call to SetComplete (or SetAbort).

COMMIT and ROLLBACK

You have the option to disable the TransactionServer context object and use the COMMIT and ROLLBACK statements instead to specify the EAServer transaction state for a component. This capability is provided to allow you to migrate PowerBuilder 6 objects to EAServer without modifying the code. To disable the TransactionServer context object, set the **UseContextObject** DBParm parameter to No. When you do this, COMMIT is equivalent to SetComplete and ROLLBACK is equivalent to SetAbort.

COMMIT and ROLLBACK in nontransactional components

In nontransactional components that disable the TransactionServer context object, COMMIT does not invoke SetComplete and ROLLBACK does not invoke SetAbort. For example, if you specify Not Supported as the transaction type, disable Automatic Demarcation/Deactivation (set tx_vote to TRUE), and set the UseContextObject parameter to No, the PowerBuilder virtual machine does *not* issue a SetComplete when you execute a COMMIT (or a SetAbort when you execute a ROLLBACK). In this case, EAServer never releases the component because it is waiting for a call to SetComplete or SetAbort.

If you disable Automatic Demarcation/Deactivation for a component that performs no database access whatsoever, then you must use the TransactionServer object to call SetComplete (or SetAbort) to deactivate the component. Otherwise, the component will never be deactivated.

Transaction handling and runtime errors

You can control the behavior of EAServer when an internal exception occurs in the PBVM or a PowerBuilder component raises a runtime exception. To do so, set the PBOnFatalError or PBRollbackOnRTError environment variables in a batch file or as a system environment variable on the server on which the component runs.

Table 24-6: Environment variables for handling exceptions

Variable	Description
PBOnFatalError	Specifies whether EAServer should continue, shut down, or restart when an internal exception occurs in the PBVM. The default behavior is that EAServer shuts down. An unhandled internal exception raised by a PowerBuilder component running in EAServer can cause the PBVM to become unstable, resulting in unpredictable behavior. Values are: • continue – EAServer continues to run, and the CORBA_TRANSACTION_ROLLEDBACK exception is thrown • restart – EAServer restarts automatically • shutdown – EAServer shuts down automatically (default)
PBRollbackOnRTError	Specifies how a transaction is handled when a runtime exception is raised by a PowerBuilder component running in EAServer. By default, the transaction is rolled back and the exception is thrown back to the client. Values are: • n, no, or false – the transaction is committed before the exception is thrown back to the client • y, yes, or true – the transaction is rolled back before the exception is thrown back to the client (default)

Transactions and the component lifecycle

EAServer's transaction model and the component lifecycle are tightly integrated. Component instances that participate in a transaction are never deactivated until the transaction ends or until the component indicates that its contribution to the transaction is over (its work is done and ready for commit or its work must be rolled back). An instance's time in the active state corresponds exactly to the beginning and end of its participation in a transaction.

For more information, see the EAServer documentation.

Accessing a database from an EAServer component

Database connectivity

You can access a database from an EAServer component. If you want to take advantage of EAServer's support for connection pooling and transaction management, you need to use one of the database interfaces supported by EAServer to connect to your database. For more information about EAServer database connections for components developed in PowerBuilder, see *Connecting to Your Database*.

Using DataStores

EAServer components developed in PowerBuilder can use **DataStores** to interact with the database. DataStores are nonvisual DataWindow controls. DataStores act just like DataWindow controls except that they do not have visual attributes.

DataStores can be useful in a distributed application: they give you the ability to perform database processing on a remote server instead of on each client machine.

RichText presentation style is not supported
A server component cannot contain a DataStore that has a DataWindow object that uses the RichText presentation style. Rich text processing is not supported in distributed applications.

Sharing data between the server and the client

If you want to provide a visual interface to the data retrieved on the server, include a window in the client that has a DataWindow control. Whenever data is retrieved on the server, refresh the DataWindow control to show the result set for the DataStore on the server. Similarly, whenever the user makes modifications to the data on the client, refresh the contents of the DataStore on the server to reflect the current state of the DataWindow control on the client.

To share data between a client and a server, synchronize the server DataStore and the client DataWindow control programmatically. If you want your application to handle database updates, this involves moving the DataWindow data buffers and status flags back and forth between the client and the server.

For more information about synchronizing a server DataStore with a client DataWindow, see “Performing updates” on page 476.

ShareData function is not supported in distributed applications
You cannot use the ShareData function to share data between a DataWindow control on a client and a DataStore on a server.

Using connection caching

Benefits of connection caching

To optimize database processing, EAServer provides support for connection caching. Connection caching allows EAServer components to share pools of preallocated connections to a remote database server, avoiding the overhead imposed when each instance of a component creates a separate connection. By establishing a connection cache, a server can reuse connections made to the same data source. Connection caches are called data sources in EAServer 6.x.

How it works

Ordinarily, when a PowerBuilder application connects to a database, PowerBuilder physically terminates each database connection for which a DISCONNECT statement is issued. By contrast, when a PowerBuilder component uses an EAServer connection cache, EAServer logically terminates the database connection but does not physically remove the connection. Instead, the database connection is kept open in the connection cache so that it can be reused for other database operations.

Do not disconnect in destructor event
EAServer releases all connection handles to the cache when a transaction is completed or when the component is deactivated. If you place a DISCONNECT statement in the destructor event, which is triggered after the deactivate event, the connection has already been logically terminated and the DISCONNECT causes a physical termination. DISCONNECT statements can be placed in the deactivate event.

All connections in a cache must share a common user name, password, server name, and connectivity library.

Accessing a cache by user

If you want to retrieve a connection from the cache that uses a specified set of user name, password, server, and connectivity library values, you do not need to modify your database access code to enable it to use the cache. You simply need to create a new cache in EAServer Manager that has the database connection properties (user name, password, server name, and connectivity library) required by the component. In EAServer 6.x, you create a data source (cache) by selecting Resources>Data Sources>Add in the Management Console. At runtime, when the component tries to connect to the database, EAServer automatically returns a connection from the cache that matches the connection values requested by the component.

Accessing a cache by name

If you want to retrieve a connection from a cache by specifying the cache name, set the CacheName DBParm to identify the cache you want to use. Accessing a cache by name allows you to change the user name, password, or server in the Management Console without requiring corresponding changes to your component source code.

This code for a PowerBuilder component shows how to access a cache by name:

```
SQLCA.DBMS = "ODBC"
SQLCA.Database = "EAS Demo DB"
SQLCA.AutoCommit = FALSE
SQLCA.DBParm = "ConnectString='DSN=EAS Demo DB;
   UID=dba;PWD=sql',CacheName='mycache'"
```

Cache names are case-sensitive
Cache names are case-sensitive; therefore, make sure the case of the cache name you specify in your script matches the case used for the name in EAServer.

Retrieving a connection by proxy

Regardless of whether you access a cache by user or name, you can retrieve a connection by proxy. Retrieving a connection by proxy means that you can assume the identity and privileges of another user by providing an alternative login name.

This feature can be used with any database that recognizes the SQL command **set session authorization**. In order for user A to use the ProxyUserName DBParm to assume the identity of another user B, user A must have permission to execute this statement. For example, for SQL Anywhere, user A must have DBA authority, and for ASE, user A must have been granted permission to execute set session authorization by a System Security Officer.

For more information about the PowerBuilder database interfaces that support proxy connections, see *Connecting to Your Database*.

To use proxy connections, set the **ProxyUserName** DBParm to identify the alternative login name. This example shows how to retrieve a connection by proxy:

```
SQLCA.DBMS = "ODBC"
SQLCA.DBParm = "CacheName='MyEAServerCache',
   UseContextObject='Yes',ProxyUserName='pikachu'"
```

Before you can use a connection by proxy

Set-proxy support must be enabled in the cache properties file before components can take advantage of it. In EAServer 5.x, EAServer Manager does not automatically create an individual cache properties file when you create a cache, so you must create this file manually. Name the file *cachename.props* and put it in the *EAServer\Repository\ConnCache* directory. Once you have created the cache properties file, add the following line:

```
com.sybase.jaguar.conncache.ssa=true
```

For this setting to take effect, you must refresh EAServer.

In EAServer 6.x, you create a data source by selecting Resources>Data Sources>Add in the Management Console. Select Set Session Authorization and specify a name in the Set Session Authorization System ID box. The properties file for the data source is stored in the Repository in the *Instance\com\sybase\djc\sql\DataSource* directory.

For more information on managing connection caches (or data sources), see the EAServer documentation.

You must also set up your database server to recognize and give privileges to the alternative login name defined in the ProxyUserName DBParm.

What happens when all connections are in use

You can control what happens if all connections in a cache are in use. To do this, set the **GetConnectionOption** DBParm to one of the following values:

Value	Description
JAG_CM_NOWAIT	Causes the attempt to connect to fail with an error if no connection can be returned.
JAG_CM_WAIT	Causes the component to wait until a connection becomes available.
JAG_CM_FORCE	Allocates and opens a new connection. The new connection is not cached and is destroyed when it is no longer needed.

By default, PowerBuilder uses JAG_CM_FORCE.

What happens when a connection is released

You can also control what happens when a connection is released. To do this, set the **ReleaseConnectionOption** DBParm to one of the following values:

Value	Description
JAG_CM_DROP	Closes and deallocates the connection. If the connection came from a cache, a new connection is created in its place. Use JAG_CM_DROP to destroy a connection when errors have made it unusable.
JAG_CM_UNUSED	If the connection was taken from a cache, it is placed back in the cache. A connection created outside of a cache is closed and destroyed.

By default, PowerBuilder uses JAG_CM_UNUSED.

EAServer connection caches for Unicode support

The following EAServer native connection caches support Unicode connections for PowerBuilder components.

For EAServer 5.x:

- OCI_9U – Oracle9*i* Unicode Cache
- OCI_10U – Oracle 10*g* Unicode Cache
- ODBCU – ODBC Unicode Cache

For EAServer 6.x:

- JCM_Oracle_Unicode – Oracle9*i* or 10*g* Unicode Cache
- JCM_Odbc_Unicode – ODBC Unicode Cache

These connection cache types accept Unicode connection parameters and then send a request to the database driver to open a Unicode connection to the database. With a Unicode connection, PowerBuilder components can communicate with the database using Unicode.

If you are using the Oracle9*i* native interface (O90) to access an Oracle9*i* database in a PowerBuilder component in EAServer 5.x, use the database driver type OCI_9U for the connection cache. If you do not, access will fail.

For an ODBC connection cache in EAServer 5.x, use the database driver type ODBCU to access multiple-language data in a SQL Anywhere Unicode database or DBCS data in a SQL Anywhere DBCS database and set the database parameter ODBCU_CONLIB to 1. For example:

```
SQLCA.DBParm = "CacheName='EASDemo_u',
   UseContextObject='Yes',ODBCU_CONLIB=1"
```

Performing retrieval operations

To use a DataStore to perform retrieval operations, you first need to create an instance of the DataStore object in a script and assign the DataWindow object to the DataStore. Then set the Transaction object for the DataStore. Once these setup steps have been performed, you can retrieve data into the DataStore, print the contents of the DataStore, or perform other processing against a retrieved result set.

Example: passing an array by reference

Description

This example demonstrates the use of a DataStore to retrieve data in a server component. The server component uo_customers has a function called retrieve_custlist. retrieve_custlist generates an instance of the DataStore ds_datastore and then uses this DataStore to retrieve all of the rows in the Customer table. Once the data has been retrieved, retrieve_custlist passes the data back to the client application.

Function declaration

The retrieve_custlist function has an argument called customers, which is defined as an array based on the structure st_custlist. The structure st_custlist has the same layout as d_custlist, the DataWindow object used to access the database. The return value for retrieve_custlist, which is used to return the number of rows retrieved, is of type Long.

Here is the signature of the retrieve_custlist function:

retrieve_custlist(REF st_custlist customers []) returns long

Script

Here is the script for the retrieve_custlist function:

```
datastore ds_datastore
long ll_rowcount

ds_datastore = create datastore
ds_datastore.dataobject = "d_custlist"
ds_datastore.SetTransObject (SQLCA)

IF ds_datastore.Retrieve() <> -1 THEN
   ll_rowcount = ds_datastore.RowCount()
END IF

customers = ds_datastore.object.data
destroy ds_datastore

return ll_rowcount
```

At the conclusion of processing, the function retrieve_custlist destroys the DataStore and returns the number of rows retrieved back to the client.

Performing updates

DataWindow synchronization

In a conventional client/server application, where database updates are initiated by a single application running on a client machine, PowerBuilder can manage DataWindow state information for you automatically. In a distributed application, the situation is somewhat different. Because application components are partitioned between the client and the server, you need to write logic to ensure that the data buffers and status flags for the DataWindow control on the client are synchronized with those for the DataStore on the server.

PowerBuilder provides four functions for synchronizing DataWindows and DataStores in a distributed application:

- GetFullState

- SetFullState
- GetChanges
- SetChanges

Although these functions are most useful in distributed applications, they can also be used in nondistributed applications where multiple DataWindows (or DataStores) must be synchronized.

Moving DataWindow buffers and status flags

To synchronize a DataWindow control on the client with a DataStore on the server, move the DataWindow data buffers and status flags back and forth between the client and the server whenever changes occur. The procedures for doing this are essentially the same whether the source of the changes resides on the client or the server.

To apply *complete state information* from one DataWindow (or DataStore) to another, you need to:

1. Invoke the GetFullState function to capture the current state of the source DataWindow.
2. Invoke the SetFullState function to apply the state of the source DataWindow to the target.

To apply *changes* from one DataWindow (or DataStore) to another, you need to:

1. Invoke the GetChanges function to capture changes from the source DataWindow.
2. Invoke the SetChanges function to apply changes from the source DataWindow to the target.

SetChanges can be applied to an empty DataWindow
You can call SetChanges to apply changes to an empty DataWindow (or DataStore). The target DataWindow does not need to contain a result set from a previous retrieval operation. However, the DataWindow must have access to the DataWindow definition. This means that you need to assign the DataWindow object to the target DataWindow before calling SetChanges.

DataWindow state is stored in blobs

When you call GetFullState or GetChanges, PowerBuilder returns DataWindow state information in a Blob. The Blob returned from GetFullState provides everything required to recreate the DataWindow, including the data buffers, status flags, and complete DataWindow specification. The Blob returned from GetChanges provides data buffers and status flags for changed and deleted rows only.

Synchronizing after Update

By default, the Update function resets the update flags after a successful update. Therefore, when you call the Update function on the server, the status flags are automatically reset for the server DataStore. However, the update flags for the corresponding client DataWindow control are *not* reset. Therefore, if the Update function on the server DataStore succeeds, call ResetUpdate on the client DataWindow to reset the flags.

One source, one target

You can synchronize a single source DataWindow (or DataStore) with a single target DataWindow (or DataStore). *Do not try to synchronize a single source with multiple targets, or vice versa.*

Typical usage scenario

Suppose the server has a component that uses a DataStore called DS_1. This DataStore is the source of data for a target DataWindow called DW_1 on the client. In the Activate event, the component connects to the database, creates a DataStore, and assigns the DataWindow object to the DataStore.

In one of its methods, the server component issues a Retrieve function for DS_1, calls GetFullState on DS_1, and then passes the resulting Blob to the client. Because the component's Automatic Demarcation/Deactivation setting is disabled (the component is stateful), it also calls SetComplete before the method returns to cause the component instance to be deactivated.

If Automatic Demarcation/Deactivation were enabled

If the Automatic Demarcation/Deactivation setting were enabled for the component, it would not need to call SetComplete after the retrieval because the component instance would automatically be deactivated when the method finished execution.

Once the client has the DataWindow Blob, it calls SetFullState to apply the state information from the Blob to DW_1. At this point, the user can insert new rows in DW_1 and change or delete some of the existing rows. When the user makes an update request, the client calls GetChanges and invokes another component method that passes the resulting Blob back to the server. The component method then calls SetChanges to apply the changes from DW_1 to DS_1. After synchronizing DS_1 with DW_1, the server component updates the database and calls SetComplete or SetAbort to indicate whether the update was successful.

If the update was successful, the client calls ResetUpdate to reset the status flags on the client DataWindow.

After the completion of the first update operation, the client and server can pass change Blob results (rather than complete state information) back and forth to handle subsequent updates. From this point on, the update process is an iterative cycle.

Example

The following example shows how you might synchronize DataWindows between a PowerBuilder client and an EAServer component. This example uses a stateless component.

Client window definition

Suppose the client has a window called w_employee that has buttons that allow the user to retrieve and update data. The Retrieve button on the client window has the following script:

```
// Global variable:
// connection myconnect
// Instance variable:
// uo_employee iuo_employee

blob lblb_data
long ll_rv

myconnect.CreateInstance(iuo_employee)
iuo_employee.RetrieveData(lblb_data)

ll_rv = dw_employee.SetFullState(lblb_data)

if ll_rv = -1 then
   MessageBox("Error", "SetFullState call failed!")
end if
```

The Update button on the client window has the following script:

```
blob lblb_data
long ll_rv

ll_rv = dw_employee.GetChanges(lblb_data)

if ll_rv = -1 then
   MessageBox("Error", "GetChanges call failed!")
else
   if iuo_employee.UpdateData(lblb_data) = 1 then &
      dw_employee.ResetUpdate()
end if
```

Server object definition

The server has an object called uo_employee that has the following functions:

- RetrieveData
- UpdateData

Instance variables

The uo_employee object has these instance variables:

```
protected TransactionServer ts
protected DataStore ids_datastore
```

Activate

The Activate event for the uo_employee object instantiates the TransactionServer service. In addition, it connects to the database and creates the DataStore that will be used to access the database:

```
this.GetContextService("TransactionServer", ts)
SQLCA.DBMS="ODBC"
SQLCA.DBParm="ConnectString=
   'DSN=EAS Demo DB;UID=dba;PWD=sql',
   UseContextObject='Yes'"
CONNECT USING SQLCA;
IF SQLCA.SQLCode < 0 THEN
   //Handle the error
END IF
ids_datastore = CREATE datastore
ids_datastore.dataobject = "d_emplist"
ids_datastore.SetTransObject (SQLCA)
```

Script for the RetrieveData function

The RetrieveData function takes an argument called ablb_data, which is a Blob passed by reference. The function returns a Long value.

Here is the script for the RetrieveData function:

```
long ll_rv
ids_datastore.Retrieve()
ll_rv = ids_datastore.GetFullState(ablb_data)
ts.SetComplete()
return ll_rv
```

Script for the UpdateData function

The UpdateData function takes an argument called ablb_data, which is a Blob passed by reference. The function returns a Long value.

Here is the script for the UpdateData function:

```
long ll_rv
if ids_datastore.SetChanges(ablb_data) = 1 then
   ll_rv = ids_datastore.Update()
end if
if ll_rv = 1 then
   ts.SetComplete()
else
```

```
       ts.SetAbort()
    end if
    return ll_rv
```

Deactivate

The Deactivate event for the uo_employee object destroys the DataStore and disconnects from the database:

```
DESTROY ids_datastore
DISCONNECT USING SQLCA;
```

Passing result sets

PowerBuilder provides two system objects to handle getting result sets from components running in EAServer and returning result sets from PowerBuilder user objects running as EAServer components. These system objects, ResultSet and ResultSets, are designed to simplify the conversion of transaction server result sets to and from DataStore objects and do not contain any state information. They are not designed to be used for database updates. You use the CreateFrom and GenerateResultSet functions on the DataStore object to convert the result sets stored in these objects to and from DataStore objects.

About GenerateResultSet
GenerateResultSet has an alternative syntax used for returning a Tabular Data Stream result set when using MASP (Method as Stored Procedure) with EAServer. For more information, see the *DataWindow Reference*.

Component methods that return result sets use the TabularResults module. Single result sets are returned as TabularResults::ResultSet structures. Multiple result sets are returned as a sequence of ResultSet structures using the TabularResults::ResultSets datatype.

Accessing result sets in EAServer components from PowerBuilder clients

When you generate an EAServer proxy object in PowerBuilder for an EAServer component method that returns TabularResults::ResultSet, the method on the proxy object returns a PowerBuilder ResultSet object. Methods that return multiple result sets return a PowerBuilder ResultSets object.

Viewing proxies in the Browser
You can view the properties and methods of EAServer proxy objects on the Proxy tab in the PowerBuilder Browser.

You can access the result set from a PowerBuilder client by creating an instance of the component, calling the method, and then using the result set to populate a DataStore object with the CreateFrom function.

This example creates an instance of the SVUBookstore component and calls the GetMajors method:

```
SVUBookstore lcst_mybookstore
resultset lrs_resultset
datastore ds_local
integer li_rc

// myconnect is a Connection object
li_rc = myconnect.CreateInstance(lcst_mybookstore)
IF li_rc <> 0 THEN
   MessageBox("Create Instance", string(li_rc) )
   myconnect.DisconnectServer()
   RETURN
END IF

lrs_resultset = lcst_mybookstore.GetMajors()
ds_local = CREATE datastore
ds_local.CreateFrom(lrs_resultset)
```

Returning result sets from EAServer components

To pass or return result sets from a PowerBuilder user object that will be deployed to EAServer, set the datatype of a function's argument or return value to ResultSet (for a single result set) or ResultSets (for multiple result sets). When the user object is deployed as an EAServer component, the ResultSet and ResultSets return values are represented in the IDL interface of the component as TabularResults::ResultSet and TabularResults::ResultSets datatypes.

In this example, a DataStore object is created and data is retrieved into it, and then the GenerateResultSet function is used to create a result set that can be returned to a client:

```
datastore ds_datastore
resultset lrs_resultset
integer li_rc

ds_datastore = create datastore
ds_datastore.dataobject = "d_empdata"
ds_datastore.SetTransObject (SQLCA)
IF ds_datastore.Retrieve() = -1 THEN
   // report error and return
END IF
li_rc = ds_datastore.generateresultset(lrs_resultset)
```

```
IF li_rc <> 1 THEN
   // report error and return
END IF
return lrs_resultset
```

Defining the component interface

How the interface is specified

EAServer stores all component interfaces in CORBA Interface Definition Language (IDL) modules. IDL is defined by the Object Management Group as a standard language for defining component interfaces. When you deploy a PowerBuilder custom class user object as an EAServer component, the methods (functions and events) and instance variables defined for the object are added to the component interface. You do not need to write IDL for the interface, because the EAServer component generator writes the IDL for you.

In EAServer 6.0 and later, PowerBuilder components are wrapped as EJBs. For more information, see the *CORBA Components Guide* in the EAServer documentation set on the Sybase Product Manuals Web site.

What gets included in the interface

The EAServer component generator includes all public functions declared for the user object in the component interface. Depending on the build options you specify for the component, the generator may also include accessor methods for the public instance variables and also expose user events as methods.

Method names and method overloading

Although IDL does not provide support for method overloading, you can nonetheless deploy PowerBuilder custom class user objects to EAServer that have overloaded methods. To work around the IDL restriction, the component generator appends two underscores (__) and a unique suffix to the method name that will be overloaded. If you look at the IDL generated for a PowerBuilder object, you therefore see suffixes appended to methods that were overloaded in PowerBuilder.

When you generate stubs or proxy objects for components that have overloaded methods, EAServer strips off the IDL suffix so that the client can access the method by using its correct name.

For more information about IDL, see the EAServer documentation.

Do not use two consecutive underscores in your method names
Because EAServer treats two underscores (__) as a reserved delimiter, you should not use two consecutive underscores in a function name in a custom class user object that you plan to deploy as an EAServer component.

Datatypes

You can use the following datatypes in the interface of a user object that you deploy as an EAServer component:

- Standard datatypes (except for the Any datatype)
- Structures
- Custom class (nonvisual) user objects that have been deployed as EAServer components

These datatypes can be used for public instance variables as well as for the arguments and return values of public methods. Private and protected instance variables and methods can use all datatypes supported by PowerBuilder.

The Any datatype is not supported in the public interface of a component. In addition, with the exception of the ResultSet and ResultSets objects, the component interface cannot include built-in PowerBuilder system objects (for example, the Transaction or DataStore object). The component interface also cannot include visual objects (such as windows or menus).

Component methods can pass arrays of standard datatypes and arrays of structures, and they can use custom class user objects to pass arrays.

int and uint map to short
In EAServer 6.x, both the int and uint PowerBuilder datatypes map to short, so if you have defined a function that returns int and has an int argument, deployment will fail if you define a function with the same name on the same component that returns uint and has a uint argument.

For a list of datatypes used in EAServer, their CORBA IDL equivalents, and the PowerBuilder datatypes to which they map, see the *PowerScript Reference* or the online Help. For a list of PowerBuilder to EJB datatype mappings, see the *CORBA Components Guide* in the EAServer documentation set on the Sybase Product Manuals Web site.

Passing by reference

You can pass arguments to component methods by reference. However, the behavior is somewhat different in a distributed application than in a nondistributed application.

When you pass by reference, the variable is actually copied to the server before the method is executed and then copied back when the method completes execution. This behavior is usually transparent to the application, but in some situations it can affect the outcome of processing.

For example, suppose you define a method called increment_values that takes two arguments called x and y, both of which are passed by reference. The script for the method increments x and y as shown below:

```
x = x + 1
y = y + 1
```

The client uses the following code to call the method:

```
int z
z = 1
increment_values(z,z)
```

In a nondistributed application, the value of z after the method completed execution would be 3 (because the local invocation passes a *pointer* to z, and z is incremented twice). In a distributed application, the value of z would be 2 (because the remote invocation passes two *copies* of z, which are incremented separately).

Passing a read-only value

When you pass a read-only value, the behavior is similar to passing by value, except that the data cannot be modified. A copy of the data is passed across the wire to the server.

Passing objects

Objects created within EAServer components can be passed back to clients, but these objects must be installed EAServer components. If you try to pass back a PowerBuilder object that is not an EAServer component, you will get a runtime error. To use a component that was passed back from the server, the client must have the corresponding EAServer proxy (for a PowerBuilder client) or stub (for a non-PowerBuilder client).

A client application *cannot* pass a PowerBuilder object reference to EAServer. Therefore, you cannot use a PowerBuilder object reference to push messages from the server back to a PowerBuilder client. However, you can simulate this behavior by using a shared object on the client to communicate with EAServer.

To simulate server push, the client uses the SharedObjectRegister and SharedObjectGet functions to create a shared object. Once the object has been created, the client can post a method to the shared object, passing it a callback object that should be notified when processing has finished on the server. The method on the shared object makes a synchronous call to the EAServer component method that performs processing. Since the shared object is running in a separate thread on the client, the client application can proceed with other work while the process is running on the server.

Providing support for NULL values

PowerBuilder allows you to specify whether the methods of an EAServer component can accept NULL values as function arguments or return types. To provide support for NULL values in the component interface, check the Support NULL Values check box in the property sheet for the project used to generate the EAServer component. If this box is not checked, clients cannot pass NULL values in any argument and the server cannot set any argument to NULL or return a NULL value.

If you allow null values in the prototype for a component method, PowerBuilder appends a "_N" suffix to the method name in the EAServer proxy that you generate from the Project painter. To call this method, you must create an instance of the proxy, rather than an instance of the NVO, and you must reference the method with the "_N" suffix. For example, if of_gen is the name of a method in the NVO, and you create an EAServer proxy that allows null return values, you must instantiate the proxy and call of_gen_N to use this method.

EAServer validation

If you are designing a custom class user object that you plan to deploy as an EAServer component, you can have PowerBuilder warn you when you use code elements that are not valid in EAServer. EAServer validation checks public instance variables and public functions for system types, visual types, structures, and any variables.

EAServer validation is on by default if you created the user object using an EAServer wizard. To check, select the Design menu in the User Object painter and make sure EAServer Validation is checked. When you save the object, the Output window lists warnings such as the following:

```
---------- Compiler: Information messages
Information C0197: Component Validation
Warning    C0198: illegal Jaguar type: 'window' return
type for function: 'of_badfunc'
Warning     C0198: illegal Jaguar type: 'any' return
type for function: 'of_badfunc'
```

Validation is associated with the object you are editing, not with the User Object painter. When you reopen an object, it has the same validation state as when you closed it.

Throwing exceptions

When you declare an exception on a function of a user object deployed to EAServer, the exceptions are translated to CORBA IDL as part of the method prototype. The exceptions can be handled by any type of EAServer client application or calling component. For more information, see “Exception handling in PowerBuilder” on page 36.

Implementing an existing interface

You can create PowerBuilder implementations of existing interfaces using the EAServer Component Wizard on the Target or PB Object tab in the New dialog box. A typical use of this feature is to create an implementation of a standard API, such as protocols for online banking and securities trading.

Selecting an interface

On the Specify Interface Implementation page in the wizard, select Implement an Existing EAServer Remote Interface, then select the EAServer profile for the server that contains the IDL interface you want to implement. You can select only one interface from the list that displays when you expand the list of packages in the wizard.

For PowerBuilder components, the interface name is usually the same as the component name, but the list of interfaces does not map directly to the list of components on the server. The list includes all IDL modules of type interface.

Setting options in the wizard

Once you have selected the interface to implement, you can enter the EAServer name for the component. The name of the PowerBuilder custom class user object cannot be changed—it is always the same as the name of the remote interface. You can set most other options, such as package name, instance pooling, and so forth, as if you were creating a new interface.

If you are building a PowerBuilder implementation of a standard API, you will usually use the component name of the remote component, but you should not use the same package name.

Since the interface of the remote component cannot be changed, options that would change method signatures, such as supporting NULL values for arguments, cannot be set in the wizard.

Editing the user object in the painter

In the custom class user object created by the wizard, public attributes of the remote interface are represented as public instance variables, and public methods as public functions. The scripts for functions contain return statements so that they do not produce compilation errors, but you need to provide a script to implement each function. If the remote interface includes other dependencies, such as structures, the wizard creates them in the same PBL as the user object.

If you are using EAServer 6.0 or later, PowerBuilder components are wrapped as EJB components, and a Remove method is generated by EAServer as part of the component interface. You do not need to use this method.

You can edit the user object just as you would any other custom class user object—the User Object painter does not impose any restrictions. However, you should not make any changes that affect its interface. You should not delete any instance variables and functions that correspond to attributes and methods in the existing interface or change their mode from public to private or protected. Functions cannot be overloaded and the return value or arguments cannot be NULL values.

Deploying the component to EAServer

The project created by the wizard contains information about the interface from which the wizard built the component. When you run the project, PowerBuilder checks that:

- All public attributes and methods in the existing IDL interface are defined as public instance variables and functions in the user object.
- No methods defined in the IDL interface are overloaded in the user object.

If one of these checks fails, the component is deployed but a warning displays in the Project painter and the Output window.

Using a different project

These checks are performed only if the component is deployed using the project that was created when the component was created. If you create a new project or add the component to another project, no checks are performed when you run the project.

When you deploy using the project created with the component, the new implementation always uses the existing IDL on the server. You should be cautious if you use a different project, because you will be able to deploy the component to the original package and overwrite the existing IDL without seeing any warnings about changes in the interface.

Generating proxies
When you generate a proxy for an object that implements an existing interface and uses the existing IDL on the server, the proxy is based on the existing IDL. As a result, if you select the Prepend EAServer Package Name to Object Name option, the name prepended to the object name will be the name of the IDL module, not the new package name.

Invoking another server component's methods

EAServer allows the methods of one server component to call methods of another server component. The other server component does not need to be another PowerBuilder component; it can be implemented in any language supported by EAServer.

Accessing a component in the current server

To access methods of another EAServer component in the current server, you can use the Connection object to communicate with the component, just as you would from a PowerBuilder client. Alternatively, you can use the transaction service context object that PowerBuilder provides called **TransactionServer**. The TransactionServer interface provides a method called CreateInstance that allows you to access other components that are available locally. CreateInstance uses the same user and password information that applies to the component from which it is called.

Before you can use the transaction context service, declare a variable of type TransactionServer and call the GetContextService function to create an instance of the service. You must use a proxy object for your intercomponent calls. Without a proxy object, the TransactionServer object cannot obtain the correct method names of the component you are calling.

For information on creating a proxy object for EAServer components, see "Generating EAServer proxy objects" on page 516.

Example In the Activate event for a component, you can call GetContextService to instantiate the TransactionServer service:

```
// Instance variable:
// TransactionServer ts

Integer rc
rc = this.GetContextService("TransactionServer", ts)
IF rc <> 1 THEN
```

```
   // handle the error
END IF
```

In one of the component methods, you can then call CreateInstance to instantiate the second component and call one of its methods. Your application should include a proxy for the second component:

```
// Instance variable for the second component:
// nvo_comp2 mycomp2
Integer rc
rc = ts.CreateInstance(mycomp2, "mypackage/nvo_comp2")
IF rc <> 0 THEN
   // handle the error
ELSE
   mycomp2.method1()
END IF
```

Accessing a component in a different server

The procedure for accessing a server component on a different server is essentially the same as the procedure for accessing a server component from a PowerBuilder client. To access an EAServer component on a different server, create a Connection object, set properties for the Connection object, and call ConnectToServer.

Accessing an EJB component

A PowerBuilder component can access an EJB component using the Lookup method of either the Connection or TransactionServer objects. The Lookup method on the TransactionServer object has an optional third argument you can use to specify the name of the home interface. You use this argument only if the home interface name does not follow typical naming conventions.

Example This script instantiates the Cart component and invokes several component methods. In this example, the second argument to the Lookup method specifies the component name as well as the EAServer package name:

```
//Instance variable:
//Connection myconnect

CartHome MyCartHome // EJB's home interface
Cart MyShoppingCart // EJB's remote interface
TransactionServer ts
long ll_result

This.GetContextService("TransactionServer", ts)

//Get the home interface
ll_result = &
ts.Lookup(MyCartHome, "Shopping/Cart")
```

```
//Get a reference to Cart component's business logic
MyShoppingCart = MyCartHome.Create()

//Use the shopping cart
MyShoppingCart.AddItem(66)
MyShoppingCart.Purchase()
```

For information about accessing EJB components from PowerBuilder clients, see “Invoking an EJB component method” on page 520.

Component-demarcated transactions

EAServer components marked as OTS style can create, control, and obtain information about EAServer transactions using functions of the CORBACurrent context service object. The CORBACurrent object provides most of the methods defined for the CORBA Current interface.

For more information, see “Client- and component-demarcated transactions” on page 528.

Accessing component properties

ContextKeyword service object

You can use the ContextKeyword service object to get a specific component property value from the component’s *.properties* file. To retrieve a property value, you call the GetContextKeywords function (to enumerate all the properties of an EAServer component, use the Jaguar::Repository API).

Before you can use the ContextKeyword service object, create a reference to the object by calling the GetContextService function, using Keyword as the service name.

EAServer properties for PowerBuilder

Table 24-7 lists component properties that pertain to PowerBuilder custom class user objects running as EAServer components. Component properties are prefixed with the string com.sybase.jaguar.component.

The values of all component properties are displayed on the All Properties tab of the Component Properties dialog box in EAServer Manager in EAServer 5.x. Some properties also map to items on other tabs in this dialog box as shown in Table 24-7.

For descriptions of the properties in EAServer 6.x, see the *CORBA Components Guide* on the Sybase Product Manuals Web site at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.dc00547_0600/html/eascorba/BABCDICB.htm.

Table 24-7: EAServer component properties for PowerBuilder components in EAServer 5.x

Property	Description	Where displayed
auto.failover	Enables client proxies for the component to transfer to alternative servers when a server becomes unavailable. This property cannot be enabled unless you have enabled automatic demarcation/deactivation. Automatic failover requires that your application use a cluster of servers, so that redundant servers are available to run the application's components. The cluster must include at least one name server and clients must resolve proxy references using naming services. See "Load Balancing, Failover, and Component Availability" in the *EAServer System Administration Guide* for more information.	Transactions (Automatic Failover)
bind.thread	Indicates whether the component instance must always be invoked on the creating thread. Valid values are TRUE and FALSE. This property must be set to TRUE for live editing but should be set to FALSE otherwise to improve scalability.	Instances (Bind Thread)
code.set	Specifies the name of the coded character set used by the component. By default, the component uses the server's coded character set (specified on the General tab in the Server Properties window). For European or Asian languages, you may need to set this property to a value such as iso_1 or big5.	All Properties
interfaces	Identifies the interfaces that the component implements. This is a comma-separated list of IDL interface names, each of which takes the form: `module::interface.`	All Properties
minpool	When instance pooling is enabled, specifies the minimum number of instances that can be pooled. To release idle pooled instances, EAServer has a garbage collector thread that runs periodically. Each time it runs, the garbage collector removes one idle instance from the pool, unless the minimum pool size has been reached. The default is 0.	Resources
maxpool	When instance pooling is enabled, specifies the maximum number of instances that can be pooled. If the maximum pool size is reached, EAServer destroys excess instances after deactivation. The default is 0, which means no maximum pool size is in effect.	Resources
name	Specifies the name of the component. The value must take the form: `package/component`	General (component part only)
pb.appname	Specifies the name of the PowerBuilder application.	General
pb.class	Specifies the name of the PowerBuilder custom class user object.	General

Property	Description	Where displayed
pb.cookie	Provides a number used to construct the path for the library list. The path takes this form: *Repository\Component\package\component\Ccookie*	All Properties
pb.debug	Indicates whether you can debug the component in the PowerBuilder debugger.	All Properties
pb.librarylist	Specifies the PowerBuilder library list. When a library name has a prepended dollar sign ($), EAServer assumes that its location is relative to the EAServer Repository directory. When a library name does not have a prepended dollar sign, EAServer assumes the name specifies an absolute path.	General
pb.live_edit	Specifies whether you can build the project from the User Object painter instead of the Project painter. See "Live editing" on page 497.	All Properties
pb.trace	Specifies trace options for logging activity against the component (currently disabled).	All Properties
pb.version	Specifies the version of PowerBuilder in which the component was built.	All Properties
pooling	Indicates whether the component is pooled. When the pooling property is set to TRUE, the component is always pooled and the CanBePooled event is never triggered. When the pooling property is set to FALSE, the CanBePooled event is triggered, allowing selective refusal of pooling. If the tx_vote property is set to FALSE, the component is pooled after each method; otherwise, it is pooled at the end of the transaction.	Instances
sharing	Indicates whether this is a shared component. When the sharing property is set to TRUE, all clients share a single component instance. Pooling options do not apply to shared components. To make a shared component a service, add it to the list of services specified for the com.EAServer.server.services property.	Instances
state	When using automatic persistence, specifies the name of an IDL type. For PowerBuilder, the named type is a user-defined structure that must encapsulate all data to be persisted. When you select Automatic Persistent State, enter the name of the structure in the State text box on the Persistence tab page, and click OK, default values are supplied for other properties on the page. Stateful failover is supported for PowerBuilder components when using automatic persistence. For more information, see the chapter on managing persistent component state in the *EAServer Programmer's Guide*.	Persistence

Property	Description	Where displayed
state.gs	The names of methods in the state structure that get and set the state datatype, specified as a 2-item comma-separated list. The default is: `getState, setState`	Persistence (State Methods)
stateless	Applies only to EJB session Beans and non-EJB components that use the control interface CtsComponents::ObjectControl. Setting this property has the same effect as setting the tx_vote property to FALSE, but it also disables the activate and deactivate events. Do *not* set this property if you want to specify that a component is stateless. Instead, set pooling to TRUE and tx_vote to FALSE.	Instances
storage	Specifies the name of a component that reads and writes component state information from a remote database server. Required when using automatic persistence, or when using component-managed persistence with an implementation that delegates to EAServer's built-in storage component. The default is: `CtsComponents/JdbcStorage` Also specifies the connection cache and table used by the storage component.	Persistence (Storage Component, Connection Cache, Table)
thread.safe	Indicates whether multiple invocations can be processed simultaneously. For more information see "Concurrency property" on page 459.	Instances (Concurrency)
timeout	Specifies how long an active component instance can remain idle between method calls before being automatically deactivated.	Resources (Instance Timeout)
tx_outcome	Determines whether a CORBA::TRANSACTION_ROLLEDBACK exception is thrown to the client when a transaction is rolled back. The allowable settings are: • **always** The default. The server sends the exception to the client when a transaction is rolled back. • **failed** EAServer does not send the exception to the client when a transaction is rolled back. If you use this setting, you can code your components to raise a different exception with a descriptive message after calling the RollbackWork transaction primitive. With the failed setting in effect, EAServer may still throw a CORBA system exception if unable to commit a transaction at your component's request. You can set this property to failed if you require that no exceptions be sent to the client for rolled-back transactions or that an alternative exception be sent. This setting is useful if you require that the client be able to retrieve output parameters after a transaction is rolled back: if an exception is thrown, the output parameters are not available.	All Properties

Property	Description	Where displayed
tx_timeout	Specifies the maximum duration of an EAServer transaction. EAServer checks for timeouts after each method returns.	Resources (Transaction Timeout)
tx_type	Indicates whether the component participates in EAServer transactions and in what way. Valid values are: • `not_supported` • `supports` • `requires` • `requires_new` • `mandatory` • `user-managed` • `never`	Transactions
tx_vote	Indicates whether the component supports automatic demarcation and deactivation. When tx_vote is set to TRUE, the component must control transaction state and component deactivation by explicitly calling methods of the TransactionServer service object. When tx_vote is set to FALSE, EAServer automatically deactivates the component after each method invocation. You do not need to make explicit calls to SetComplete to cause deactivation because SetComplete is assumed by default. To override the default state, you can call SetAbort.	Transactions (when the Automatic Demarcation/ Deactivation check box is checked, the value of this property is FALSE)
type	Specifies the type of component. For PowerBuilder objects, EAServer sets this property to pb.	General

Exposing an NVO as a Web service

The EAServer Component wizard includes a page that allows you to expose the component that the wizard generates as a Web service.

The Expose Component as Web Service page of the wizard has options for exposing a component as an EJB 2.1 Web service (for EAServer 6.x) or as an EAServer 5.x Web service. If you expose the components as an EJB 2.1 Web service, you must specify a Java package name in the wizard or on the General tab page in the Project painter.

For an EAServer 5.x Web service, you need to set the properties described in Table 24-8.

Table 24-8: Properties of a component exposed as a Web service

Property	Description
Web Application	The name of the Web application to which you deploy the component. If the Web application you enter does not exist in EAServer, the application is created before you deploy the component. If no application is specified, the component is deployed to the “ws” default Web application in EAServer.
Service	The service name you want to use for the component when it is exposed as a Web service. If you do not specify a service name, the service name defaults to *packageName_componentName*.
HTTP Port	The port you use for Web services in EAServer. If you do not specify a port, the default value 8080 is used.

The Expose This Component As Web Service check box and the three text box fields are also listed on the Advanced tab of the Properties dialog box for an EAServer component. This allows you to set these properties without using the wizard or to modify these properties after completing the wizard.

If you reference a structure object in an EAServer component that you deploy as a Web service, the structure object is automatically rendered as a custom datatype.

The following limitation currently applies to components exposed as Web services in EAServer: If the PowerBuilder component has a function that passes a character datatype by reference, the component cannot be exposed as a Web service. An error message such as: “Can't find prefix for 'http://DefaultNamespace'.” displays.

Testing and debugging the component

This section describes three techniques you can use to test your component:

- Live editing
- Remote debugging
- Putting messages into the EAServer log

Troubleshooting EAServer components
For more information about troubleshooting components, see the EAServer documentation.

Live editing

To test or debug a component, you can use a feature of PowerBuilder called **live editing** that allows you to build the project automatically from the User Object painter. When live editing is enabled, PowerBuilder builds the project for an EAServer component each time you save the corresponding user object. The generator does not deploy PBDs to EAServer, but instead tells EAServer how to access the PBLs that contain the required object definition(s).

Service components
You cannot use live editing to test a component that you have set up as a service component. Service components are always in use when the server is running, so the changes that you make in the User Object painter cannot be saved.

How to enable live editing

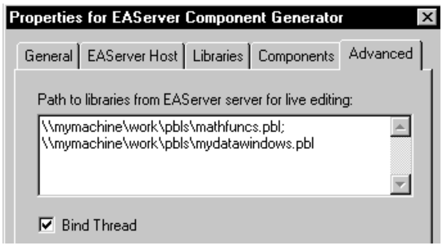
To enable live editing for a user object, you need to:

1 Create a project that includes the user object for which you want to generate an EAServer component.

You can use an existing PBL that allows for deployment to EAServer, or alternatively, you create a new project and use this just for testing purposes.

2 Optionally modify the live editing library list for the project.

When you are testing a component with a server that resides on a remote machine, you need to tell EAServer where to find the PBLs. To do this, you modify the library list on the Advanced page of the component's property sheet in the Project painter, shown below:



Path to libraries from EAServer server for live editing:

```
\\mymachine\work\pbls\mathfuncs.pbl;
\\mymachine\work\pbls\mydatawindows.pbl
```

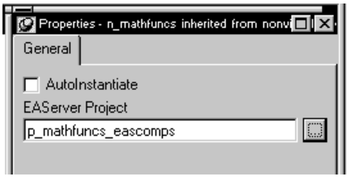
☑ Bind Thread

The library list you specify must contain fully qualified paths that use Universal Naming Convention (UNC) names. UNC names take the form: *\\servername\sharename\path\file*

By default, the live editing library list is based on the application library list. You do not need to modify the live editing library list if your server is local.

3 Specify the project that will be used to generate the component in the User object painter.

Enter the project name in the EAServer Project field, which is located on the General property page of the user object property sheet shown below.



Properties - n_mathfuncs inherited from nonv

General

□ AutoInstantiate

EAServer Project

p_mathfuncs_eascomps

The project name you specify must meet these requirements:

- It must be an EAServer component project.
- It must include the user object that you currently have open in the User Object painter.
- The library list for the project must match the current application library list.

How to generate the component

To generate an EAServer component from the User Object painter, select File>Save.

What happens when you generate the component

When you build a project from the User Object painter, PowerBuilder performs these operations:

- Generates CORBA IDL that describes the nonvisual object you saved

  The IDL is in turn used to build stubs and skeletons. The names of the IDL file, the stubs, and the skeletons are based on the name of the object.

  The component generator stores the new IDL in the Repository subdirectory of the EAServer installation directory.

- Generates a *PROPS* file that describes the properties of the EAServer component

  The *PROPS* file is stored in the following subdirectory of the EAServer installation directory: *Repository\Component\package-name*

PowerBuilder builds the component just as it would at deployment time, except that it does *not* generate *PBD*s for the component. In addition, it sets the pb.live_edit property to TRUE and assigns the library list you specified for live editing to the pb.librarylist property.

If the project build results in errors, PowerBuilder displays the error messages in the Output window.

If instance pooling is enabled for the user object, the generator disables pooling for the current build. Pooling is not supported with live editing because PowerBuilder cannot save the user object if the *PBL* that contains the user object is locked by EAServer.

Remote debugging

When you are building a PowerBuilder custom class user object as an EAServer component, you can use the PowerBuilder debugger to debug the EAServer component. You can debug the component whether you use the live editing feature in the User Object painter or deploy the component to EAServer from the Project painter.

For more information about live editing, see “Live editing” on page 497.

Getting ready to debug a component

Before you begin debugging a remote component, check that your configuration meets the following requirements:

- You are using the same version of the application and *PBL*s as were used to develop the deployed component. If you want to debug several deployed components in the same session, they must all have been built using the same versions of the *PBL*s, the same application name, and the same library list.
- The Supports Remote Debugging check box on the Components properties page in the Project painter is checked. You can also set the debugging option by checking the Supports Remote Debugging check box in the Project wizard.
- You have a client application that exercises the methods and properties in the deployed components. This can be a compiled executable built with any compatible development tool or a PowerBuilder application running in another PowerBuilder session.

Selecting EAServer components for debugging

When you debug an EAServer target, the set of components that can be debugged is determined from the project. The set includes all components selected on the Components page in the Project painter for which the Remote Debugging check box is selected. If you want to select a different set of components or debug components from more than one package, select Debug>Select Components from the menu bar in the debugger or click the Select Components button on the PainterBar.

Starting the debugger

To begin debugging, open the target that contains the deployed components. Click the Start Remote Debugging button in the PainterBar and complete the wizard. You can select only components that were generated in PowerBuilder with remote debugging support turned on. Remote debugging support is a security setting that does not add any debugging information to the component. You turn remote debugging support on when you are testing a component, then turn it off when you deploy the component to a user's site to prevent users from stepping into and examining your code.

Set breakpoints as you would when debugging a local application, then start the client application that invokes the remote components (if it is not already running).

Differences from local debugging

You will notice two major differences between debugging local and remote applications:

- When you start the debugger, it does not minimize.
- The new Instances view shows each instance of the components you are debugging. For each instance, it shows the component and package names, an instance number, and its current state: running, idle, or stopped. If there is more than one instance, a yellow arrow indicates which one is currently being debugged.

Unsupported features
The Objects In Memory view, expression evaluation, and changing variable values are not supported.

About states

The instances view shows the state of each instance of each component:

- **Idle** The component is idle or in the instance pool.
- **Running** The component is currently executing code.
- **Stopped** The component is stopped at a breakpoint waiting for a debugger action.

When an instance is destroyed, it is removed from the Instances view.

Multiple instances

Multiple component instances can be stopped at the same time, but actions you take in the debugger act only on the first instance that hits a breakpoint. This instance is indicated by a yellow arrow in the Instances view. The current instance changes to the next instance in the queue when the method completes or when you click Continue.

You can also change context from one instance to another by double-clicking the new instance in the Instances view. You might want to do this if you step over a call to another component instance and the Instances view shows that the called instance stopped.

Putting messages into the EAServer log

To record errors generated by PowerBuilder objects running in EAServer to the EAServer log, create an instance of the ErrorLogging service context object and invoke its log method. For example:

```
ErrorLogging inv_el
this.GetContextService("ErrorLogging", inv_el)
inv_el.log("Write this string to log")
```

You can use the ErrorLogging service to provide detailed information about the context of a system or runtime error on the server. This information is useful to system administrators and developers in resolving problems.

While you are developing components, you can use the ErrorLogging service to trace the execution of your component. For example, you can write a message to the log when you enter and exit functions. The message can identify the name of the component, whether it is entering or exiting a function, and which function it is in.

Automatic recording of exception information
Information about the exception type and location of an exception caused by a PowerBuilder component running on the server is recorded automatically in the server log. It is no longer necessary to invoke the error logging service to obtain minimal information about these exceptions.

When you use the XSL-FO technique to generate a PDF file, detailed informational and warning messages are sent to the log. You can suppress these messages by setting the PB_FOP_SUPPRESSLOG environment variable to 1.

Printing data

You can use a DataStore to print data on a remote server if the server is running on Windows or Solaris.

Platform note
The following example would not work on HP-UX or AIX. On these platforms, EAServer uses a windows-free version of the PowerBuilder runtime environment that does not provide support for graphical operations such as printing. Printing using DataStore print functions is currently supported only on Solaris. However, DataStore objects can be printed on all UNIX platforms using the SaveAs function with the PDF SaveAsType.

For more information, see “Printing to PDF” on page 506.

In this example, the server component uo_employees has a function called print_employees. print_employees generates an instance of the DataStore ds_datastore, and then prints the contents of this DataStore.

Here is the signature of the print_employees function:

print_employees( ) returns integer

Here is the script for the print_employees function:

```
datastore ds_datastore
int li_rc
ds_datastore = create datastore
ds_datastore.dataobject = "d_empdata"
ds_datastore.SetTransObject (SQLCA)
ds_datastore.Retrieve()
li_rc = ds_datastore.Print()
return li_rc
```

Printing on the Solaris operating system

On Solaris, you can print directly to a PostScript or PCL5 file. You do not need to make any code changes to print reports on Solaris instead of Windows—you can use the same properties, functions, and events as in Windows.

To print DataStores, use the DataStore Print method or PrintDataWindow(*PrintJobName*, *DataStoreName*). If you link a DataStore to a DataWindow and print the DataWindow object on Solaris, the printed output has the fonts and layouts defined in the DataWindow object.

Blank spaces
A blank space is not supported in a print job on Solaris, so the PBVM substitutes a hyphen for each blank space in a print job name before sending it to a printer.

Using fonts

The fonts used in printing are those provided in the dwprinter/fontmetrics directory. AFM and TFM files are ASCII formatted files that contain information about specific PostScript (AFM) and PCL (TFM) fonts. Each PostScript and PCL font has a corresponding font metric file.

The print mechanism reads AFM and TFM files to acquire font metric information. This includes such things as character widths, baseline location, ascender size, descender size, underline stroke width, and underline position. It then translates this metric information into the formats expected by Xlib APIs such as XTextWidth.

The best approach is to select fonts that are available on both Windows and Solaris when you design the DataWindow. However, each platform has its own font-rendering engine, so you might notice differences in the sizes of fonts between Windows to Solaris. You should test the printed output on Solaris early in the development process.

Limitations

Support for printing DataWindow objects is based on the Wind/U product from Bristol Technology. The Wind/U GDI library and Xprinter library have the following limitations:

- No support for multibyte character sets (MBCS) or Unicode
- Xprinter is not thread safe, therefore print jobs are serialized

Setting up a printer

To set up a printer to print DataWindow objects, you must add access to the printer, set up the *dwprint.ini* configuration file, and create an XPPATH environment variable.

Adding access to the printer

As the root user, add access to the printer on Solaris using the Solaris admintool utility. For more information, see the Solaris documentation.

Setting up dwprint.ini

The *dwprint.ini* file in the *$EAServer/bin* directory is the configuration file for DataWindow printing. It closely follows the Microsoft Windows approach to printer configuration. As a result, it includes [windows], [devices], and [ports] sections where you must provide appropriate entries for your printers.

You usually do not need to modify other sections in this file. However, some problems can be resolved by adding or changing other sections. For example, you can try adding an entry like the following to the [intl] section to change a date format:

```
[intl]
sShortDate=m/d/yyyy //Set the year to 4 digit.
```

The entries in *dwprint.ini* are based on entries in the *.WindU* file. For more information about setting up this file, see the Wind/U User's Guide at http://www.bristol.com/support/windu/wu_ug/ch13.htm

Specifying ports

Each line in the [ports] section of *dwprint.ini* contains a user-defined port name and an associated command that is used to spool the output file. For example, the command to send a print job to a printer called myprinter connected directly to your system is:

```
lp -s -d myprinter -t$XPDOCNAME
```

$XPDOCNAME represents the name of the output file sent to the printer. The -s option suppresses the display of messages sent from lp in the EAServer Server console.

The following is an example of the [ports] section of the *dwprint.ini* file with two ports defined for remote printers called prnt1 and prnt2, one for a local printer, and an entry for printing to a file. The name of the output file is enclosed in quotes. This enables file names with multiple words to be used. The quotes must be escaped for remote servers because rsh strips them out:

```
[ports]
colorpr1=rsh prntsvr lp -s -d prnt1 -t\"$XPDOCNAME\"
colorpr2=rsh prntsvr lp -s -d prnt2 -t\"$XPDOCNAME\"
LOCAL=lp -d myprinter -t"$XPDOCNAME"
FILE: =
```

Matching a printer type to a defined port

The [devices] section contains a list of all currently configured printers. Each line contains a user-defined alias for the printer and three arguments: the printer model, the printer mode (PCL4, PCL5, or PostScript), and one or more ports to which the printer is connected.

The printer model is the name of the printer description file (PPD) used by the printer. PPD files are installed in the *dwprinter/ppds* directory in your PBVM installation. The text file *filename_map.txt* in that directory maps the name of the file that contains the printer description to the type of printer. For example, these are the mappings for the color_lj model used in the rest of the examples:

```
color_lj.pcl:"HP Color LaserJet PCL Cartridge"
color_lj.ps:“HP Color LaserJet PS"
```

The printer model and mode are separated by a space. The mode and port are separated by a comma. For example, for the first device specified in the following [devices] section, the alias is HP Color LaserJet PS, the model is color_lj, the mode is PostScript, and two ports are specified: FILE: and colorpr1.

```
[devices]
HP Color LaserJet PS=color_lj PostScript,FILE:,colorpr1
HP Color LaserJet PS=color_lj PCL5,colorpr2
HP Color LaserJet PS=color_lj PostScript,LOCAL
HP LaserJet PS=NULL PostScript,FILE:
HP LaserJet PCL=NULL PCL,FILE:
```

Specifying a default printer

The [windows] section contains default printer information. Like the ports specification, each device line has three arguments: the name of the PPD file, the driver, and the port, but in the [windows] section they are all separated by commas.

The following example shows a default entry for printing to a file (when the printer file description is set to NULL) as well as two other entries. The semicolon at the beginning of two of the lines is a comment character, so the current default printer is the HP Color LaserJet printer on the port colorpr1.

```
[windows]
device=color_lj,PostScript,colorpr1
;device=color_lj,PostScript,colorpr2
;device=NULL,PostScript,FILE:
```

Setting printer options

The *dwprint.ini* file must contain a configuration section for each model you have defined in the [windows], [devices], and [ports] sections. The configuration section provides default setup information for the printer, including the number of copies, orientation, page size, and DPI.

For example, for the color_lj printer used in the preceding examples, add configuration sections like this:

```
[color_lj,PostScript]
Filename=jaguar.ps
Scale=1.00
Copies=1
Orientation=Portrait
PageSize=Letter
DPI=300

[color_lj,PCL5]
Filename=jaguar.pcl
Scale=1.00
Copies=1
Orientation=Portrait
```

```
PageSize=Letter
DPI=300
```

Setting the XPPATH environment variable

Before you start a print job, set the XPPATH environment variable. The XPPATH variable must contain the path to a directory that includes printer description files and printer-specific font mapping files. This information is installed in the dwprinter directory in your PBVM installation.

For a C shell, set the path as follows:

```
setenv XPPATH $EAServer/dwprinter
```

For a Korn shell or a Bourne shell, set the path as follows:

```
XPPATH = $EAServer/dwprinter;export XPPATH
```

Printing to PDF

You can use two techniques to save the data in a DataStore to PDF: Using the GNU Ghostscript distiller, and using the ability to process data using XSL Formatting Objects (XSL-FO). You can save the DataWindow object's data to XSL-FO or PDF, and print using Java printing.

Using the GNU Ghostscript distiller

To use the GNU Ghostscript distiller, you must make sure that Ghostscript files and the default PostScript printer driver and related files are installed on the server in the same directory as the PowerBuilder runtime files. The Ghostscript method is not supported on UNIX.

Using XSL-FO

To use XSL-FO, the Apache XSL Formatting Objects processor (FOP) must be installed on the server in the same directory as the PowerBuilder runtime files, and the following JAR files must be in the classpath:

fop-0.20.4\build\fop.jar
fop-0.20.4\lib\batik.jar
fop-0.20.4\lib\xalan-2.3.1.jar
fop-0.20.4\lib\xercesImpl-2.1.0.jar
fop-0.20.4\lib\xml-apis.jar
fop-0.20.4\lib\avalon-framework-cvs-20020315.jar

You can add these files to the CLASSPATH environment variable or to *User_setenv.bat* or *Serverstart.bat.*

When you use XSL-FO with EAServer to produce a PDF file, detailed informational and warning messages are written to the Jaguar log. To suppress all these messages, set the PB_FOP_SUPPRESSLOG environment variable to 1.

For more information, see the chapter on enhancing DataWindow objects in the PowerBuilder *User's Guide*.

Deploying a component to EAServer

PowerBuilder VM must be available on the server

You can deploy components to EAServer hosts running Windows, UNIX, and Linux. The version of the PowerBuilder VM on the development computer must also be available on the server. To find out which versions of EAServer and the PowerBuilder VM are available on each platform, contact your sales representative or check the Sybase Web site at http://www.sybase.com.

The PowerBuilder VM includes *PBVM110.DLL*, *PBJAG110.DLL*, *PBDWM110.DLL*, and other PowerBuilder files required at runtime. On UNIX and Linux, the shared libraries are called *libpbvm110x.ext*, *libdwm110x.ext*, and so forth, where *ext* is the shared library extension for each platform. EAServer uses a version of the PowerBuilder runtime files, identified by the *x* at the end of the file name, that does not provide support for Windows API calls or graphical operations, including printing.

Consuming a .NET Web service from a PowerBuilder NVO
If you call a .NET Web service from a PowerBuilder component running in EAServer, you must deploy the *Sybase.PowerBuilder.WebService.Runtime.dll*, *Sybase.PowerBuilder.WebService.RuntimeRemoteLoader.dll* and the dynamically generated .NET assembly to the EAServer *bin* directory.

EAServer supports multiple versions of the PowerBuilder VM on the same server. Components built with different versions of PowerBuilder can coexist on the same server as long as the required version of the PowerBuilder VM is available on the server.

When you deploy a component from PowerBuilder 11 to EAServer, the component is associated with the version of the PowerBuilder VM that you are using. The com.sybase.jaguar.component.pb.version property is set to 11.0 on the All Properties tab page of the component's property sheet in EAServer Manager for EAServer 5.x. In EAServer 6.x, it displays on the General property page for the component under the CORBA Packages node.

If you do not use the PowerBuilder development environment to deploy a PowerBuilder component to EAServer, you can specify the correct version of the VM for the component on its property sheet in EAServer Manager.

If you deploy a PowerBuilder component to a server that does not have the version of the PowerBuilder VM used in the development environment, the deployed component cannot be instantiated.

About deploying EAServer components

To deploy a component to EAServer, create a new project and build the project. The new project lists the objects that will be included and specifies the name of the output library that will contain the generated components.

Making DataWindow definitions available
If your scripts reference DataWindow objects dynamically, you must check the Include Unreferenced Objects in Consolidated PBD box in the wizard or painter to make the DataWindow definitions available to the component.

How to deploy the component

To deploy a component to EAServer, open the project created by the wizard and select Design>Build Project.

What happens when you deploy to EAServer

When you deploy components to EAServer, the component generator performs these operations:

- Generates CORBA IDL that describes the nonvisual objects you selected for deployment

  The IDL is in turn used to build stubs and skeletons. The names of the IDL file, the stubs, and the skeletons are based upon the name of the object.

  The component generator stores the new IDL in the *Repository* subdirectory of the EAServer installation directory.

- Generates a *PROPS* file that describes the properties of the EAServer component

  The *PROPS* file is stored in the following subdirectory of the EAServer installation directory: *Repository\Component\package-name*.

- Generates one or more *PBD* files for the deployed component

  The *PBD* files are stored in the following subdirectory of the EAServer installation directory:
  Repository\Component\package\component\Ccookie

  where *cookie* represents the number for this generation of the build. If the library name is not qualified (no path is specified), the component generator prepends a dollar sign ($) to the name. By default, EAServer uses the most recent version of the component.

Cleaning up the EAServer repository

You can reclaim disk space by removing directories you no longer need, using the following procedure.

❖ **To delete unwanted directories and PBD files:**

1. Delete all directories except the most recent.
2. Rename the remaining directory to *C1*.
3. Set the value of the pb.cookie property to 1 on the All Properties tab page of the property sheet for the component in EAServer Manager or the Advanced tab page in the Management Console.
4. Restart EAServer.

Changing the component's codeset

Any EAServer component deployed by PowerBuilder automatically uses the code set of the server. If you want your component to use a different code set, set the component's com.sybase.jaguar.component.code.set property to an appropriate value.

In EAServer Manager, you can do this in the Component Properties dialog box on the All Properties tab page. Add the com.sybase.jaguar.component.code.set property and specify an appropriate value such as big5 or iso_1. In the Management Console, select a value from the Code Set drop-down list on the General page.

If EAServer was started using the utf-8 codeset and your component returns a string containing the Euro and/or British pound symbol, set the code.set property to cp1252.

CHAPTER 25 Building an EAServer Client

About this chapter

This chapter explains how to build a PowerBuilder client that accesses an EAServer component. For information about secure connections, see Chapter 26, “Using SSL in PowerBuilder clients.”

Contents

About building an EAServer client

A PowerBuilder application can act as a client to an EAServer component. To access a method associated with a component on the server, the PowerBuilder client needs to connect to the server, instantiate the component, and invoke the component method.

In general, you use an instance of a Connection object to connect to EAServer. If you want to create a CORBA-compatible client, you can use the JaguarORB object to establish the connection to the server. By using the JaguarORB object, a PowerBuilder client can access EAServer in the same way that a C++ client would.

You can use the techniques described in this chapter to build clients for EJB components running in EAServer. For information about building a client for EJB components on EAServer and other J2EE-compliant servers, see Chapter 29, “Building an EJB client.”

About using the wizards

PowerBuilder provides two wizards to facilitate the development of EAServer clients:

- **Connection Object wizard** Adds code required to connect to the server
- **EAServer Proxy wizard** Helps you create projects for building proxy objects for the EAServer components you want to access from the client

About the development process

Steps for building an EAServer client

To build and deploy an EAServer client, you need to complete the following steps:

1. Use the EAServer Connection Object Wizard to create a standard class user object inherited from the Connection object. You can then use this object in a script to establish a connection.

   If you use the Template Application Start Wizard to create the client application, you can create the Connection object in that wizard.
2. Use the EAServer Proxy Wizard to create a project for building proxy objects. Then generate the proxy objects.
3. Create the windows, menus, and scripts required to implement the user interface.
4. Write the code required to create the EAServer component instance and call one or more component methods from the client.
5. Test and debug the client.
6. Deploy the application.

Connecting to EAServer

Using the Connection object

The simplest way to connect to EAServer is to use the capabilities of the **Connection object**, a nonvisual object that handles communications with the server. You can write all the code to connect to the server by hand, or you can use the Connection Object wizard to get started.

Writing the code by hand

Declaring the connection variable

The Connection object is not a built-in global object. You need to declare a global or instance variable of type connection.

Establishing a connection

To establish a connection to the server, you need to execute the PowerScript statements required to perform these operations:

1. Use the Create statement to instantiate the Connection object.
2. Set properties for the Connection object.
3. Invoke the ConnectToServer function to establish a connection to the server.
4. Check for errors.

You can perform these operations in a single script or in several scripts, but they must be performed in the order shown.

Example The following script instantiates the myconnect Connection object and sets the connection properties to identify the communications driver for EAServer, the host name and port number of the server, and the default package. Then the script invokes the ConnectToServer function to establish a connection to the server and checks for errors:

```
// Global variable:
// connection myconnect
long ll_rc
myconnect = create connection
myconnect.driver = "jaguar"
myconnect.location = "Jagserver1:2000"
myconnect.application = "PB_pkg_1"
myconnect.userID = "bjones"
myconnect.password = "mypass"
ll_rc = myconnect.ConnectToServer()
IF ll_rc <> 0 THEN
      MessageBox("Connection failed", ll_rc)
END IF
```

Setting the Connection object properties

Table 25-1 provides some guidelines for setting Connection object properties when you are communicating with EAServer.

Table 25-1: Connection object properties for EAServer

Property name	Description	Examples
Application	The default package to be used for EAServer components	"PB_pkg_1"
Driver	The name of the EAServer driver.	"jaguar"
Location	The host name and port number for the server, separated by a colon. The Location property can also specify a fully qualified URL that uses one of the following formats: iiop://*host*:*port* iiops://*host*:*port* http://*host*:*port* https://*host*:*port* To take advantage of EAServer's load balancing and failover support, you can also specify a semicolon-separated list of server locations.	"localserver:2000" "iiop://srv1:2000" "iiops://srv3:2001" "http://srv5:8000" "iiop://s1:2000;iiop://s2:2000"
Password	The EAServer password.	"mypass"
UserID	The EAServer user ID.	"bjones"
Options	One or more EAServer ORB property settings.	"ORBLogFile='jaglog.log'"

Establishing multiple connections

PowerBuilder allows you to instantiate multiple Connection objects. This makes it possible for you to establish multiple connections in a single client application. For example, you could instantiate two separate Connection objects to connect a client to two different servers.

Setting options

When you connect to EAServer using either the Connection object or the JaguarORB object, you are using the EAServer client ORB. You can set its properties in the Options string of the Connection object or using JaguarORB's Init function.

Using a different code set

To connect to an EAServer component that handles double-byte characters, make sure the component is using the correct code set. The code set can be changed in the Component Properties dialog box in the Management Console. You must also set your PowerBuilder client's code set to use the component. To do so, set the Options property of the Connection object. For example, if you want to handle Korean characters in the eucksc code set, use the following script for a Connection object called myConnection:

```
myConnection.Options = "ORBCodeSet='eucksc'"
```

If EAServer was started using the utf-8 codeset and you are calling a Java or PowerBuilder component that returns a string containing the Euro and/or British pound symbol, set the ORBCodeSet property to cp1252. For example:

```
myConnection.Options = "ORBCodeSet='cp1252'"
```

Troubleshooting connections

When a connection fails, you can obtain more information about the failure in a log file by enabling the ORBLogIIOP option and specifying a value for the ORBLogFile option. If you want to set multiple options, they must be set in the same statement. Separate the options with commas:

```
myConnection.Options = &
  "ORBLogIIOP='TRUE', ORBLogFile='d:\temp\ORBLog.txt'"
```

For a complete list of options, see the online Help for the Connection object or the EAServer documentation.

Using the wizard to create a Connection object

When you select EAServer as the connection type in the Connection Object wizard, PowerBuilder creates a standard class user object inherited from the Connection object. You supply the Connection object properties in the wizard and specify whether connection information will be provided in the registry, an *INI* file, or a script. The Connection Object wizard gets information about the server you want to connect to from the EAServer profiles you have set up. For how to create an EAServer profile, see "Creating an application server profile" on page 455.

The Constructor event of the Connection object calls a function, of_getconnectioninfo, that gets the stored connection information from the source you specified.

Once you have used the Connection Object wizard to create a Connection object, you need to execute the PowerScript statements required to perform these operations:

1. Use the Create statement to instantiate the Connection object.
2. Invoke the ConnectToServer function to establish a connection to the server.
3. (Optional) Check for errors.

You do not need to set properties for the Connection object, but you can modify them in the of_getconnectioninfo function. You can also set options for the Connection object in its constructor event, for example:

```
this.options = "ORBHttp='TRUE'"
```

Example The following script instantiates the myconnect instance of the n_myclient_connect object created by the wizard, invokes the ConnectToServer function to establish a connection to the server, and checks for errors:

```
long ll_rc
myconnect = create n_myclient_connect
ll_rc = myconnect.ConnectToServer()
IF ll_rc <> 0 THEN
   MessageBox("Connection failed", ll_rc)
END IF
```

Establishing multiple connections

You can establish multiple connections in a single client application. If you want to connect a client to two different servers, run the Connection Object wizard again to create a new user object with different connection properties.

Generating EAServer proxy objects

About EAServer proxy objects

Each EAServer component has a corresponding proxy object in the client application. To access an EAServer component, you need to communicate through the EAServer proxy.

An EJB component has two corresponding proxy objects in the client application—one for the home interface and one for the remote interface. For example, an EJB component named Cart produces two proxies, CartHome and Cart. To access an EJB component, you need to communicate through these two proxies.

Before you can generate proxy objects for an EAServer client, you need to create a new project. The new project lists the objects that will be included and specifies the name of the output library that will contain the generated proxy objects.

Using the EAServer Proxy wizard

The EAServer Proxy wizard helps create projects for building EAServer proxy objects. It allows you to connect to EAServer and select the components you want to be able to access from the client. Once you have created the project, you can then use the Project painter to modify your project settings and build the proxy library.

Building EJB clients
To build an EJB client that can use the services of an EJB component on EAServer and other J2EE-compliant servers, you can use the EJB Client Proxy wizard. For more information, see Chapter 29, “Building an EJB client.”

Reserved words in method names

When you generate a proxy for an EAServer component that was not created in PowerBuilder, the names of any methods that use a PowerBuilder reserved word are changed. The proxy generator automatically adds an underscore (_) prefix to these methods. For example, if the component has a method with the name destroy, the method in the proxy will be _destroy.

Using arrays with a TO clause

When you generate a proxy for a PowerBuilder component containing an array that uses a TO clause, the proxy object represents the range as a single value because CORBA IDL does not support the TO clause. For example, `Int ar1[5 TO 10]` is represented as `Int ar1[6]`, with `[6]` representing the number of array elements. Client applications must declare the array using a single value instead of a range.

Prepending module names

You can choose to prepend the name of the IDL module that defines the component to the name of the proxy object created, which makes it easier to identify proxy objects with similar names. For example, if you select the SessionInfo component in the CTSSecurity module and check the Prepend EAServer Package Name to Object Name option in the wizard or Project painter, the proxy object will be named `ctssecurity_sessioninfo`. For some EAServer system modules, currently CtsComponents and XDT, the module name is always prepended to objects to avoid naming conflicts.

The package name and the name of the IDL module are often the same, but they can be different, and it is always the name of the IDL module that is prepended.

Excluding exceptions

Many EAServer components throw exceptions that you can handle in your client application. If you want to use the proxy you are generating with an existing client application that does not handle exceptions, or if you do not want to declare the exceptions in the client you are building, you can choose to exclude exceptions from the generated proxy, either in the wizard or in the Project painter. See "Handling errors" on page 535 for more information about handling errors in clients.

Datatype mappings

All EAServer component interfaces are defined in standard CORBA IDL. For a list of datatypes used in EAServer, their CORBA IDL equivalents, and the PowerBuilder datatypes to which they map, see the *PowerScript Reference* or the online Help.

Invoking component methods

Once a connection to EAServer has been established and a proxy object or objects created, the client application can begin using the server components.

Invoking a component method

To invoke a method on most component types, you need to execute the PowerScript statements required to perform these operations:

1. Use the CreateInstance method to create an instance of the component.
2. Invoke the method.

You use a different technique to invoke EJB component methods. See "Invoking an EJB component method" on page 520.

Example 1 This script instantiates a component on the server and invokes a component method. In this example, the CreateInstance method does not specify a package; therefore, EAServer uses the default package specified in the Application property of the Connection object:

```
// Global variable:
// connection myconnect

uo_customer iuo_customer
string ls_custid
long ll_rc
```

```
ls_custid = Trim(sle_custid.text)
ll_rc = myconnect.CreateInstance(iuo_customer)
if ll_rc <> 0 then
   MessageBox("CreateInstance failed", ll_rc)
   return 999
end if
if iuo_customer.retrieve_balance(ls_custid) != 1 then
    MessageBox("Error", "Retrieve failed!")
end if
```

Example 2 This script instantiates a component on the server and assigns the object reference to a variable whose datatype is an ancestor of the class for the component. The second argument to the CreateInstance function specifies the class name for the component as well as the EAServer package name:

```
// Global variable:
// connection myconnect

uo_person lnv_object
string ls_custid
long ll_rc

ls_custid = Trim(sle_custid.text)
ll_rc = myconnect.CreateInstance(lnv_object, &
   "PB_pkg_1/uo_customer")
if ll_rc <> 0 then
   MessageBox("CreateInstance failed", ll_rc)
return 999
end if
if iuo_customer.retrieve_balance(ls_custid) != 1 then
   MessageBox("Error", "Retrieve failed!")
end if
```

Invoking a local instance

By default, the TransactionServer CreateInstance method invokes the EAServer name service to create proxies. Proxies for remote components might be returned by the name service rather than an instance that is running locally. To guarantee that a locally installed instance is used, specify the component name as "`local:`***`package`***`/`***`component`***", where *package* is the package name and *component* is the component name. The call fails if the component is not installed in the same server.

Invoking an EJB component method

To invoke an EJB component method, you need to execute the PowerScript statements required to perform these operations:

1. Use the Lookup function to access the component's home interface.
2. Invoke the method on the interface to create or find an instance of the component and get a reference to the component's remote interface.
3. Invoke the business methods on the remote interface.

Does not apply to EJBConnection methods
This section applies to client applications that use an EAServer proxy object and PowerScript functions. For information about invoking EJB methods in client applications that use an EJB client proxy and EJBConnection methods, see Chapter 29, "Building an EJB client."

Specifying the home interface name

PowerBuilder provides an optional third argument to the Lookup function to specify the name of the home interface. EJB components have a property in EAServer called com.sybase.jaguar.component.home.ids. You do not need to specify the third argument to the Lookup function if the home.ids property looks like this:

IDL:*PackageName*/*ComponentName*Home:1.0

For example:

```
IDL:vacation/TripFinderHome:1.0
```

In most cases, however, the home.ids property uses the java package naming scheme and you should use the third argument to make sure that the EJB home interface can be located. The string that you specify for this argument must match the component's com.sybase.jaguar.component.home.ids property without the leading *IDL:* and trailing *:1.0*.

For example, suppose the home.ids property is this:

```
IDL:com/myproj/myejbs/TripFindHome:1.0
```

Your Lookup function call should look like this::

```
myconn.lookup(myTripFindHome,"MyEJBs/TripFindHome", &
   "com/myproj/myejbs/TripFinderHome")
```

Alternatively, you can use the fully-qualified Java class name of the home interface specified in dot notation. For example:

```
ts.lookup(MyCartHome, "MyEJBs/TripFindHome", &
   "com.myproj.myejbs.TripFinderHome")
```

Lookup is case sensitive
Lookup in EAServer is case sensitive. Make sure that the case in the string you specify for the arguments to the Lookup function matches the case in the home.ids property.

Creating or finding an instance of an EJB

EAServer supports three types of EJBs—session beans, entity beans, and message-driven beans.

A session bean is *created* in response to a client request. A client usually has exclusive use of the session bean for the duration of that client session.

An entity bean represents persistent information stored in a database. A client uses an entity bean concurrently with other clients. Since an entity bean persists beyond the lifetime of the client, you must use a primary key class name to identify or *find* a preexisting component, if the bean has already been created.

A message-driven bean is similar to a stateless session bean, but it responds only to JMS messages and has no direct client interface.

The following examples assume that an EJB component that provides e-commerce shopping cart functionality is running on EAServer. This component is called Cart and is included in a package called Shopping.

Example 1 This script instantiates the Cart component and invokes several component methods. In this example, the second argument to the Lookup method specifies the component name as well as the EAServer package name:

```
//Instance variable:
//Connection myconnect

CartHome MyCartHome // EJB's home interface
Cart MyShoppingCart // EJB's remote interface
long ll_result

//Get the home interface
ll_result = &
myconnect.Lookup(MyCartHome, "Shopping/Cart", &
   "com.sybase.shopping.Cart")

//Get a reference to Cart component's business logic
TRY
   MyShoppingCart = MyCartHome.Create()
CATCH (ctscomponents_createexception ce)
   MessageBox("Create exception", ce.getmessage())
   // handle exception
END TRY
```

```
//Use the shopping cart
MyShoppingCart.AddItem(66)
MyShoppingCart.Purchase()
```

Example 2 If the Cart EJB component is defined as an entity bean, then the script must use the findByPrimaryKey method to find and reference a preexisting or persistent component if one exists:

```
//Instance variable:
//Connection myconnect

CartHome MyCartHome // EJB's home interface
Cart MyCart // EJB's remote interface
long ll_result

//Get the home interface
ll_result = &
myconnect.Lookup(MyCartHome, "Shopping/Cart", &
   "com.sybase.shopping.Cart")

//Get a reference to Cart from a previous session
TRY
   MyCart = MyCartHome.findByPrimaryKey("MYkey")
CATCH ( ctscomponents_finderexception fe )
   MessageBox("Finder exception", &
      fe.getmessage())
   // handle exception
END TRY
//Use the shopping cart
MyCart.AddItem(66)
MyCart.Purchase()
```

Restrictions

PowerBuilder clients to EJB objects act as CORBA clients, which means that they do not have the full capabilities of Java clients. Java clients can use methods inherited from the javax.ejb.EJBObject interface.

For example, a Java client can obtain a handle for a remote interface instance. The handle is a binary encoding of the session state between the client and the bean. The client can obtain a handle, save it to disk or mail it to another location, then reestablish the session at a later time. PowerBuilder clients can obtain similar functionality using the Object_To_String and String_To_Object functions of the JaguarORB object.

Handling exceptions

The remote interface of an EJB component can indicate errors or warnings. Standard exceptions thrown by the EJB component are mapped to CORBA system exceptions. The EJB component can also throw user exceptions. For information about handling exceptions thrown by EAServer components, see “Handling errors” on page 535.

For information about calling an EJB component from a PowerBuilder component in EAServer, see “Accessing an EJB component” on page 490.

Destroying instances

Destroying the proxy object instance

After you have finished using an EAServer component, you can explicitly destroy the EAServer proxy object by using the DESTROY statement, or you can let PowerBuilder’s garbage collection facility clear the object out of memory for you automatically. In either case, the destruction of the client-side proxy object has no effect on the lifecycle of the server component. Destruction of the server component is handled by EAServer.

Deactivating the component instance

If the Automatic Demarcation/Deactivation setting is disabled for a component, and you close the client application while the component is still bound to the client (the component did not call SetComplete or SetAbort), the component is *not* deactivated. To ensure that the component instance is deactivated, you can do one of the following things:

- In the Close event of the client application, invoke a method of the server component that deactivates the component (by calling SetComplete or SetAbort).
- Set the Timeout property for the component to a value other than 0. If the component’s Timeout property is set to 0, the component will never time out.

Using the JaguarORB object

To create a CORBA-compatible client, you can use the JaguarORB object instead of the Connection object to establish the connection to the server. The JaguarORB object allows you to access EAServer from PowerBuilder clients in the same way as C++ clients.

Two techniques

The JaguarORB object supports two techniques for accessing component interfaces, using its String_To_Object and Resolve_Initial_References functions.

Using the String_To_Object function works in the same way that the ConnectToServer and CreateInstance functions on the Connection object do internally. The String_To_Object function allows you to instantiate a proxy instance by passing a string argument that describes how to connect to the server that hosts the component. The disadvantage of this approach is that you lose the benefits of server address abstraction that are provided by using the naming service API explicitly.

If you want to use the EAServer naming service API, you can use the Resolve_Initial_References function to obtain the initial naming context. However, this technique is not recommended because it requires use of a deprecated SessionManager::Factory create method. Most PowerBuilder clients do not need to use the CORBA naming service explicitly. Instead, they can rely on the name resolution that is performed automatically when they create EAServer component instances using the CreateInstance and Lookup functions of the Connection object.

About the naming service

The EAServer naming service is an implementation of the CORBA CosNaming component, which is a collection of interfaces that provide support for object binding and lookup. For more information about the CosNaming module, see the EAServer interface repository documentation. The interface repository documentation can be viewed in a Web browser by connecting to your server with the URL http://*yourhost*:*yourport*/ir/ where *yourhost* is the server's host name and *yourport* is the HTTP port number.

Instantiation using String_To_Object

Obtaining proxies for SessionManager interfaces

To instantiate a proxy without explicitly using the CORBA Naming Service, you use the String_To_Object function of the JaguarORB object in conjunction with interfaces defined in the SessionManager module. Before using the Manager, Session, and Factory interfaces, you need to use the EAServer Proxy wizard to create a proxy library project for the SessionManager module, build the project, and include the generated proxy library in the library list for the client target.

Identifying the server

You use the SessionManager::Manager interface to interact with the server. You can identify the server using its Interoperable Object Reference (IOR) or its URL. The IOR string encodes the server's host address and the port at which the server accepts IIOP requests. Each time a server is started, it writes a hex-encoded IOR string with standard encoding to two files for each listener, one containing the IOR string by itself, and the other containing the IOR as part of an HTML PARAM definition that can be inserted into an APPLET tag. The files reside in the *HTML* subdirectory of the EAServer directory. You can code the client to obtain the IOR string from one of these files.

Creating an authenticated session

After initializing the ORB and obtaining the IOR or URL of the server, use the String_To_Object function to convert the string to a CORBA object reference that you can convert to a reference to the Manager interface using the _Narrow function. Then use the createSession method of the Manager interface to create an authenticated session between the client application and the server.

Creating a reference to the component's interface

Use the session's lookup method to return a factory for proxy object references to the component you want to call. Then call the create method of the Factory object to obtain proxies for the component. The create method returns a CORBA object reference that you can convert into a reference to the component's interface using the _Narrow function.

A component's default name is the package name and the component name, separated by a slash, as in calculator/calc. However, you can specify a different name with the component's com.sybase.jaguar.component.naming property. For example, you can specify a logical name, such as USA/MyCompany/FinanceServer/Payroll. For more information on configuring the naming service, see the EAServer documentation.

Examples

In this example, the first argument to the String_To_Object function includes the URLs for two servers in a cluster:

```
// PowerBuilder objects
JaguarORB my_JaguarORB
CORBAObject my_corbaobj
n_bank_acct my_acct

// Proxy objects
Manager my_manager
Session my_session
Factory my_factory

long ll_return
my_JaguarORB = CREATE JaguarORB

// Initialize the ORB
```

```
ll_return = my_JaguarORB.init("ORBRetryCount=3,
    ORBRetryDelay=1000")

// Convert a URL string to an object reference
ll_return = my_JaguarORB.String_To_Object
    (''iiop://JagOne:2000;iiop://JagTwo:2000'',
    my_corbaobj)

// Narrow the object reference to the Manager interface
ll_return = my_corbaobj._narrow(my_manager,
    "SessionManager/Manager")

// Create a session object reference
my_session = my_manager.createSession("admin", "")

// Create a Factory for proxy object references to
// the remote interface
my_corbaobj = my_session.lookup("Bank/n_bank_acct ")
my_corbaobj._narrow(my_Factory,
   "SessionManager/Factory")

// Obtain a proxy, narrow it to the remote
// interface, and call a method
my_corbaobj = my_Factory.create()
my_corbaobj._narrow(my_acct, "Bank/n_bank_acct")
my_acct.withdraw(1000.0)
```

In this example, the component is an EJB component. The home interface effectively performs the same role for the EJB that the factory interface does for a CORBA component:

```
JaguarORB my_orb
CORBAObject my_corbaobj
Manager my_mgr
Session my_session
CartHome my_cartHome
Cart my_cart

my_orb = CREATE JaguarORB
my_orb.init("ORBLogFile='c:\temp\orblog'")
my_orb.String_to_Object("iiop://svr1:2000", &
    my_corbaObj)
my_corbaObj._narrow(my_mgr, "SessionManager/Manager" )
my_Session = my_mgr.createSession("admin", "")
my_corbaObj = my_session.lookup("Cart")
my_corbaObj._narrow(my_CartHome, "shopping/CartHome")
my_corbaObj = my_CartHome.create()
```

```
my_Cart.addItem()
```

Using a Connection object
You can use the Lookup function on the Connection object to obtain a reference to the home interface of an EJB component. See "Invoking an EJB component method" on page 520.

Instantiation using the naming service API

Obtaining proxies for CosNaming and SessionManager interfaces

To instantiate a proxy using the CORBA naming service API, you need to generate proxies for the naming service interface and include these proxies in the library list for the client. Use the EAServer Proxy wizard to create a proxy project for the CosNaming module, build the project to create a proxy library, and add the proxy library to the client target's library list. You also need a proxy for the SessionManager module.

Getting an initial naming context

After initializing the ORB, call the Resolve_Initial_References function to obtain an initial naming context and use _Narrow to convert it to a reference to the CORBA naming context interface. You must identify the CosNaming package by including omg.orb in the class name as shown in the example below.

Resolving the naming context

You need to resolve the naming context to obtain a reference to a Factory object for the component and then narrow that reference to the SessionManager::Factory interface. The resolve method takes a name parameter, which is a sequence of NameComponent structures. Each NameComponent structure has an *id* attribute that identifies the component and a *kind* attribute that can be used to describe the component. In the example below, the name has only one component.

Creating a reference to the component's interface

Call the create method of the Factory object to obtain proxies for the component. The create method returns a CORBA object reference that you can convert into a reference to the component's interface using the _Narrow function.

Example

The NamingContext and NameComponent types used in the example are proxies imported from the CosNaming package in EAServer, and the Factory type is imported from the SessionManager package:

```
CORBAObject my_corbaobj
JaguarORB my_orb
NamingContext my_nc
NameComponent the_name[]
```

```
Factory my_Factory
n_jagcomp my_jagcomp

my_orb = CREATE JaguarORB
// Enclose the name of the URL in single quotes
my_orb.init("ORBNameServiceURL='iiop://server1:2000'")

my_orb.Resolve_Initial_References("NameService", &
    my_corbaobj)
my_corbaobj._narrow(my_nc, &
    "omg.org/CosNaming/NamingContext")

the_name[1].id = "mypackage/n_jagcomp"
the_name[1].kind = ""

my_corbaobj = my_nc.resolve(the_name)
my_corbaobj._narrow(my_Factory, &
    "SessionManager/Factory")
my_corbaobj = my_Factory.create("admin","")
my_corbaobj._narrow(my_jagcomp,
    "mypackage/n_jagcomp")

my_jagcomp.getdata()
```

Client- and component-demarcated transactions

Client applications and EAServer components marked as OTS style or Bean Managed can create, control, and obtain information about EAServer transactions using functions of the CORBACurrent context service object. The CORBACurrent object provides most of the methods defined for the CORBA Current interface.

Two-phase commit

Components in a client- or component-demarcated transaction must be running on a server that is using the OTS/XA transaction coordinator. This transaction coordinator supports the two-phase commit protocol, which uses detailed records from all participants to protect against system failures. In the prepare phase, the transaction coordinator obtains a guarantee from every participant in the transaction that it can be committed and writes a prepare record to the log. In the commit phase, the coordinator notifies all participants, resources are released, the transaction is committed, and a commit record is written to the log.

Components using two-phase commit must connect to a database using a PowerBuilder database interface that is XA-compliant.

An OTS/XA transaction coordinator uses XA resources instead of connection caches to manage transactions. For more information about creating and managing XA resources, see the EAServer documentation.

Creating components that can manage transactions

To create an EAServer component that can manage transactions, check the OTS Style box in the EAServer Project wizard or the Project painter. You can also select OTS Style on the Transaction tab of the property sheet for the component in EAServer Manager or Bean Managed in the Management Console after you have deployed the component.

Initializing the CORBACurrent object

Before you can invoke the functions of the CORBACurrent context service object, you need to create an instance of the object using the GetContextService function, and then initialize it using the Init function.

For transactions managed by a component, call the Init function with no arguments:

```
GetContextService("CORBACurrent", myCorbCurr)
myCorbCurr.Init()
```

For client-demarcated transactions, you must call the Init function with an argument: either an instance of the Connection object with which you have already established a connection, or a URL that identifies a valid EAServer host.

Because the Connection object is more portable, using it is the preferred technique.

```
myCorbCurr.Init( myconnect )
// OR
myCorbCurr.Init( "iiop://localhost:2000")
```

Beginning and ending transactions

You begin a client- or component-demarcated transaction by calling the BeginTransaction function and end it by calling CommitTransaction or RollbackTransaction. Components you instantiate to participate in the transaction must support transactions.

```
// Instance variables:
// CORBACurrent corbcurr
// Connection myconnect


int li_rc
long ll_rc
boolean lb_rc, lb_success
```

```
ll_rc = myconnect.CreateInstance(mycomponent)
li_rc = this.GetContextService("CORBACurrent", &
      corbcurr)
IF li_rc <> 1 THEN
      // handle error
      RETURN
END IF

li_rc = corbcurr.Init( myconnect )
IF li_rc <> 0 THEN
      // handle error
      RETURN
END IF

lb_rc = corbcurr.BeginTransaction()
// perform some processing on the server and
// test for success
...
IF lb_success THEN
      corbcurr.CommitTransaction()
ELSE
      corbcurr.RollbackTransaction()
END IF
```

No nested transactions
You cannot begin a second transaction until the first transaction has been committed or rolled back.

If a component is marked as OTS style, EAServer does not start a transaction when the component is instantiated. EAServer expects the component to start a transaction by calling the BeginTransaction function on an instance of a CORBACurrent object.

Do not call SetComplete
A component should not begin a transaction and then call SetComplete before committing or rolling back the transaction. The transaction will be orphaned until it either times out or is picked up by another transaction.

Getting information about the transaction

CORBACurrent provides two functions for obtaining information about the transaction: GetStatus and GetTransactionName. GetStatus returns an Integer that indicates whether the transaction is active, has been marked for rollback, is in the prepare phase or commit phase, or has been committed or rolled back. GetTransactionName returns a String that identifies the current transaction. It is intended for use in debugging.

Suspending and resuming a transaction

A calling thread can suspend a transaction while the thread performs some non-transactional processing and then resume it. SuspendTransaction returns a handle to the transaction that can be passed to the ResumeTransaction function. ResumeTransaction can be called from a different thread in the same execution context. In this example, the transaction is reassociated with the same thread:

```
long ll_rc
unsignedlong ll_handle
...
ll_rc = corbcurr.BeginTransaction()
// do some transactional work
ll_handle = corbcurr.SuspendTransaction()
// do some non-transactional work
corbcurr.ResumeTransaction(ll_handle)
// do some more transactional work
```

Setting a timeout period for transactions

A calling thread can specify a timeout period after which a transaction will be rolled back. This example sets the timeout period to three minutes (180 seconds):

```
integer li_rc

li_rc = this.GetContextService("CORBACurrent", &
      corbcurr)
IF li_rc <> 1 THEN
      // handle error and return
END IF
li_rc = corbcurr.Init()
IF li_rc <> 1 THEN
      // handle error and return
END IF
corbcurr.SetTimeout(180)
corbcurr.BeginTransaction()
```

Requesting a message back from the server

Simulating server push

A client application *cannot* pass a PowerBuilder object reference to EAServer. Therefore, you cannot use a PowerBuilder object reference to push messages from the server back to a PowerBuilder client. However, you can simulate this behavior by using a shared object on the client to communicate with EAServer. This technique can be thought of as client pull, because the shared object on the client pulls data back from the server.

How it works

To simulate server push, the client uses the SharedObjectRegister and SharedObjectGet functions to create a shared object. Once the object has been created, the main thread on the client makes an asynchronous call to a method on the shared object, passing it a callback object that should be notified when processing has finished on the server. The method on the shared object makes a synchronous call to the EAServer component method that performs processing. Since the shared object is running in a separate thread on the client, the main thread on the client can proceed with other work while the process is running on the server.

Asynchronous processing in EAServer

In this example, POST is used to make an asynchronous call to a method on a shared object on the client. Using POST is *not* supported in the context of calls to EAServer components. For information about asynchronous processing in EAServer, see the EAServer documentation for the ThreadManager and MessageService modules.

For EAServer 6.x, see the Automated Configuration Guide at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.dc00548_0600/html/easautoconfig/title.htm for more information about the Thread Manager. See the Java Message Service User's Guide at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.dc00486_0600/html/easjms/title.htm for information about using the message service.

Example

This example shows how you might use a shared object to make an asynchronous request against an EAServer component method and return data to a client application window.

Client application window

The client application has a window called w_employee that displays employee data in a DataWindow control. When the user clicks the Retrieve button in the window, the client creates a shared object that communicates with EAServer. In addition, it creates an instance of a user object that is used to handle callbacks from the shared object.

Instance variables

The w_employee window has these instance variables defined:

```
uo_sharedobject iuo_sharedobject
uo_callback iuo_callback
```

Retrieve button

The Retrieve button creates the shared object that will communicate with EAServer. In addition, it creates an instance of the user object that will be used to handle callbacks from the shared object. To allow the callback object to notify the window of the completion of processing, the script calls a function called PassObject on the callback object, passing it a reference to the window. Finally, it makes an asynchronous call to the RetrieveData function on the shared object, passing it a reference to the callback object.

The Retrieve button has the following script:

```
long ll_rv

SharedObjectRegister("uo_sharedobject","myshare")
SharedObjectGet("myshare",iuo_sharedobject)

iuo_callback = CREATE uo_callback
iuo_callback.passobject (parent)

iuo_sharedobject.post retrievedata(iuo_callback)
```

SetDW function

The SetDW function applies the contents of the DataWindow Blob returned from the EAServer component to a DataWindow control in the window. The SetDW function takes the argument ablb_data, which is of type Blob, and returns a Long value. The function has the following script:

```
long ll_rv

ll_rv = dw_employee.SetFullState(ablb_data)
if ll_rv = -1 then
   MessageBox("Error", "SetFullState call failed!")
end if

return ll_rv
```

EAServer component

The EAServer component is a PowerBuilder user object called uo_employee. The uo_employee object has a function called RetrieveData that uses a DataStore to retrieve employee rows from the database.

Instance variables

The uo_employee object has these instance variables defined:

```
protected TransactionServer txnsrv
protected DataStore ids_datastore
```

RetrieveData function

The RetrieveData function takes the argument ablb_data, which is of type Blob, and returns a Long value. The function has the following script:

```
long ll_rv
ll_rv = ids_datastore.Retrieve()
ll_rv = ids_datastore.GetFullState(ablb_data)
txnsrv.SetComplete()
return ll_rv
```

Shared object definition

The client application uses a shared object called uo_sharedobject to communicate with the EAServer component. The shared object has a single function called RetrieveData.

Instance variables

The uo_sharedobject object has these instance variables defined:

```
uo_employee iuo_employee
n_jagclnt_connect myconnect
```

Constructor event

The Constructor event uses a custom Connection object called n_jagclnt_connect to connect to the server. Then it creates an instance of the EAServer component:

```
long ll_rc, ll_rv
myconnect = create n_jagclnt_connect
ll_rc = myconnect.ConnectToServer()
ll_rv = myconnect.CreateInstance(iuo_employee, &
    "uo_employee")
```

RetrieveData function

The RetrieveData function makes a synchronous call to the RetrieveData function on the EAServer component. When the function completes processing, it calls the Notify function on the callback object, passing it the DataWindow Blob returned from the server component.

The RetrieveData function takes an argument called auo_callback, which is of type uo_callback:

```
blob lblb_data
```

```
long ll_rv
ll_rv = iuo_employee.retrievedata(lblb_data)
auo_callback.notify(lblb_data)
return ll_rv
```

Callback object definition

When the EAServer component has finished processing, the shared object notifies a user object called uo_callback, which in turn notifies the w_employee window. The uo_callback object has two functions, Notify and PassObject.

Notify function

The Notify function calls a function called SetDW on the w_employee window, passing it the DataWindow Blob returned from the server component. The Notify function takes the argument ablb_data, which is of type Blob, and returns a Long value. The function has the following script:

```
long ll_rv
ll_rv = iw_employee.setdw(ablb_data)
if ll_rv = -1 then
   MessageBox("Error", "SetDW call failed!")
end if
return ll_rv
```

PassObject function

The PassObject function caches a reference to the w_employee window in the *iw_employee* instance variable. The function takes the argument aw_employee, which is of type w_employee, and returns a Long value:

```
iw_employee = aw_employee
return 1
```

Handling errors

PowerBuilder provides three layers of error handling that can be used by clients connecting to EAServer:

- A mechanism, using try/catch/finally blocks, for handling exceptions thrown by components running in EAServer

  All system and runtime errors are converted into objects that descend from the type RuntimeError.

- The Error event on the Connection and JaguarORB objects to handle errors that occur in the context of an EAServer connection

- The SystemError event on the Application object to handle errors that have not been trapped by any other mechanism

PowerBuilder records information about errors in a built-in Error structure. This structure is used by the Error and SystemError events.

What the client can do

A client application can handle communications errors in a number of ways. For example, if a client connects to a server and tries to invoke a method for an object that does not exist, the client can disconnect from the server, connect to a different server, and retry the operation. Alternatively, the client can display a message to the user and give the user the opportunity to control what happens next.

When an error occurs, if the client connects to a new server to retry the operation, it must instantiate the remote object on the new server before invoking a method of the remote object.

Where errors are handled

This is the sequence in which PowerBuilder executes error-handling code in an EAServer client:

1. If an error occurs in the context of the Connection or JaguarORB object, and the Error event of that object has a script associated with it, PowerBuilder executes the event script, if any.
2. If any one of the following is true, any active exception handler for a RuntimeError or its descendants is invoked:
   - The Error event is not scripted
   - The action argument of the Error event is set to ExceptionFail!
   - The error does not occur in the context of the Connection or JaguarORB object
3. If no exception handler exists, or if the existing exception handlers do not handle the exception, the SystemError event on the Application object is executed. If the SystemError event has no script, an application error occurs and the application is terminated.

System exception handler

PowerBuilder has a system exception handler that tries to catch fatal system errors. It helps prevent the server from terminating because of problems with individual client connections. Whenever a fatal error occurs in a client connection, PowerBuilder tries to terminate the client connection without bringing down the server or interfering with other client connections. Once the problem has been detected, the system exception handler triggers a SystemError event in the Application object, just as it would for any other runtime error on the client.

Context-sensitive error handling

Using the try/catch mechanism lets you handle errors where they occur, making it less likely that an error thrown by a component results in a fatal error in the client application. Scripting the Error event of the Connection object is less precise and, unless the action argument of the Error event is set to ExceptionFail!, bypasses any try/catch exception handlers.

You should therefore leave the Error event unscripted and add try/catch blocks to your code to achieve the most effective error handling. You can use the GetMessage function to retrieve the exception's error message.

For more information about exceptions, see "Exception handling in PowerBuilder" on page 36.

Because your error-handling code might not trap all the errors that occur, you should always script the SystemError event of the Application object.

Handling CORBA exceptions

CORBA provides a standard way for components to indicate errors or warnings. CORBA supports two types of exceptions:

- System exceptions
- User-defined exceptions

A system exception is one of a standard set of errors raised by the server. These exceptions are defined in the CORBA specification.

A user-defined exception is an error or warning defined in the component's IDL. A user exception is a new datatype that describes a set of data elements that are returned to the client when the exception is raised.

System exceptions

In PowerBuilder, CORBA system exceptions are mapped to a set of objects that inherit from the RuntimeError object. To see a list of these exceptions, select CORBASystemException on the System tab in the PowerBuilder Browser, select ShowHierarchy from the pop-up menu, and expand the Treeview item.

The names of the CORBASystemException objects in PowerBuilder map to the names of CORBA system exceptions as defined in the CORBA/IIOP Specification with underscore characters omitted. For example, the PowerBuilder CORBACommFailure exception maps to the CORBA_COMM_FAILURE exception. For detailed information about CORBA exceptions, see the CORBA/IIOP Specification, which can be downloaded from the OMG Web site at http://www.omg.org/.

You might want to provide error handling for the following exceptions when you invoke methods on a component:

```
TRY
... // invoke methods
CATCH (corbacommfailure cf)
... // A component aborted the EAServer transaction,
    // or the transaction timed out. Retry the
    // transaction if desired.
CATCH (corbatransactionrolledback tr)
... // possibly retry the transaction
CATCH (corbaobjectnotexist one)
... // Received when trying to instantiate
    // a component that does not exist. Also
    // received when invoking a method if the
    // object reference has expired
    // (this can happen if the component
    // is stateful and is configured with
    // a finite Instance Timeout property).
    // Create a new proxy instance if desired.}
CATCH (corbanopermission np)
... // tell the user they are not authorized
CATCH (corbasystemexception se)
... // report the error but don't bother retrying
FINALLY
    // put cleanup code here
END TRY
```

User-defined exceptions

User-defined exceptions are mapped to the CORBAUserException object, which inherits from the Exception object. PowerBuilder clients can handle exceptions thrown by any component type.

If an EAServer component has a method on it that is defined to throw an exception, that method on the PowerBuilder proxy object is also declared to throw a user-defined exception. The definition of the user-defined exception is created when you create the component proxy.

CORBA does not support exception hierarchies
Exception hierarchies are not supported in the CORBA IDL. As a result, when you generate proxies for a server component that has an inherited exception, the generated proxies all inherit directly from CORBAUserException.

All Create, Remove, and FindByPrimaryKey methods on EJB components throw the EJB CreateException, RemoveException, and FinderException exceptions. These exceptions are represented by IDL exceptions with the same name in the CtsComponents package in EAServer.

Scripting the Error event

What you do

To handle errors in the Error event of the Connection object, you create a user object that customizes the definition of the object. Once you have created the custom Connection object, you can refer to it anywhere in your scripts where you use a Connection object. If you use the JaguarORB event, you can script its Error event in the same way.

The Connection Object wizard creates a custom Connection object for you. See "Using the wizard to create a Connection object" on page 515.

Arguments to the Error event

The Error event of the custom Connection object has several arguments that provide information about the condition that caused the error. For example, these arguments provide the error number and error text, as well as the name of the object that caused the error and the full text of the script where the error occurred.

In addition to the arguments that provide error information, the Error event has an argument that lets you specify what action to take. To specify the action, you assign one of four enumerated values (ExceptionFail!, ExceptionRetry!, ExceptionIgnore!, or ExceptionSubstituteReturnValue!) to the Action argument of the Error event.

Example

In this example, the Error event script informs the user of the condition that caused the communications error and gives the user the opportunity to control what happens next. Depending on the user's input, the client application fails, retries the operation, or ignores the error and continues processing:

```
int li_choice
li_choice = MessageBox("Connection error " + &
    string(ErrorNumber), ErrorText, &
    Question!,AbortRetryIgnore!)
CHOOSE CASE li_choice
    CASE 1
       Action = ExceptionFail!
    CASE 2
       Action = ExceptionRetry!
    CASE 3
       Action = ExceptionIgnore!
END CHOOSE
```

Scripting the SystemError event

What you do

In the SystemError event of the Application object, you can write a script to tell PowerBuilder to halt application execution or ignore the error.

Example

In this example, the SystemError event script displays a message informing the user of the condition that caused the communications error and gives the user the opportunity to control what happens next. Depending on the user's input, the client application halts execution or ignores the error and continues processing:

```
string ls_logline = "SYSTEM ERROR: "
ls_logline += String(error.number) + " " + error.text
ls_logline += " occurred at line " + &
      String(error.line) + " "
ls_logline += " in event " + error.objectevent
ls_logline += " of object " + error.object

if Messagebox("System Error", ls_logline + &
   "~r~n~r~nDo you want to stop the program?", &
   Question!, YesNo!) = 1 then
   HALT CLOSE
end if
```

Deploying the client application

The procedure for deploying a client application in a distributed computing environment is more or less the same as the procedure for deploying any other PowerBuilder application. You have two basic ways to package your client application:

- As a standalone executable (*EXE*) file that contains all the objects in the application
- As an executable file and one or more dynamic libraries

You might also need to provide some additional resources that your application uses, such as bitmaps and icons. You can provide resources in your executable and/or dynamic libraries, or you can distribute them separately.

For complete instructions on creating an executable version of your application, see the PowerBuilder *User's Guide*. For information about the files you need to deploy with your application, see Chapter 41, "Deploying Applications and Components."

CHAPTER 26

Using SSL in PowerBuilder clients

PowerBuilder clients can use the Secure Sockets Layer (SSL) to connect to EAServer. Among other security features, SSL provides certificate-based authentication of the server, optional certificate-based authentication of the client, and optional encryption of data transmitted over the network.

Contents

Using secure connections with EAServer

The SSL protocol allows connections to be secured using public-key encryption and authentication algorithms that are based on digital certificates. SSL is a wrapper protocol: packets for another protocol are secured by embedding them inside SSL packets. For example, HTTPS is HTTP secured by embedding each HTTP packet within an SSL packet. Similarly, IIOPS is IIOP embedded within SSL.

EAServer’s built-in SSL driver supports dynamic negotiation, cached and shared sessions, and authorization for client and server using X.509 Digital Certificate support.

For an overview of security in EAServer and more information about EAServer and SSL, see the EAServer documentation. For EAServer 6.x, see the Security Administration and Programming Guide at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.dc38035_0600/html/eassec/title.htm.

Quality of protection

The quality of protection (QOP) for EAServer packages, components, and methods can be set in the Management Console. QOP establishes a minimum level of encryption and authentication that a client must meet before it can access a component's business logic. For example, to set the quality of protection for a component, add the com.sybase.jaguar.component.qop property on the All Properties page of the component's property sheet and set it to a security characteristic provided with EAServer, such as sybpks_intl.

For a description of configuring QOP on the server and a list of security characteristics provided with EAServer, see the EAServer documentation. This chapter describes configuring QOP on the client.

SSL certificate-based authentication

In the Management Console, you can configure a secure IIOP or HTTP port by configuring a listener and associating a security profile with the listener. The profile designates a security certificate to be sent to clients to verify that the connection ends at the intended server, as well as other security settings.

PowerBuilder clients need a public key infrastructure (PKI) system to manage digital certificates. You can use Security Manager, which manages the EAServer certificate database.

For more information about PKI and configuring secure ports and authentication options, see the EAServer documentation.

Client installation requirements

EAServer provides several sets of client runtime files. Because SSL support in PowerBuilder clients is provided through the client ORB, you should install the SSL runtime files on the computer on which PowerBuilder SSL clients will run. The installation includes the client-side security database, SSL support libraries, and the client-side Security Manager. You also need to configure the client installation to load the client libraries when you run your application. See the EAServer *Installation Guide* for more information.

SSL connections in PowerBuilder

PowerBuilder provides two system objects for use with secure connections:

- **SSLServiceProvider service object** The SSLServiceProvider object is an implementation of the EAServer CtsSecurity::SSLServiceProvider interface. For more information about this interface, you can view the EAServer interface repository documentation in a Web browser by connecting to your server at http://*hostname*:*portnumber*.

You use the GetGlobalProperty and SetGlobalProperty functions of the SSLServiceProvider object to configure global SSL properties. For a description of the global properties you can set and/or get, see “SSL properties” on page 543.

You can also set SSL properties at the connection level by specifying them in an options string for the Connection or JaguarORB object. Interactive applications typically use the SSLServiceProvider object in conjunction with the SSLCallback object. Applications that run without user interaction typically configure SSL settings at the connection level. For information about setting properties at the connection level, see “ORB properties” on page 545.

- **SSLCallback object** To enable EAServer to request additional information from the client using a callback method, you can implement your own logic for the callback methods in an instance of the SSLCallBack object. The SSLCallback object is an implementation of the EAServer CtsSecurity::SSLCallback interface.

SSL properties

Table 26-1 lists the properties that can be set or retrieved using SetGlobalProperty or GetGlobalProperty. For any SSL connection, you must set the qop (quality of protection) property and, unless you implement a callback to obtain it, you must also set the pin property. You also need to connect to a server address that can support your chosen level of security, as described in “Secure server addresses” on page 545.

Setting global properties in a PowerBuilder session
When you run a client application in PowerBuilder, you can set global properties only *once* during the PowerBuilder session. You will need to restart PowerBuilder each time you test the code that sets global SSL properties.

If some properties are not set or are set incorrectly, an SSL callback method is invoked. If you do not specify an instance of the SSLCallback object, the default callback implementation aborts the connection attempt.

Table 26-1: List of SSL properties

Property	Description	Get	Set
callbackImpl	Instance of the SSLCallback object. For more information, see “Using SSL callbacks” on page 549.	Yes	Yes

Property	Description	Get	Set
certificateLabel	The client certificate to use if the connection requires mutual authentication. The label is a simple name that identifies an X.509 certificate/private key in a PKCS #11 token. Required for mutual authentication. If not set and the connection requires mutual authentication, invokes the getCertificateLabel callback method, passing an array of available certificate names as an input parameter.	Yes	Yes
qop	The name of a security characteristic to use. Required for SSL. See “Choosing a security characteristic” on page 545 for more information.	Yes	Yes
cacheSize	The size of the SSL session ID cache. Default is 100.	Yes	Yes
SessLingerTime	The number of seconds that a session ID entry is kept in the cache after the last connection that used it is terminated. Default is 28800 seconds (8 hours).	Yes	Yes
SessShareCount	The number of concurrent SSL sessions that can use the same session ID. Default is 10.	Yes	Yes
pin	The PKCS #11 token PIN. This is required for logging in to a PKCS #11 token for client authentication and for retrieving trust information. Required for SSL. If not set, set to `any`, or set incorrectly, the getPin callback method is invoked.	No	Yes
availableQop	A list of available security characteristics. The qop property can be set only to values that appear in this list.	Yes	No
availableQopDesc	A list of descriptions for the available security characteristics, in the same order as listed in the value of the availableQop property.	Yes	No
availableVersions	A list of SSL protocol versions supported by the SSL runtime engine.	Yes	No
entrustReady	TRUE if Entrust PKI software is available on the client, FALSE otherwise.	Yes	No
entrustIniFile	The path name for the Entrust INI file that provides information on how to access Entrust. Required when the useEntrustid property is set to true. If not set, the getCredentialAttribute callback method is invoked.	Yes	Yes
entrustUserProfile	The full path to the file containing an Entrust user profile. Optional when the Entrust single-login feature is available, required otherwise. If not set, the getCredentialAttribute callback method is invoked.	Yes	Yes

Property	Description	Get	Set
useEntrustID	Whether to use the Entrust ID or the Sybase PKCS #11 token for authentication. This is a boolean property. If this property is set to FALSE, Sybase PKCS #11 token properties are valid and Entrust-specific properties are ignored. If this property is set to TRUE, Entrust-specific properties are valid and Sybase PKCS #11 token properties are ignored.	Yes	Yes
entrustPassword	The password for logging in to Entrust with the specified user profile. Optional when the Entrust single-login feature is available, required otherwise. If the password is required but not set or set incorrectly, the getPin callback method is invoked.	No	Yes

Choosing a security characteristic

To use SSL, you must specify the name of an available security characteristic for the qop property. The characteristic describes the CipherSuites the client uses when negotiating an SSL connection. When connecting, the client sends the list of CipherSuites that it uses to the server, and the server selects a CipherSuite from that list. The server chooses the first CipherSuite in the list that it can use. If the server cannot use any of the available CipherSuites, the connection fails.

The EAServer documentation describes the security characteristics that are provided with EAServer. You can retrieve a list of characteristics available on the server and their descriptions by retrieving the availableQop and availableQopDesc properties with GetGlobalProperty.

Secure server addresses

You can connect only to a server listener that uses a security level that is equivalent to or greater than the level requested in the qop setting. If you use JaguarORB.string_to_object to instantiate a proxy for the SessionManager::Manager interface, the listener specified by the server address must use a security profile that matches the client’s qop setting.

ORB properties

When you connect to EAServer using either the Connection object or the JaguarORB object, you are using the EAServer client ORB. You can set its properties in the Options string of the Connection object or using JaguarORB’s Init function. These are the ORB properties that apply specifically to secure connections:

- ORBqop
- ORBcertificateLabel

- ORBpin
- ORBuseEntrustID
- ORBentrustPassword
- ORBentrustIniFile
- ORBentrustUserProfile

The meaning of each of these properties is the same as that of the corresponding SSL property, but the value affects only the connection that is being established and not the entire session. Set ORBqop to `sybpks_none` to prevent any use of SSL on a connection. This setting is useful if you have set the QOP globally for all ORBs using the SSLServiceProvider object, and you want to override the QOP for a single connection.

For a complete list of ORB properties, see the Help for the Connection object.

This example sets the ORBqop property to sybpks_simple and specifies a log file:

```
myconnect.options = "ORBqop='sybpks_simple', " &
   + "ORBLogFile='C:\tmp\log.txt'"
```

Establishing a secure connection

To establish a secure connection to EAServer, follow these steps:

1. Create an instance of the SSLServiceProvider object.
2. Optionally use the GetGlobalProperty function to obtain security information from the server.
3. Set properties required by the server using the SetGlobalProperty function.
4. Connect to the server using the ConnectToServer function of the Connection object.

Creating an instance of SSLServiceProvider

This code creates an instance of the SSLServiceProvider object:

```
SSLServiceProvider sp
GetContextService( "SSLServiceProvider", sp )
```

Getting information from the server

Use GetGlobalProperty to obtain information about the security characteristics of the server. This example gets information about supported CipherSuites from the availableQop property, and displays the information in a drop-down list:

```
int i, rc
string ls_values[]

rc = sp.GetGlobalProperty("availableQop", ls_values)

IF rc <> 0 THEN
   MessageBox("Get Qop Failed", "rc = " + string(rc))
   RETURN
END IF

FOR i = 1 to UpperBound(ls_values)
   ddlb_1.AddItem( ls_values[i] )
NEXT
RETURN
```

Setting global properties

Before you connect to the server, you must set required global properties. This code sets qop to the value sybpks_intl and pin to the value sybase:

```
int rc

rc = sp.SetGlobalProperty( "qop", "sybpks_intl" )
IF rc <> 0 THEN
   MessageBox( "Setting QOP Failed", &
      "rc = " + string(rc) )
ELSE
   MessageBox( "Set SSL QOP Property", "succeeded" )
END IF
rc = sp.SetGlobalProperty( "pin", "sybase" )
IF rc <> 0 THEN
   MessageBox( "Setting PIN Failed", &
      "rc = " + string(rc) )
ELSE
   MessageBox( "Set SSL PIN Property", "succeeded" )
END IF
```

Most of the properties set using SetGlobalProperty can be set only once for the lifetime of the client executable. The properties remain in effect when the client disconnects from or reconnects to the server.

Restarting PowerBuilder
When you run a client application in PowerBuilder, you can set global properties only *once* during the PowerBuilder session. You will need to restart PowerBuilder each time you test the code that sets global SSL properties.

If you want to use an instance of the SSLCallback object to obtain user input interactively, you need to set the global property CallBackImpl. See "Using SSL callbacks" on page 549.

Connecting to the server

When you begin a secure session, the client and server exchange messages in the SSL handshake process. The client provides information that the server needs in order to communicate with the server, then the server must always authenticate itself to the client before the process can continue. If the server requires client authentication, the client must be authenticated before the process can continue. When the required authentication is complete, the client and server create symmetric keys that will be used for encryption, decryption, and tamper detection in the SSL session. To catch any exceptions that are raised during this process, you should enclose the ConnectToServer call in a try-catch block.

When you establish a secure connection, use `iiops` instead of `iiop` in the location property of the connection object. The server typically listens for secure connections on ports 2001 or 2002. This example uses a Connection object, g_connect, that has been declared as a global variable. The example uses the options property of the Connection object to specify a different CypherSuite for this connection:

```
long l_rc
g_connect.userid   = sle_user.text
g_connect.password = sle_password.text
g_connect.driver   = "jaguar"
g_connect.application = "myserverpkg"
g_connect.location = "iiops://myserver:2001"
g_connect.options = "ORBqop='sybpks_simple'"

TRY
   l_rc = g_connect.ConnectToServer()
CATCH (userabortedexception uae)
   MessageBox("UserAbortedException Caught", &
      "ConnectToServer caught: " +  uae.getMessage() )
   l_rc = 999

CATCH ( CORBASystemException cse )
   MessageBox("CORBASystemException Caught", &
      "ConnectToServer caught: " +  cse.getMessage() )
   l_rc = 998
END TRY
IF l_rc <> 0 THEN
   MessageBox("Error", "Connection Failed - code: " &
      + string(l_rc) )
   MessageBox("Error Info", "ErrorCode= " &
      + string(g_connect.ErrCode) + "~nErrText= " &
      + g_connect.ErrText)
ELSE
   MessageBox("OK", "Connection Established")
END IF
```

Troubleshooting connections

When a secure connection fails, the error message that displays is the same as for insecure connections. It does not provide any additional information about the reason for failure. To obtain more information in a log file, you can enable the ORBLogIIOP option and specify a value for the ORBLogFile option. In the example above, you would replace the `g_connect.options` statement with something like this:

```
g_connect.options = "ORBqop='sybpks_simple'" + &
  "ORBLogIIOP='TRUE', ORBLogFile='d:\temp\ORBLog.txt'"
```

Alternatively, you can set the JAG_LOGFILE environment variable to specify the log file in which initialization errors are recorded.

Using SSL callbacks

The SSLCallback object handles SSL requests for additional authentication information from a server to a client application. The C++ ORB invokes callback methods when a required setting, such as a pin, has not been specified, or when the value specified is invalid.

The callback can respond to exceptional conditions, such as server certificates that have expired. When using mutual authentication, the callback getCertificateLabel method allows you to present a list of available certificates to the user. Using a callback can also simplify handling of retry logic when the user enters an invalid certificate or password.

To use the SSL callback mechanism, you need to follow these steps:

1 Create proxy objects for the CTS Security module in EAServer to obtain SSL session information.

2 Create a standard custom class user object inherited from the SSLCallback object and implement the callback functions you need.

3 Set the global SSL property CallBackImpl to the name of your SSLCallback object and connect to the server.

Getting session information

SSL callback functions all have access to the SSL session information. You should use this information to provide the user of the client application with information needed to supply the required authentication information.

To make the SSL session information available to the callback functions, create an EAServer proxy for the CTSSecurity module.

❖ **To create a proxy for the CTSSecurity module:**

1 Select the EAServer Proxy wizard from the Project page in the New dialog box and select your client application target from the Target drop-down list.

2 Connect to any EAServer host and select the CTSSecurity module.

The CTSSecurity module is a standard module that is available on all servers.

3 Complete the wizard and build the project.

Among the proxy objects you will see in the System Tree is the Sessioninfo object that is passed to all the SSLCallback functions.

Implementing the SSLCallback object

There are four callback functions.

Table 26-2: SSL callback functions

Function	When it is called
GetCertificateLabel	Called when the client application has not set a certificate label for client authentication and the server has requested client authentication.

Function	When it is called
GetCredentialAttribute	Called when the client application has not set credential attributes. These attributes are used when the client application has set the UseEntrustId property using the SSLServiceProvider object. GetCredentialAttribute is useful only if you are using Entrust IDs. For more information about Entrust and PKCS 11 tokens, see the EAServer documentation.
GetPin	Called if the PKCS11 token is not logged in and the PIN has not been set as a property of the SSLServiceProvider object. It can also be called if the login session has timed out.
TrustVerify	Called when the server's internal SSL trust verification check fails to verify the server's certificate chain or when the pin to log in to the Sybase PKCS11 token was not supplied or is incorrect. TrustVerify can be invoked when you are using any SSL protocol, because server authentication is a required step in the SSL handshake process. The user can choose whether to override the internal check and proceed with the SSL connection.

Each of these functions is implemented by the SSLCallback class and has a default implementation. You need to implement any function for which you want to use the callback. For sample implementations of each function, see the *PowerScript Reference* or the online Help.

❖ **To implement the SSLCallBack class:**

1 Select Standard Class from the PBObject page of the New dialog box.

2 Select SSLCallback in the Select Standard Class Type dialog box and click OK.

3 Code a callback function to provide the user with information about the session and prompt the user to return the required authentication information.

4 Repeat step 3 for any other callback functions you want to implement.

Default implementations

If you do not provide an implementation, or if your implementation returns an empty string, the default implementation of the callback is used.

For both GetCertificateLabel and GetCredentialAttribute, the argument list includes an array of string values that are valid return values for the callback. The default implementation of these callbacks throws an exception if the array is empty, and returns the first value in the array if it exists. As a result, the connection process continues if the first value in the array is acceptable to the server, but fails if the value is unacceptable.

For TrustVerify, the default implementation rejects the current connection.

Handling exceptions

Your implementation of GetPin, GetCertificateLabel, and GetCredentialAttribute should allow users to cancel the connection if they are unable to provide the requested information. You can do this by throwing an exception in your implementation of the function and catching it in a try-catch block that surrounds the call to ConnectToServer. Exceptions thrown in any of the callback functions raise the CTSSecurity::UserAbortedException exception. You should add any exceptions that can be thrown by the function to the throws clause of the function’s prototype.

Specifying the SSLCallback object

Before you connect to the server, specify the name of your SSLCallback object in the CallbackImpl property of SSLServiceProvider:

```
SSLServiceProvider sp
int    rc

getcontextservice("SSLServiceProvider", sp)
rc = sp.setglobalproperty( "CallbackImpl", &
   "uo_sslcallback" )
IF rc <> 0 THEN
   MessageBox("Set CallbackImpl Failed", "rc= " + &
      string(rc))
   RETURN
END IF
MessageBox( "Set CallbackImpl Property", "succeeded" )
RETURN
```

To make sure that the executable version of your client application can reference your callback object, you need to declare a variable of its type in your application, for example:

```
uo_sslcallback iuo_sslcb
```

This is because the callback object is referenced only by its string name so that it is technically an unreferenced object and is not included in the executable file. Your code does not need to use the declared variable.

Retrieving session security information

The CtsSecurity.SSLSession and CtsSecurity.SSLSessionInfo classes allow a client application to determine whether SSL is used on connections from a proxy to the server, and if so, to retrieve the SSL session settings and display them to the user. For a list of the properties for which you can retrieve values, see the EAServer interface repository documentation for SessionInfo in a Web browser by connecting to your server at http://*hostname*:*portnumber*/ir/CtsSecurity__SSLSessionInfo.html.

```
long rc
string stmp

CTSSecurity_sslSessionInfo mySessionInfo
rc = thesessioninfo._narrow( mySessionInfo, &
   "SessionInfo" )
MessageBox( str_header, "Narrow: rc=" + string(rc))

sTmp  = "Properties"
sTmp  = "~nVersion: "
stmp += mySessionInfo.getProperty( "Version" )
sTmp  = "~nHost: "
stmp += mySessionInfo.getProperty( "host" )

stmp += "~nport: "
stmp += mySessionInfo.getProperty( "port" )
stmp += "~nciphersuite: "
stmp += mySessionInfo.getProperty( "ciphersuite" )
stmp += "~nCertificateLabel: "
stmp += mySessionInfo.getProperty( "certificateLabel"
)
stmp += "~nUserData: "
stmp += mySessionInfo.getProperty( "UserData" )
stmp += "~ntokenName: "
stmp += mySessionInfo.getProperty( "tokenName" )
stmp += "~nuseEntrustID: "
stmp + = mySessionInfo.getProperty( "useEntrustID" )
MessageBox( str_header, stmp)
```

CHAPTER 27

Building a COM or COM+ Component

About this chapter

This chapter explains how to use PowerBuilder to build a COM or COM+ component.

Contents

About building COM and COM+ components

You can develop a custom class user object containing business logic in PowerBuilder and then package the object as a COM server or COM+ application.

On platforms that support COM+, such as Windows 2000 and Windows XP, you can build COM+ applications and deploy them to COM+.

The rest of this chapter uses the term COM components to refer to components that support COM and COM+.

A PowerBuilder COM application can include one or more PowerBuilder custom class user objects. You code the user objects in the User Object painter and then build the server in the Project painter. You can also deploy the application directly to a local COM+ server or create a COM+ Microsoft Installer package from the Project painter.

Once you have generated and deployed a PowerBuilder COM application, users can call methods on the PowerBuilder objects it contains from COM-enabled client applications built with tools such as PowerBuilder, Visual Basic, and C++ compilers.

About using the wizards

PowerBuilder provides several wizards to facilitate the development and deployment of COM components:

- **Target wizard** Creates a new application, a new custom class user object, and a new project
- **Object wizard** Creates a new custom class user object in an existing application and creates a new project
- **Project wizard** Creates a project you use to build a COM server and optionally a COM+ package from one or more existing custom class user objects

When you create a new user object using a COM/COM+ Target or Object wizard:

- The object has two new events: Activate and Deactivate.
- The object has COM validation support enabled.
- If you checked the box on the last page of the wizard, the wizard adds items to a To-Do List to remind you to complete all phases of development.

When to use the wizards

New objects If you plan to build a server containing a single new custom class user object, use the Target or Object wizard to take advantage of these features.

If you plan to build a server containing multiple new custom class user objects, using the Target or Object wizards will give you a separate project for each object, so you will need to add all the objects to one of the projects and discard the other projects.

Alternatively, you can create new custom class user objects without using the Target or Object wizards and then create the project later using the Project wizard or the Project painter.

You can enable COM validation and add the Activate and Deactivate events as user-defined events in the User Object painter. For more information, see “COM validation” on page 563.

Existing objects If you have one or more custom class user objects that you want to deploy as COM servers, make sure that they meet the requirements described in “Defining the component interface” on page 560 and the subsequent sections. As for new user objects, you can enable COM validation and add the Activate and Deactivate events as user-defined events in the User Object painter. When you are ready to build the server, you can use the Project wizard to set up the project.

About the development process

To build and deploy a PowerBuilder COM server from one or more custom class user objects, you need to complete the following steps:

1. Create a new custom class user object or open an existing user object.

   If you use a Target or Object wizard, you also create a project object that you use to build and deploy the new object.

2. Code the user object in the User Object painter.

   See “Defining the component interface” on page 560.

3. Optionally, create and code additional user objects in the same application.

4. If you did not use a wizard to create the user object(s), create a project using the COM/COM+ Component Project wizard or the Project painter.

   See “Building COM/COM+ components in the Project painter” on page 572.

5. Open the project, modify the list of selected objects and their properties if necessary, and build the project.

6. Deploy the server.

   See “Deploying a PowerBuilder COM server” on page 576.

About the Component Object Model

The Microsoft Component Object Model (COM) defines a standard way for software components to supply services to each other. Any PowerBuilder custom class user object can be used as a COM object by providing it with a runtime environment, a registry entry, and optionally a type library. Clients built with COM-compliant tools such as PowerBuilder and Visual Basic can make use of the business logic in the COM object by creating an instance of the object and calling the methods exposed in its interface. Depending on the interfaces it supports, the COM object may also be available to Java and C++ clients.

COM+ enhances COM by handling more resource management tasks and providing thread pooling, object pooling, and just-in-time object activation.

About PowerBuilder COM servers

PowerBuilder creates a single PowerBuilder COM server that contains a PowerBuilder COM object for every custom class user object you select when you build the project.

COM specifies how objects are created and destroyed, how their interfaces are exposed, and how their methods are invoked. PowerBuilder COM servers follow the COM specification; this means that from a client perspective, the fact that a PowerBuilder COM object interacts with a custom class user object (through the PowerBuilder virtual machine) is transparent.

Comparing automation servers and PowerBuilder COM servers

PowerBuilder provides two ways to generate COM objects: in PowerBuilder COM servers or automation servers. Both are accessible from wizards and from the Project painter. PowerBuilder COM servers offer many more features than automation servers.

PowerBuilder COM servers PowerBuilder COM servers can contain more than one custom class user object. After you code the user objects, you use the Project painter to generate a single self-registering DLL for all the objects. You can also deploy the server directly to COM+ if it is running on the build computer, or create a COM+ package. PowerBuilder COM objects in a COM server can share runtime sessions, and references can be passed between objects created from the same COM client.

The COM server also contains an embedded PowerBuilder dynamic library (PBD) file that contains compiled versions of all the custom class user objects you selected and any dependent objects, as well as registry and type library information.

Automation servers Automation servers built from a PowerBuilder custom class user object contain only one object. After you code the user object, you build a runtime library from the PBL that contains it, and then use the Project painter to create a registry file and optional type library file. When you deploy the automation server, you customize the registry file for the user's computer and then run the file to register the automation server.

You deploy the PowerBuilder virtual machine with an automation server or a PowerBuilder COM server, along with any other PowerBuilder runtime files the server requires.

Dispatch, dual, and custom interfaces

Automation servers use a dispatch interface (also called dispinterface) that allows users to invoke methods on the server using the Invoke method of a standard COM interface called IDispatch.

COM servers can use custom or dual interfaces, which provide better performance than dispatch-based interfaces. A custom interface provides access to methods on the server through a virtual table (also called VTBL or vtable) that contains pointers to methods in the server's interface. A dual interface enables the client to invoke methods using IDispatch::Invoke or a virtual table.

For more information, see "Choosing a custom or dual interface" on page 574.

Summary of differences between servers

Table 27-1 summarizes the differences between PowerBuilder automation servers and PowerBuilder COM servers using dual or custom interfaces:

Table 27-1: Comparison of automation servers and COM servers

Feature	Automation server (dispatch interface)	COM server (dual interface)	COM server (custom interface)
In-process server support	Yes*	Yes	Yes
Generated IDL files	No	Yes	Yes
C++ client support	No	Yes	Yes
Java client support	Yes	Yes	No
PowerBuilder client support	Yes	Yes	No
Visual Basic 4 client support	Yes	Yes	No
Visual Basic 5 custom interface support	No	No	Yes

Feature	Automation server (dispatch interface)	COM server (dual interface)	COM server (custom interface)
Support for PowerBuilder structures as instance variables and method argument types	No	No	No
Support for all C language datatypes	No	No	No
Support for an embedded type library	No	Yes	Yes
Self-registering servers	No	Yes	Yes
Support for DCOM #	Yes	Yes	Yes
Support for EAServer	Yes	Yes	No
Support for COM+	No	Yes	Yes
Requires separate proxy/stub DLLs	No	No	No
Requires PowerBuilder runtime DLLs (minimally *PBVM110.DLL* and its dependencies)	Yes	Yes	Yes
Requires deployment of application runtime libraries built in Library painter	Yes	No	No

* In-process server is the PowerBuilder execution DLL (PBVM110.DLL)
# Requires use of surrogate host to house the PowerBuilder execution DLL

Defining the component interface

When you build a PowerBuilder custom class user object as a COM component, the functions and optionally the instance variables defined for the object appear in the component interface. PowerBuilder generates an IDL file that defines a COM class and a single interface for each custom class user object contained in the server, as well as a type library name and an associated ID for the PowerBuilder COM server.

Methods and datatypes

Functions

Each PowerBuilder COM object supports a single interface that exposes a method for each user-defined public function in the custom class user object.

The function's return value is represented by an additional retval argument. For example, if an object has these user object functions:

```
f_addtwo (long al_num1, long al_num2) returns long
f_getinfo (REF string as_name, REF integer ai_age,
    REF character ac_gender) returns integer
```

These member functions are generated in the IDL file:

```
HRESULT f_addtwo( [in] long al_num1,
    [in] long al_num2, [out, retval] long * retval );
HRESULT f_getinfo( [in, out] BSTR * as_name,
    [in, out] short * ai_age, [in, out] unsigned
    char * ac_gender, [out, retval] short * retval );
```

Instance variables

Since COM objects never expose their data, public instance variables in the custom class user object can be represented in the COM object as interface methods for getting and setting the variable value. To specify that variable accessor methods will be exposed in the interface, you can use the Project wizard or the Objects property page in the Project painter.

If the public variable is writable, the put method will be exposed. For private and protected variables and variables declared as privateread or protectedread and privatewrite or protectedwrite, no methods are generated. If the variable is publicly readable, the get method will be exposed. For example, if an object has these instance variables:

```
public string is_name
private integer ii_a
public privatewrite string is_label
constant real lr_pi = 3.14159265
```

These are the methods that are generated in the IDL file:

```
[id(4), propget] HRESULT is_name([out,retval]
    BSTR *is_name);
[id(4), propput] HRESULT is_name([in]
    BSTR is_name);
[id(1), propget] HRESULT is_label([out,retval]
    BSTR *is_label);
[id(6), propget] HRESULT lr_pi( [out,retval]
    float * lr_pi);
```

Datatype mappings

PowerBuilder datatypes map to COM datatypes as shown in Table 27-2.

Table 27-2: PowerBuilder and COM datatype mapping

PowerBuilder datatype	COM datatype (variants)
Boolean	Variant_BOOL
Character	Unsigned char
Integer	Short
UnsignedInteger	Unsigned short
Long	Long
UnsignedLong	Unsigned long
Real	Float
Double	Double
Decimal	Double
String	BSTR
Date	DATE
Time	DATE
DateTime	DATE
Blob	SAFEARRAY (Unsigned char)
Arrays (PowerBuilder datatype)	SAFEARRAY (COM datatype)
ResultSet	LPDISPATCH
Custom class user objects*	LPDISPATCH
Any	*Not supported*
Global structures	*Not supported*
OLEObjects	*Not supported*

* Custom class user objects must be created within the same client in the same COM apartment (that is, in the same thread)

Restrictions on coding

There are some elements that you cannot use in your code when you plan to deploy a user object as a COM component.

No overloaded functions

COM does not support overloaded functions in an interface to a COM object. Each function in the user object (and its ancestors) must have a unique name. PowerBuilder COM objects have a single interface, and multiple functions with the same name but different signatures would require multiple interfaces.

How ancestor variables and ancestor functions are represented

When you generate a PowerBuilder COM object from a descendent user object, the public instance variables and functions of both the ancestor and the descendant are represented in the COM object. The fact that some of the component methods were derived from the ancestor object is transparent. Because of the restriction on overloaded functions described above, functions in descendent objects can override functions in the ancestor, but they cannot overload them.

Datatypes for arguments and return values

The methods associated with a nonvisual object that you deploy as a COM object can take arguments that use the following datatypes:

- Standard OLE automation datatypes
- Custom class (nonvisual) user objects

COM component methods cannot take arguments or return values that use PowerBuilder structures or the Any datatype. Functions defined on a PowerBuilder nonvisual object that take an Any variable as an argument or return an Any variable can be called from code that is local to that object; however, these functions cannot be accessed by clients or other COM components.

The arguments to a component method cannot include visual objects (such as windows or menus) or most system types (such as the Transaction object or the DataStore object). The only system type supported is the ResultSet object.

The return value of a component method can be of any standard datatype. The return value can also be a custom class (nonvisual) user object.

COM validation

If you are designing a custom class user object that you plan to deploy as a COM component, you can have PowerBuilder warn you when you use code elements that are not valid in COM.

COM validation checks for overloaded functions and checks public instance variables and public functions for system types, visual types, structures, and Any variables.

In the User Object painter, make sure the Design>COM Validation menu item is checked. When you save the object, the Output window lists warnings such as the following:

```
Information C0197: Component Validation
Warning C0198: illegal COM type: 'any' arg type for
function: 'of_badfunc'
Warning C0198: illegal COM type: 'window' arg type for
function: 'of_badfunc'
```

Validation is associated with the object you are editing, not with the User Object painter. When you reopen an object, it has the same validation state as when you closed it. New COM objects are created with COM validation checked.

Recording errors in a log file

To record errors generated by COM objects running in COM+ to the Windows system application log, create an instance of the ErrorLogging service context object and invoke its log method. For example:

```
ErrorLogging el
this.GetContextService("ErrorLogging", el)
el.log("Write this string to log")
```

Automatic recording of exception information
Information about the exception type and location of an exception caused by a PowerBuilder component running on the server is recorded automatically in the server log. It is no longer necessary to invoke the error logging service to obtain minimal information about these exceptions.

Accessing a database from a COM component

To take advantage of COM+ support for transaction management, you need to use one of the database interfaces supported by COM+ to connect to your database. For more information about database connections for components developed in PowerBuilder, see *Connecting to Your Database*.

COM components developed in PowerBuilder can use DataStores to interact with the database. DataStores are nonvisual DataWindow controls that act just like DataWindow controls except that they do not have visual attributes. They can be useful in a distributed application: they give you the ability to perform database processing on a remote server instead of on each client computer.

For more information about using DataStores for database access in a transaction server environment, see “Using DataStores” on page 471.

Passing result sets

PowerBuilder provides three system objects to handle getting result sets from components running in transaction server environments and returning result sets from PowerBuilder user objects running as transaction server components. These system objects, ResultSet, ResultSets, and ADOResultSet, are designed to simplify the conversion of transaction server result sets to and from DataStore objects and do not contain any state information. They are not designed to be used for database updates. You can use ADOResultSet only with COM components.

COM+ uses ActiveX Data Objects (ADO) RecordSet objects to return result sets. An ADO Recordset object consists of records (rows) and fields (columns) and represents the set of records in a database table.

ADO Recordsets and PowerBuilder system objects

In PowerBuilder you use functions on the PowerBuilder ADOResultSet system object to get and set data that is passed in an ADO Recordset. PowerBuilder clients use OLEObjects to handle ADO Recordsets. You use the CreateFrom and GenerateResultSet functions on the DataStore object to convert the result sets stored in ResultSet objects to and from DataStore objects.

About GenerateResultSet
GenerateResultSet has an alternative syntax used for returning result sets when using MASP (Method as Stored Procedure) with EAServer.

Table 27-3 summarizes how these objects interact.

Table 27-3: Result set objects

Use a variable of this type	In this way
ResultSet	As the return value of a method (function) defined for a COM component. The data is marshaled as an ADO Recordset.
OLEObject	To hold the ADO Recordset returned from a method on a COM component that returns a ResultSet. The OLEObject can be manipulated using ADO Recordset functions such as MoveFirst.

Use a variable of this type	In this way
ADOResultSet	Use SetResultSet to populate an ADOResultSet object with data from a ResultSet object. Use SetRecordSet to populate an ADOResultSet object with data from an OLEObject that holds an ADO Recordset. Use GetRecordSet to populate an OLEObject with data from an ADOResultSet. The OLEObject can be manipulated using ADO Recordset functions such as MoveFirst.
DataStore	Use CreateFrom to create a DataStore from an object of type ResultSet or ADOResultSet. Use GenerateResultSet to generate a ResultSet object from a DataStore object in a method on a transaction server. The ResultSet object can be returned to a client.

Accessing result sets in COM components from PowerBuilder clients

When a PowerBuilder client calls a COM component method that returns an ADO Recordset, the data returned is stored in an OLEObject object. You can manipulate the data in the ADO Recordset using Recordset functions, as described in "Using ADO Recordsets in PowerBuilder" next.

To use the data stored in the OLEObject object to populate a DataStore object, create an ADOResultSet object and then call its SetRecordSet function to populate it with data stored in the OLEObject object.

The data in the ADOResultSet object can be used to populate a DataStore object using the CreateFrom DataStore function:

```
OLEObject loo_mycomponent
OLEObject loo_ADOrecordset
ADOresultset lrs_ADOresultset
datastore ds_local
integer li_rc
loo_mycomponent = CREATE OLEObject

li_rc = loo_mycomponent.ConnectToNewObject("PB.Test")
if li_rc <> 0 then
    MessageBox("Connect Failed", string(li_rc) )
    Return
end if

// Use an OLEObject to hold ADO Recordset
// returned from method on COM component
loo_ADOrecordset = loo_mycomponent.GetTestResult()

// Create an ADOResultSet and get its data
// from OLEObject holding passed ADO Recordset
lrs_ADOresultset = CREATE ADOResultSet
lrs_ADOresultset.SetRecordSet(loo_ADOrecordset)

// Use CreateFrom to populate DataStore
// from ADOResultSet object
ds_local = CREATE datastore
ds_local.createfrom(lrs_ADOresultset)
```

Using ADO Recordsets in PowerBuilder

If you want to manipulate an ADO Recordset in PowerBuilder using ADO Recordset methods such as MoveFirst or MoveNext, you can use the SetResultSet and GetRecordSet functions. Use SetResultSet to populate a new ADOResultSet object with data from a ResultSet object, then use GetRecordSet to return the ADO Recordset.

This example generates a result set in a ResultSet object from an existing DataStore object. The ResultSet object is used to populate a new ADOResultSet object. The GetRecordSet function on the ADOResultSet object is used to return an ADO Recordset as an OLEObject that can be used with ADO Recordset methods.

```
ResultSet lrs_resultset
ADOresultset lrs_ADOresultset
OLEObject loo_ADOrecordset

// Generate a result set from an existing DataStore
ds_source.GenerateResultSet(lrs_resultset)

// Create a new ADOResultSet object and populate it
// from the generated result set
lrs_ADOresultset = CREATE ADOResultSet
lrs_ADOResultset.SetResultSet(lrs_resultset)
// Pass the data in the ADOResultSet object

// to an OLEObject you can use as an ADO Recordset
loo_ADOrecordset = CREATE OLEObject
lrs_ADOResultset.GetRecordSet(loo_ADOrecordset)

// Call native ADO Recordset methods on the OLEObject
loo_ADOrecordset.MoveFirst()
```

Returning result sets from COM and COM+ components

To pass or return result sets from a PowerBuilder user object that will be deployed to COM or COM+, set the datatype of a function’s argument or return value to ResultSet. When the GenerateResultSet function is called to create a result set from a DataStore object in COM or COM+, the result set is marshaled and returned to the client as an ADO Recordset.

In this example, a DataStore object is created and data is retrieved into it, and then the GenerateResultSet function is used to create a result set that can be returned to a client:

```
datastore ds_datastoreresultset lrs_resultset
integer li_rc

ds_datastore = create datastore
ds_datastore.dataobject = "d_empdata"
ds_datastore.SetTransObject (SQLCA)
IF ds_datastore.Retrieve() = -1 THEN
    // report error and return
END IF

li_rc = ds_datastore.generateresultset(lrs_resultset)
IF li_rc <> 1 THEN
    // report error and return
END IF
return lrs_resultset
```

Providing support for transactions

If a component supports transactions, COM+ ensures that the component's database operations execute as part of a transaction.

Using the transaction service context object

PowerBuilder components running in COM+ can use a transaction context service to control transactions. A PowerBuilder COM object creates a context object that enables the component to take advantage of Microsoft DTC support. The **TransactionServer** object enables a COM object running in COM+ to access the context associated with the object, giving it tighter control of transactions and activation. It also provides some control of the security context.

For more information about the TransactionServer object and its methods, see *Objects and Controls* and the *PowerScript Reference*.

Before you can use the transaction context service, you need to declare a variable of type TransactionServer and call the GetContextService function to create an instance of the service:

```
TransactionServer txninfo_base
this.GetContextService("TransactionServer", &
    txninfo_base)
```

To access information about the component’s transaction context and control the transaction, call methods on the instance of the TransactionServer object—for example:

```
IF txninfo_base.IsInTransaction() THEN
    txninfo_base.DisableCommit()
END IF
...
txninfo_base.SetComplete()
```

When you no longer need the reference to the service, you should destroy it:

```
DESTROY txninfo_base
```

Behavior of COMMIT and ROLLBACK

When a PowerBuilder component is running in COM+, the TransactionServer interface is used to control transactions when the **UseContextObject** DBParm parameter is set to Yes. When the **UseContextObject** DBParm parameter is set to Yes, COMMIT and ROLLBACK statements result in a database error. The transaction remains active until SetComplete or SetAbort is issued using an instance of the TransactionServer context object.

Migrating PowerBuilder 6 components
Components built with PowerBuilder 6 relied on the PowerBuilder database driver to issue SetComplete or SetAbort calls to COM+. If you are migrating these components to the current version of PowerBuilder, modify your code to use the TransactionServer interface. Alternatively, you can set the UseContextObject DBParm to No. In this case, COMMIT is equivalent to SetComplete and ROLLBACK is equivalent to SetAbort. This approach is recommended only when you want to migrate PowerBuilder 6 objects without modifying the code.

Specifying whether a component supports transactions

Each COM+ component has a transaction property that indicates how the component participates in transactions. PowerBuilder COM objects create a new context object when the component transaction property in COM+ is set to *Requires a new transaction*. A PowerBuilder COM object whose component transaction property is set to *Requires a transaction* or *Supports transactions* either inherits a transaction from an existing object or creates a new transaction.

You set this property in the COM/COM+ Project wizard or the Project painter. Table 27-4 describes the values you can set.

Table 27-4: Component transaction property values

Transaction type	Description
Not supported	The component never executes as part of a transaction. If the component is activated by another component that is executing within a transaction, the new instance's work is performed outside the existing transaction.
Supports Transaction	The component can execute in the context of a COM+ transaction, but a connection is not required to execute the component's methods. If the component is instantiated directly by a client, COM+ does not begin a transaction. If component A is instantiated by component B and component B is executing within a transaction, component A executes in the same transaction.
Requires Transaction	The component always executes in a transaction. When the component is instantiated directly by a client, a new transaction begins. If component A is activated by component B, and B is executing within a transaction, A executes within the same transaction; if B is not executing in a transaction, A executes in a new transaction.
Requires New Transaction	Whenever the component is instantiated, a new transaction begins. If component A is activated by component B, and B is executing within a transaction, then A begins a new transaction that is unaffected by the outcome of B's transaction; if B is not executing in a transaction, A executes in a new transaction.

For more information about how transactions work, see the Microsoft COM+ documentation in the MSDN Library at http://msdn.microsoft.com/library/default.asp.

Invoking another server component's methods

COM+ allows the methods of one server component to call methods of another. The other server component does not need to be another PowerBuilder component; it can be implemented in any language supported by COM+.

Using an OLEObject object

To access methods of another component, declare a variable of type OLEObject and call the ConnectToNewObject function to connect to the component, just as you would when calling the component from a client. ConnectToNewObject inherits the server object's transaction context automatically.

Using a TransactionServer object

To access methods of another component in the current server, you can also use the transaction service context object that PowerBuilder provides called TransactionServer. The TransactionServer interface provides a method called CreateInstance that allows you to access other components that are available locally. CreateInstance uses the same user and password information that applies to the component from which it is called.

Security issues

When you develop a component for deployment to COM+, you can define roles that determine which users or groups of users are authorized to perform specific transactions. Then when you deploy the component, you assign roles to specific users in the COM+ Component Services tool.

Enabling authorization in the Project painter or wizard

When you create a COM/COM+ project using a wizard, you can instruct COM+ to check the security credentials of any client that calls the component. In the Project painter, you can specify checking at both the component and package level on the COM+ Component and COM+ Package property pages.

To ensure that security is enabled, add a role to the COM application in the Microsoft Management Console, add users to the role, and grant the role to the component.

Programmatic security

PowerBuilder provides functions on the transaction service object that you can use in the component to determine programmatically whether the caller is authorized to call a specific method. IsSecurityEnabled determines whether security is enabled for the component. IsCallerInRole determines whether the client process or server process calling a method on the component is in a role that is authorized to call it.

Impersonation

IsCallerInRole looks at the role of the direct caller of the current method. If a client calls a method on a component, and that method accesses a database, the access rights to the database are determined by the security context of the component, not the client. PowerBuilder provides additional functions on the transaction service object to enable the component to assume the security context of the client before performing an operation that the client may not be authorized to perform. ImpersonateClient assumes the security context of the client, IsImpersonating determines whether the component is running in its client's security context, and RevertToSelf restores the component's security context.

Building COM/COM+ components in the Project painter

You build PowerBuilder COM servers in the Project painter.

If you did not create a project when you created the user object, you can use the COM/COM+ Project wizard to create one. You can also set up a project to build a COM component directly in the Project painter by selecting the COM/COM+ Component icon on the Projects tab.

If you have already created a COM/COM+ project for one or more of the objects you want to build into a COM server, you can modify it in the Project painter, adding additional objects if necessary.

The Project painter workspace is a read-only display that shows the options you selected in the wizard or in the Select Objects and Properties dialog boxes in the painter. When you build a PowerBuilder COM server, the workspace also displays an object inspection report and the status of each phase of the build process.

❖ **To define and build a PowerBuilder COM server project:**

1. Select COM/COM+ Component Wizard from the Projects tab in the New dialog box.
2. Specify project properties including the project’s name and location.
3. Select one or more objects that you want to build into a server.
4. Specify the properties of each object and optional COM+ deployment options and click Finish.

   For help specifying properties, see the context-sensitive Help in the wizard or the following topics:

Property	For information see
Interface options	“Choosing a custom or dual interface” on page 574 and “Instance variables” on page 561
Build options	“Setting up the embedded PBD” on page 574
Registering components	“Registering components automatically” on page 573
COM+ deployment and package options	“Deploying components to COM+” on page 574
COM+ transaction settings	“Specifying whether a component supports transactions” on page 569
COM+ security	“Security issues” on page 571

5 Select File>Open and select the project you just created to open the Project painter.

You can verify that the correct objects are selected by selecting Edit>Select Objects from the menu bar.

6 Select Edit>Properties to verify and optionally modify the properties you set in the wizard.

Some advanced COM+ package properties can be set only in the Project painter.

7 Click the Build button in the Project painter to build the PowerBuilder COM server.

The build process creates an IDL file and a PowerBuilder COM server (DLL) containing a PowerBuilder COM object for each user object you selected. If you specified deployment to COM+ and it is installed and running on the computer where you are generating the component, the component is deployed directly to the server, and additional deployment files can be created.

The PowerBuilder COM server also contains an embedded PBD file that contains the custom class user objects and any additional objects they reference, and an embedded type library.

Building a project after calling an object
When you call a COM object in the development environment, the COM object stays in memory until you exit PowerBuilder, because it is hosted in the same process as PowerBuilder. If you make a change to the object after calling it and try to regenerate it in the Project painter, you receive compile and link errors. Before you build the project, shut down and restart PowerBuilder to free the object from memory.

Registering components automatically

In the Project wizard or Project painter, you can choose to register all generated PowerBuilder COM objects automatically when a build is successful. This makes it easy to test the component on your computer. To avoid making unnecessary registry entries on your computer, select this option only when you are ready to test the PowerBuilder COM server.

Deploying components to COM+

You can deploy components to COM+ automatically when a build is successful. You can also export the application to a Microsoft Windows Installer (MSI) file that you can use to import the package into a COM+ server on another computer, and export an application proxy as an MSI file that can be installed on a client computer so that it can access the COM+ server remotely. For more information about deploying COM+ components, see the Microsoft COM+ documentation in the MSDN Library at http://msdn.microsoft.com/library/default.asp.

Choosing a custom or dual interface

When you generate a PowerBuilder COM object, you must choose whether to expose a custom interface or a dual interface to clients. In PowerBuilder COM objects, both interface types are currently restricted to the use of standard OLE automation datatypes.

Custom interfaces

Custom interfaces provide access to server object virtual function tables (VTBL), offering higher performance than dispatch-based interfaces and a cleaner usage model than dual interfaces. Consider using a custom interface if the intended client is in a compiled language such as C++ or if you want to take advantage of the support for custom interfaces in a tool such as Visual Basic.

PowerBuilder COM objects that use custom interfaces use the standard marshaling provided by COM.

Dual interfaces

Dual interfaces allow programmers to call methods in the COM object through virtual function tables as well as through the dispatch interface. They support a wide range of clients and give fast access to methods.

Setting up the embedded PBD

The COM server contains an embedded PowerBuilder dynamic library (PBD) file that contains compiled versions of all the custom class user objects you selected plus any dependent objects. You can include additional, unreferenced objects in the PBD by selecting the library that contains the objects on the Library properties page. All the objects in selected libraries are included in the PBD.

You can specify the name of a PowerBuilder Resource (PBR) file associated with the library. A PBR file is a text file that lists dynamically assigned resources such as bitmaps. For more information, see “About resources” on page 754.

How the PowerBuilder COM object executes

To establish an instance of a PowerBuilder COM object, the PowerBuilder virtual machine (*PBVM110.DLL*) loads the custom class object from the PBD embedded in the COM server DLL and creates the proper session and instance information. When a client invokes a PowerBuilder COM object method, the method calls back into the PowerBuilder virtual machine to execute the appropriate PowerScript code.

PowerBuilder COM servers use the COM single-threaded apartment (STA) model. Runtime sessions can be shared by objects created on the same thread. When a client requests an instance of a PowerBuilder COM object, the PowerBuilder COM server establishes a runtime session and creates an instance of the custom class user object within that runtime session. When the server receives a new request for an instance of a COM object from the same thread, it instantiates the object using the existing runtime session.

In all cases, separate clients use distinct PowerBuilder virtual machine sessions.

Memory allocation

The first time any client requests an instance of a PowerBuilder COM object from a PowerBuilder COM server, the server loads the PowerBuilder virtual machine, allocates a block of memory, and starts a runtime session. About 4MB of memory is allocated for the PowerBuilder virtual machine, cached memory, and the runtime session.

Subsequent requests do not require any additional memory allocation if the objects can share the same runtime session.

If a subsequent request requires a PowerBuilder COM object to be created in a different COM apartment (usually because the request was made from a different client or thread), the object is instantiated in a new runtime session. Each new session requires only about 200K of memory; it shares the instance of the PowerBuilder virtual machine and the cached memory.

Deploying a PowerBuilder COM server

Once you have generated a PowerBuilder COM server, you can use COM-enabled client applications to create the PowerBuilder COM objects and access their methods. You can use the server with any COM-enabled application or deploy it as a package to COM+.

Using a PowerBuilder COM server with a COM-enabled application

❖ **To use a PowerBuilder COM server with a COM-enabled application:**

1. Deploy the PowerBuilder COM server to the user’s computer.
2. Deploy the PowerBuilder virtual machine (*PBVM110.DLL*) and any other required modules (such as *PBCOMRT110.DLL*, *PBDWE110.DLL*, or any required database software) to the computer containing the PowerBuilder COM server.
3. Register the PowerBuilder COM server on the user’s computer.
4. Write a client application that calls PB object functions in the PowerBuilder COM server.

   See “Accessing PowerBuilder COM servers from clients” on page 577.

Registering PowerBuilder COM servers When you deploy a PowerBuilder COM server, you need to add information to the registry to enable COM to create instances of the server’s objects. PowerBuilder COM servers are self-registering—you do not need to create a separate registration file. To register a PowerBuilder COM server, use the REGSVR32 utility. For example:

```
regsvr32.exe path_to_server\mycomserver.dll
```

If you want to access the server from a remote client, you need to make an additional change to the registry and you may need to configure the client for remote access to the server. For more information, see "Using PowerBuilder COM servers and objects with DCOM" on page 581.

Registering automatically for testing
For testing purposes, you can check the box on the General property page that registers the server on the development computer automatically when it has been generated. To avoid making unnecessary registry entries on your computer, check this box only when you are ready to test the PowerBuilder COM server.

Accessing PowerBuilder COM servers from clients

You can access the methods on a PowerBuilder COM component from clients built with any COM-compliant tool. For COM+, the client must have Microsoft Windows Installer. The COM server must be registered on the client machine, or the COM+ application proxy file must be installed.

For how to access PB COM servers from a remote client, see "Using PowerBuilder COM servers and objects with DCOM" on page 581.

The following examples show how you access a PowerBuilder COM object from Visual Basic or C++. They use a PowerBuilder COM object that was generated from a user object called ccuo_employee and has the Program ID PB110.employee.

For information about building PowerBuilder clients and an example using the same COM object, see Chapter 28, "Building a COM or COM+ Client."

Visual Basic as client

In Visual Basic, you can connect to the registered object using its program ID (late binding). In Visual Basic 5 or later, you can also use its class name (early binding).

❖ **To access a PowerBuilder COM object in Visual Basic:**

1 Do one of the following:

- Declare an object and connect to it using its program ID:

  ```
  Dim EmpObj As Object
  Set EmpObj = CreateObject("PB110.employee")
  ```

- Add a reference to the generated type library for the PowerBuilder COM object to your project, then declare an instance of the object using its class name (in Visual Basic 5 or later):

  ```
  Dim EmpObj As New CoEmployee
  ```

2 Check that the connection was established:

```
Dim response
If EmpObj Is Nothing Then
response = MsgBox("Creating Employee Object",
      vbOKOnly, "Error")
End If
```

3 Access functions or properties of the object:

```
Dim units, time as Long
Dim DoubleReturn as Double
Dim StringReturn As String

DoubleReturn = EmpObj.f_calcdayavg units, time
StringReturn = EmpObj.f_teststring
EmpObj.ll_hours = 37
```

4 Destroy the object:

```
Set EmpObj = Nothing
```

C++ as client

In C++, you use COM library functions to create an instance of a PowerBuilder COM object. You also need to use C/C++ definitions of the PowerBuilder COM objects when you build the client. The Microsoft IDL (MIDL) compiler generates these definitions from the IDL file created by the PowerBuilder COM generator.

For example, using the IDL file generated for the Employee PowerBuilder COM object:

```
midl.exe employee.idl
```

The MIDL compiler generates a header file (*employee.h*) containing definitions of the interfaces of all the objects in the IDL file and a C file (*employee_i.c*) that defines the CLSIDs and Interface IDs (IIDs) associated with the object.

Additional files
The MIDL compiler also generates proxy/stub code (in *employee_p.c* and *dlldata.c*), but you do not need to use the proxy/stub code to build the C++ client executable or access the PowerBuilder COM object.

Building a client To build a C++ client executable that can access methods in a PowerBuilder COM object, you create a C++ source file that includes the generated header file, compile both the C++ client source file and the C file generated by the MIDL compiler, then link the resulting object files to create the executable.

For the Employee example:

1 Create a C++ source file called *client.cpp* (shown below).

2 Compile *client.cpp*.

3 Compile *employee_i.c*.

4 Link *client.obj* and *employee_i.obj* to create an executable—for example, *employee_ecl.exe*.

Employee.h The following code fragments from the *employee.h* header file generated by the MIDL compiler show the definitions to be used by C++ clients:

```
typedef interface DIEmployee DIEmployee;
EXTERN_C const IID IID_DIEmployee;

interface DECLSPEC_UUID("A2F59F71-D5FB-11D1-92B9-
00A0247712F1")
    DIEmployee : public IDispatch
    {
    public:
    virtual /* [id] */ HRESULT STDMETHODCALLTYPE
       f_calcdayavg(
      /* [in] */ long units,
      /* [in] */ long time,
      /* [retval][out] */ double __RPC_FAR
        *retval) = 0;

    virtual /* [id] */ HRESULT
```

```
        STDMETHODCALLTYPE f_teststring(
        /* [retval][out] */ BSTR __RPC_FAR
           *retval) = 0;
    };

EXTERN_C const CLSID CLSID_CoEmployee;
```

Client.cpp The following sample client file uses the MIDL-generated C/C++ definitions of PowerBuilder COM objects. For further information on the COM API calls shown in *client.cpp*, see the Microsoft Software Development Kit documentation.

```
#include <windows.h>
// employee.h I( from MIDL.EXE)
#include "employee.h"

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE,
LPSTR, int)
{
    HRESULT hr;
    DIEmployee      *pDIEmployee = 0;

    //  Initialize COM
    CoInitialize(0);

    hr = CoCreateInstance(CLSID_CoEmployee,NULL,
       CLSCTX_INPROC_SERVER, IID_DIEmployee,
      (void **)&pDIEmployee);
    if (FAILED(hr))
    ErrorMessage("CoCreateInstance", hr);

    // variables for methods
    long units, time;
    double dReturn;
    BSTR  strReturn = NULL;

    // call methods
    hr = pDIEmployee->f_calcdayavg(units,
                   time,&dReturn);
    if (FAILED(hr))
    ErrorMessage("f_calcdayavg",hr);

    hr = pDIEmployee->f_teststring(&strReturn);
    if (FAILED(hr))
    ErrorMessage("f_teststring",hr);

    // release the interface ptr
```

```
        pDIEmployee->Release();

        CoFreeUnusedLibraries();
        //  all done!

    CoUninitialize();
        return 0;
    }
```

Using PowerBuilder COM servers and objects with DCOM

A PowerBuilder COM object can be activated from remote clients using DCOM. The object must be activated in a server process on the designated host computer. Out-of-process servers (EXE files) create a server process, but in-process servers (DLL files) must be hosted in a surrogate process.

COM provides a general-purpose surrogate host (*DLLHOST.EXE*) that can be used to host PowerBuilder COM server DLLs. Marking PowerBuilder COM servers to use a surrogate host is the primary step in enabling remote client access to your PowerBuilder COM objects. You can use the DCOM configuration utility (*DCOMCNFG.EXE*) to change values for location, security, and identity, but in most cases the defaults are adequate. For more information, see the online Help for the DCOMCNFG utility.

Enabling PowerBuilder COM servers to use a surrogate host

There are two ways to enable PowerBuilder COM servers to use a surrogate:

- Use the registry editor (*REGEDIT32.EXE*) to edit the PowerBuilder COM server's AppID registry entry.
- Use the OLE/COM Object Viewer (*OLEVIEW.EXE*) provided with Microsoft Visual C++ 5.0 or greater.

Using OLEVIEW is the preferred approach, because manually editing your computer's registry may render all or parts of your computer's configuration inoperable.

❖ **To enable a COM server to use a surrogate process using the registry editor:**

1. Open the project used to generate the server and copy the PowerBuilder COM server's AppID value from the General property page.
2. Run *REGEDIT.EXE*, find the server's AppID key in *My Computer\HKEY_CLASSES_ROOT\AppID*, and select it.
3. Select Edit>New>String Value from the menu bar.

4 Enter the name `DllSurrogate` and leave the data field empty.

An empty data field tells COM to use the default surrogate host (*DLLHOST.EXE*). The AppID value keys should look like this:

Name	Data
(Default)	PowerBuilder 11.0 generated server: *servername*.dll
DllSurrogate	""

❖ **To enable a COM server to use a surrogate process using the OLE/COM Object Viewer:**

1 Run *OLEVIEW.EXE*.

2 Expand the Automation Objects in the list view and select an object in your PowerBuilder COM server.

3 Select an object associated with your PowerBuilder COM server.

4 Select the Implementation tab.

5 Select the Inproc Server tab and check the Use Surrogate Process check box.

Configuring client computers to activate remote PowerBuilder COM objects

To activate a remote component, a client application must pass the object's class identifier (CLSID) in the activation request to the remote host.

Some clients, such as those built with PowerBuilder 11.0 or C++, can use the object's CLSID in the method call when they create an instance of a remote object. These client applications do not require any client-side configuration.

Most clients reference an object by its name (ProgID) rather than its CLSID. In these cases the client computer's registry must contain the information necessary to map a ProgID to a CLSID in order to make the remote activation request. You can use either of two methods to add the required registry information to the client computer:

- Register the PowerBuilder COM server on each client computer that requires remote access.

  To use this method, you must be able to locate the appropriate version of *PBVMn0.DLL*.

- On the host where the PowerBuilder COM server is registered, export the required registry information into .REG files using *REGEDIT.EXE*, then copy and import these files into the registry of each client computer that requires remote access.

For each PowerBuilder COM object, export the following registry keys:

```
HKEY_CLASSES_ROOT\PB110.objectname
HKEY_CLASSES_ROOT\PB110.objectname.version_number
HKEY_CLASSES_ROOT\CLSID\{objects_clsid}
```

You may also need the following registry keys for the PowerBuilder COM server:

```
HKEY_CLASSES_ROOT\TypeLib\{comserver_typelib_id}
HKEY_CLASSES_ROOT\AppID\{comserver_application_id}
```

Connecting to remote objects using PowerBuilder

PowerBuilder clients can use the ConnectToNewRemoteObject function to activate remote objects, as shown in this code fragment:

```
OLEObject remobj
remobj = CREATE OLEObject
remobj.ConnectToNewRemoteObject("myremotehostname",  &
    "PB110.employee")
```

You can also use the remote object's CLSID string in the classname parameter:

```
remobj.ConnectToNewRemoteObject("myremotehostname",  &
    "clsid:0EA53FED-646A-11D2-BF8E-00C04F795006")
```

The use of the object's CLSID as the classname parameter eliminates the need for any client-side configuration.

Connecting to remote objects using C++

C++ clients that use header files created from the generated PowerBuilder COM server IDL file can use the remote object's CLSID in the activation request:

```
COSERVERINFO  ServerInfo;
MULTI_QI      mqi[1];
OLECHAR       wszHostName[MAXFILE];
LPTSTR        pszHost=NULL;

memset(&ServerInfo,0,sizeof(ServerInfo));
pszHost =GetUserRequestedHostName();
mbstowcs(wszHostName,pszHost,MAXFILE);

ServerInfo.pwszName = wszHostName;
mqi[0].pIID = &IID_Iemployee;
mqi[0].pItf = NULL;
mqi[0].hr   = S_OK;

// Create employee object on the desired server
hr = CoCreateInstanceEx(CLSID_Employee,NULL,
       CLSCTX_REMOTE_SERVER,&ServerInfo,1,mqi);
```

CHAPTER 28 Building a COM or COM+ Client

About this chapter

This chapter explains how to build a PowerBuilder client that accesses a COM or COM+ server component.

Contents

About building a COM or COM+ client

A PowerBuilder application can act as a client to a COM server. The server can be built using PowerBuilder or any other COM-compliant application development tool and it can run locally, on a remote computer as an in-process server, or in COM+.

You can use the Template Application start wizard to help you build COM and COM+ clients.

Configuring a client computer to access a remote component

When a COM component is running on a remote computer, the client computer needs to be able to access its methods transparently. To do this, the client needs a local proxy DLL for the server and it needs registry entries that identify the remote server. The client computer must be running Windows 2000 or Windows XP.

If the component is installed in COM+, the COM+ Component Services tool can create a Microsoft Windows Installer (MSI) file that installs an application proxy on the client computer.

If the server is not installed in COM+, the client and proxy files must be copied to the client and the server must be configured to run in a surrogate process. For more information, see “Using PowerBuilder COM servers and objects with DCOM” on page 581.

Remote server name written to registry
If the COM server is moved to a different computer, the registry entries on the client must be updated.

Connecting to a COM server

To access a method associated with a component in the COM server, the PowerBuilder client connects to the component using its programmatic identifier (ProgID) or its class identifier (CLSID).

You can use a tool such as OLEVIEW or the OLE tab in the PowerBuilder Browser to view the Program ID or CLSID and methods of registered PowerBuilder COM objects.

To establish a connection to the COM server, you need to execute the PowerScript statements required to perform these operations:

1. Declare a variable of type OLEObject and use the Create statement to instantiate it.
2. Connect to the object using its Program ID or CLSID.
3. Check that the connection was established.

Example The following script instantiates the EmpObj OLEObject object, connects to the PowerBuilder COM object PBcom.Employee, and checks for errors:

```
OLEObject EmpObj
Integer li_rc
EmpObj = CREATE OLEObject
li_rc = EmpObj.ConnectToNewObject("PBcom.employee")
IF li_rc < 0 THEN
    DESTROY EmpObj
MessageBox("Connecting to COM Object Failed",  &
   "Error: " + String(li_rc))
Return
END IF
```

Interacting with the COM component

Invoking component methods

Once a connection to a COM component has been established, the client application can begin using the component methods.

Use the REF keyword for output parameters
You must use the REF keyword when you call a method on a COM object that has an output parameter. For example: of_add(*arg1*, *arg2*, REF *sum*)

Example Using the EmpObj object created in the previous example, this example calls two methods on the component, then disconnects and destroys the instance:

```
Long units, time
Double  avg, ld_retn
String  ls_retn

ld_retn = EmpObj.f_calcdayavg(units, time, REF avg)
ls_retn = EmpObj.f_teststring()

EmpObj.DisconnectObject()
DESTROY EmpObj
```

Passing result sets

PowerBuilder provides three system objects to handle getting result sets from components running in transaction server environments and returning result sets from PowerBuilder user objects running as transaction server components. These system objects (ResultSet, ResultSets, and ADOResultSet) are designed to simplify the conversion of transaction server result sets to and from DataStore objects and do not contain any state information.

For more information, see “Passing result sets” on page 565.

Handling runtime errors

Runtime error information from custom class user objects executing as OLE automation objects, COM objects, or COM+ components is reported to the container holding the object as exceptions (or, for automation objects, as exceptions or facility errors). Calls to the PowerBuilder SignalError function are also reported to the container. To handle runtime errors generated by PowerBuilder objects, code the ExternalException event of the OLE client.

For more information about handling runtime errors in OLE or COM objects, see “Handling errors” on page 354.

Controlling transactions from a client

PowerBuilder clients can exercise explicit control of a transaction on a COM+ server by using a variable of type OleTxnObject instead of OLEObject to connect to the COM object.

Requires COM+ installation
The ConnectToNewObject call on an OleTxnObject fails if COM+ is not installed on the client computer.

The OleTxnObject object, derived from the OLEObject object, provides two additional functions (SetComplete and SetAbort) that enable the client to participate in transaction control. When the client calls SetComplete, the transaction is committed if no other participant in the transaction has called SetAbort or otherwise failed. If the client calls SetAbort, the transaction is always aborted.

Example In this example, the clicked event on a button creates a variable of type OleTxnObject, connects to a PowerBuilder COM object on a server, and calls some methods on the object. When all the methods have returned, the client calls SetComplete and disconnects from the object.

```
integer li_rc
OleTxnObject lotxn_obj

lotxn_obj = CREATE OleTxnObject
li_rc = lotxn_obj.ConnectToNewObject("pbcom.n_test")
IF li_rc <> 0 THEN
    Messagebox( "Connect Error", string(li_rc) )
    HALT
END IF

lotxn_obj.f_dowork()
lotxn_obj.f_domorework()

lotxn_obj.SetComplete()
lotxn_obj.DisconnectObject()
```

This f_dowork function on the PowerBuilder COM object on the server creates an instance of the transaction context service and calls its DisableCommit method to prevent the transaction from committing prematurely between method calls. After completing some work, the function calls SetAbort if the work was not successfully completed and SetComplete if it was.

```
TransactionServer txninfo_one
integer li_rc
```

```
li_rc = GetContextService( "TransactionServer",  &
    txninfo_one )
txninfo_one.DisableCommit()

// do some work and return a return code
IF li_rc <> 0 THEN
    txninfo_one.SetAbort()
    return -1
ELSE
    txninfo_one.SetComplete()
    return 1
END IF
```

The SetComplete call on the client commits the transaction if *all* of the methods in the transaction called SetComplete or EnableCommit.

CHAPTER 29 Building an EJB client

About this chapter

This chapter describes how to build a PowerBuilder client for an Enterprise JavaBeans component running on a J2EE-compliant application server. Reference information for the objects described in this chapter is in the *PowerBuilder Extension Reference* and in the online Help.

Contents

About building an EJB client

A PowerBuilder application can act as a client to an EJB 1.1 or 2.0 component running on an application server that is J2EE compliant. This capability relies on PowerBuilder extension files provided by Sybase.

PowerBuilder extension files are developed using the PowerBuilder Native Interface (PBNI). You do not need to know anything about PBNI to create EJB clients, but you can read more about PowerBuilder extensions in the *PowerBuilder Extension Reference,* and about PBNI in the *PowerBuilder Native Interface Programmer's Guide and Reference.*

EJB clients for EAServer

If you are developing EJB clients for EAServer, you can use the techniques described in this chapter, or you can create a client that uses the PowerBuilder Connection object and EAServer proxy objects.

The EJB client extension is a wrapper for Java, and therefore provides more flexibility in communicating with EJBs. For example, an EJB client can manipulate a Java class returned from an EJB method call through its proxy.

The PowerBuilder Connection object has a smaller footprint (and thus is easier to deploy) because it does not require a JRE to be installed on the computer where the client resides. Connectivity to the server is also faster with the connection object, because there is no delay while a JRE loads.

For more information about building an EJB client for an EJB component running on EAServer using the PowerBuilder connection object, see Chapter 25, "Building an EAServer Client."

pbejbclient110.pbx and pbejbclient110.pbd

To connect to the server and communicate with the EJB component, clients use a set of classes implemented in a DLL file with the suffix PBX, *pbejbclient110.pbx*. To use the classes in this PBX file, you must import the definitions in it into a library in the client application. You can also add the *pbejbclient110.pbd* file, which acts as a wrapper for the PBX file, to the target's library search path.

About EJB proxy objects

The PowerBuilder client uses local proxy objects for the EJB component to delegate calls to methods on the remote EJB component. At a minimum, each EJB component is represented in the client application by a proxy for the home interface and a proxy for the remote interface. For example, an EJB component named Cart has two proxies, CartHome and Cart, each containing only the signatures of the public methods of those interfaces.

Additional proxies are also generated for exceptions and ancillary classes used by the home and remote interfaces. For more information, see "Generating EJB proxy objects" on page 594.

Overview of the process

To build an EJB client, you need to complete the following steps:

1 Create a workspace and a PowerScript target.

2 Add *pbejbclient110.pbx* to the application.

3 Create a project for building proxy objects.

4 Build the project to generate the proxy objects.

5 Create the windows required to implement the user interface of the client application.

6 Instantiate a Java VM.

7 Establish a connection to the server and look up the EJB.

8 Create an instance of the EJB component and call component methods from the client.

9 Test and debug the client.

Adding pbejbclient110.pbx to your application

The simplest way to add the PBEJBClient classes to a PowerBuilder target is to import the object descriptions in the *pbejbclient110.pbx* PBX file into a library in the PowerBuilder System Tree

The *pbejbclient110.pbx* and *pbejbclient110.pbd* files are installed in the *Shared/PowerBuilder* directory when you install PowerBuilder. When you create an EJB client application, you do not need to copy *pbejbclient110.pbx* to another location, but you do need to deploy it with the client executable in a directory in the application's search path.

❖ **To import the descriptions in an extension into a library:**

1 In the System Tree, expand the target in which you want to use the extension, right-click a library, and select Import PB Extension from the pop-up menu.

2 Navigate to the location of the PBX file and click Open.

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

After you import *pbejbclient110.pbx*, the following objects display in the System Tree:

Object	Description
EJBConnection	Used to connect to an EJB server and locate an EJB.
EJBTransaction	Maps to the javax.transaction.UserTransaction interface. Used to control transactions from the EJB client.
JavaVM	Used to create an instance of the Java VM.

Generating EJB proxy objects

To generate EJB proxy objects, you need to create an EJB Client Proxy project. You can do this in the Project painter or with a wizard.

Using an EJB Proxy project

To create a new EJB Client Proxy project, select either of the following from the Projects page of the New dialog box:

- EJB Client Proxy icon
- EJB Client Proxy Wizard icon

EJB Client Proxy icon

The EJB Client Proxy icon opens the Project painter for EJB proxies so you can create a project, specify options, and build the proxy library.

❖ **To create an EJB Client Proxy project in the Project painter:**

1 Double-click the EJB Client Proxy icon on the Projects page of the New dialog box.

2 To specify the EJB, select Edit>Select Objects and enter the fully qualified name of the component’s remote interface in the text box, for example com.sybase.jaguar.sample.svu.SVULogin or portfolio.MarketMaker.

3 Enter the path of the directory or JAR file that contains the EJB’s stubs in the Classpath box and click OK.

If the stub files are in a directory and the fully qualified name of the EJB is *packagename.beanname*, enter the directory that contains *packagename*.

4 To specify the PBL where the proxy objects should be stored, select Edit>Properties and browse to the location of a library in the target’s library list.

You can specify an optional prefix that is added to the beginning of each generated proxy name. Adding a prefix makes it easier to identify the proxies associated with a specific EJB and can be used to avoid conflicts between class names and PowerBuilder reserved words. The prefix is not added to the name of proxies that are not specific to this EJB, such as the proxies for exceptions, stream objects, and ejbhome, ejbobject, ejbmetadata, handle, and homehandle.

5 Close the dialog box and select File>Save to save the project.

The new project lists the EJB component for which a proxy will be generated and specifies the name of the output library that will contain the generated proxy objects.

EJB Client Proxy Wizard icon

The EJB Client Proxy Wizard helps you create the project.

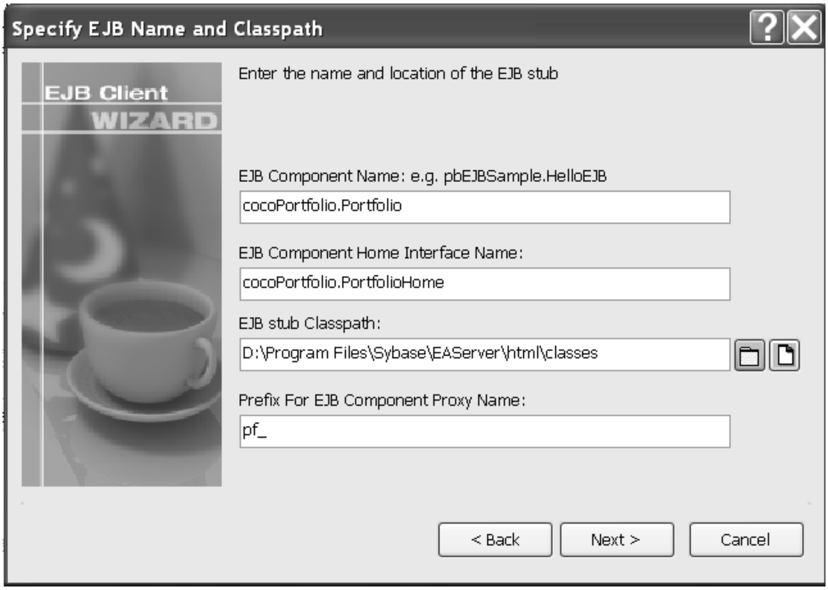
❖ **To create an EJB Client Proxy project using the wizard:**

1 Double-click the EJB Client Proxy Wizard icon on the Projects page of the New dialog box and click Next on the first page of the wizard.

2 Select a library in which to store the project object and click Next.

3 Specify a name and optional description for the project and click Next.

4 As shown, enter the fully qualified name of the component's remote interface in the text box, for example `cocoPortfolio.Portfolio`:



Specify EJB Name and Classpath

Enter the name and location of the EJB stub

EJB Component Name: e.g. pbEJBSample.HelloEJB: cocoPortfolio.Portfolio

EJB Component Home Interface Name: cocoPortfolio.PortfolioHome

EJB stub Classpath: D:\Program Files\Sybase\EAServer\html\classes

Prefix For EJB Component Proxy Name: pf_

The component's home interface name is entered automatically using the standard naming convention, although the wizard lets you modify this name if necessary.

5 Browse to select the JAR file that contains the EJB's stubs or the directory that contains the stub package.

   If the stub files are in a directory and the fully qualified name of the EJB is *packagename.beanname*, enter the directory that contains *packagename*.

6 Specify an optional prefix that is added to the beginning of each generated proxy name and click Next.

   Adding a prefix makes it easier to identify the proxies associated with a specific EJB and can be used to avoid conflicts between class names and PowerBuilder reserved words. The prefix is not added to the name of proxies that are not specific to this EJB, such as the proxies for exceptions, supporting classes, and EJBHome, EJBObject, EJBMetaData, Handle, and HomeHandle.

7 Browse to select an existing library and click Next and Finish.

   The proxy objects are generated and stored in this library, which must be added to the target's library list.

After the wizard has created the project, you can use the Project painter to modify your project settings.

Building proxies

Whether you create the EJB Proxy project using the wizard or the painter, the final step is to build the proxy objects. To do so, click the Build icon on the painter bar or select Design>Deploy Project from the menu bar.

Proxy generation requires javap.exe

PowerBuilder uses the *javap.exe* utility to generate proxy objects. This executable must be in your system path. By default, EJB client development uses the Sun JDK 1.4 installed with PowerBuilder. The path and classpath required by the Java VM are added to the path and classpath used in the current session automatically.

If you want to use a different JDK installation, select Tools>System Options, then click Set JDK Location on the Java page of the System Options dialog box. For WebSphere, the path to the IBM JDK installation can be used instead.

In addition to the proxies for the home and remote interfaces of the EJB, proxies are also generated for any Java classes referenced by the EJB, for ancestor classes, for any exceptions that can be thrown by the EJB and its supporting classes, and for the following interfaces:

Object	Description
EJBHome	Proxy for the javax.ejb.EJBHome interface, the base class for all EJB home interfaces.
EJBMetaData	Proxy for the javax.ejb.EJBMetaData interface. Allows a client to obtain the EJB's home interface and the class objects for its home and remote interfaces and primary key class (for entity beans), and to determine whether the bean is a session or stateless session object.
EJBObject	Proxy for the javax.ejb.EJBObject interface, the base class for all EJB remote interfaces.
Handle	Proxy for the javax.ejb.Handle interface. Used to provide a robust persistent reference to an EJB.
HomeHandle	Proxy for the javax.ejb.HomeHandle interface. Used to provide a robust persistent reference to a home object.

For more information about these interfaces, see the documentation for the javax.ejb package at http://java.sun.com/j2ee/sdk_1.3/techdocs/api/index.html.

The project also generates a structure that stores the mapping of Java classes to proxy names. This structure is used internally and should not be modified.

Using the *ejb2pb110* tool

You can also use the ejb2pb110 command-line tool to generate proxies. The tool generates:

- Proxies (*.srx* files) for the home and remote interfaces of the EJB you specify and for the classes on which the EJB depends.
- A PowerBuilder structure object named *ejbname_ejb_pb_mapping.srs*, where *ejbname* is the name of the EJB. This structure hosts the mapping table between the Java class name and the PowerBuilder proxy name.
- A text file called *ejbproxies.txt* or, if errors occur, *ejbproxies.err.*

These files are generated in the directory in which you invoke the command. The syntax is:

```
ejb2pb110 [ -classpath pathlist ] EJBName [EJBHomeName][ prefix ]
```

If the *pathlist* argument contains spaces, for example *D:\Program Files*, the *pathlist* must be enclosed in quotes. *EJBName* is the fully qualified remote interface class name. If you use the standard naming convention for the home interface, then including an argument for the fully qualified home interface name, *EJBHomeName*, is optional. If you specify the optional *prefix*, it is added to the beginning of the generated proxy name.

For example, the following statements generate proxies for the Portfolio class in the package cocoPortfolio on EAServer. The proxies for the home and remote interfaces of the Portfolio class have the prefix *pf_*, and the generated files are written to the directory *D:\work\proxies*:

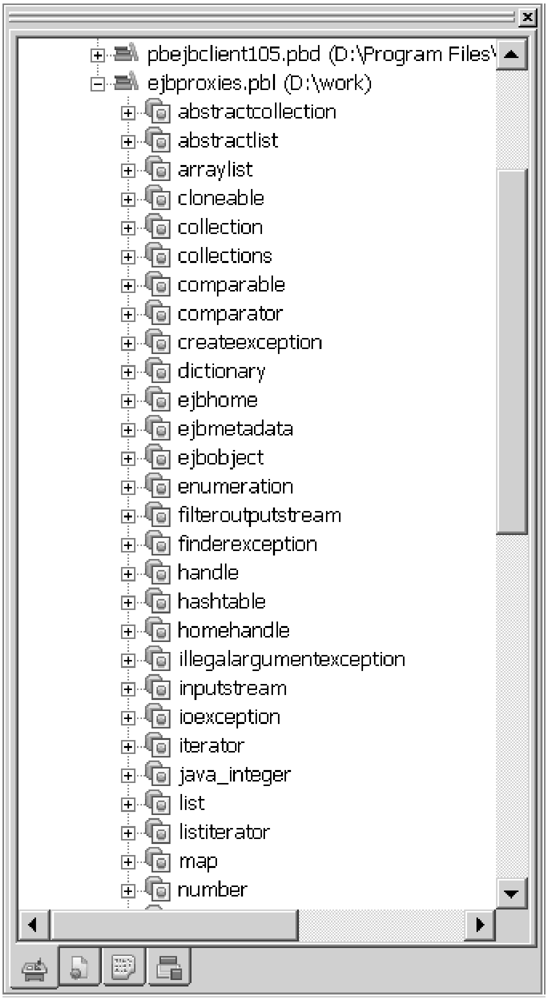
```
cd D:\work\proxies
ejb2pb110 -classpath "D:\Program
Files\Sybase\EAServer\html\classes"
cocoPortfolio.Portfolio pf_
```

The home and remote classes for the EJB and any dependent classes must be in the class path that you specify.

After generating the proxies, you import them into your target by selecting the library that contains the client, selecting Import from its pop-up menu, and selecting the *.srx* files from the dialog box that displays. The order in which you import *.srx* files is significant—you cannot import proxies that depend on other classes until you have imported the proxies for the dependent classes.

Viewing the generated proxies

The generated proxies display in the System Tree. You can expand the proxy nodes to display the signatures of the methods on the home and remote interfaces for the EJB component, as well as on all the other objects for which proxies were generated.



pbejbclient105.pbd (D:\Program Files\
ejbproxies.pbl (D:\work)
abstractcollection
abstractlist
arraylist
cloneable
collection
collections
comparable
comparator
createexception
dictionary
ejbhome
ejbmetadata
ejbobject
enumeration
filteroutputstream
finderexception
handle
hashtable
homehandle
illegalargumentexception
inputstream
ioexception
iterator
java_integer
list
listiterator
map
number

Conflicts with reserved words

If the name of a component method conflicts with a PowerBuilder reserved word, the string _j is appended to the method name in the proxy so that the methods can be imported into PowerBuilder. For example, the Java Iterator class has a Next method, which conflicts with the PowerBuilder reserved word NEXT. In the proxy, the method is named next_j.

Datatype mappings

The EJB Proxy generator maps datatypes between Java and PowerBuilder as shown in the following table:

Java type	PowerBuilder type
short	Integer
int	Long
long	LongLong
float	Real
double	Double
byte	Int
char (16-bit unsigned)	Char
java.lang.String	String
boolean	Boolean
java.util.Date	Datetime
Array of primitive type	Parameters: Array of primitive type Return values: Any
Array of java.lang.String or java.util.Date objects	Parameters: Array of String or DateTime Return values: Any
Array of arrays	Any
Java class arguments or return values	PowerBuilder proxies of Java classes
Other	Any

Different precision for double

A PowerBuilder double has 15 digits of precision (1.79769313486231E+308) and a Java double has 17 digits (1.7976931348623157e+308). For EJB client applications, the precision of a double is limited to the PowerBuilder range (2.2250738585073E-308 to 1.79769313486231E+308).

Arrays of arrays

Unlike Java, PowerBuilder does not support unbounded multidimensional arrays. If a Java method takes an array of arrays as a parameter, the corresponding PowerBuilder proxy method takes a parameter of type Any. To call the method in PowerBuilder, declare a PowerBuilder array with the same dimensions as the Java array, and pass the array as the parameter.

Creating a Java VM

Before calling an EJB component, you need to create a Java VM using the CreateJavaVM method of the JavaVM class. The first argument is a string that specifies a classpath to be added to the beginning of the classpath used by the Java VM.

A Java VM might already be loaded
The classpath argument is ignored if the Java VM is already running.

The second argument to createJavaVM is a boolean that specifies whether debug information is written to a text file. See "Debugging the client" on page 613.

The JavaVM class has other methods that you can use when you create a Java VM:

- The CreateJavaInstance method creates an instance of the Java object from a proxy name.
- The IsJavaVMLoaded method determines whether the Java VM is already loaded. Use this method before calling CreateJavaVM if you want to enable or disable some features of your application depending on whether the Java VM has already been loaded. This will ensure that the classpath argument passed to CreateJavaVM is ignored.
- The GetJavaVMVersion method determines which version of the Java VM is running.
- The GetJavaClasspath method determines the runtime classpath of the Java VM.

The JavaVM that you create using CreateJavaVM should be a global or instance variable for the client application and should not be destroyed explicitly.

The Java VM classpath in the development environment

When PowerBuilder starts a Java VM, the Java VM uses internal path and classpath information to ensure that required Java classes are always available.

In the development environment, you can check whether the JVM is running and, if so, which classpath it is using, on the Java page of the System Options dialog box. The classpath is constructed by concatenating these paths:

- A classpath added programmatically when the Java VM is started. For example, the classpath you pass to the CreateJavaVM method.

- The PowerBuilder runtime static registry classpath. This path is built into the *pbjvm110.dll* and contains classes required at runtime for EJB clients and other PowerBuilder features that use a Java VM.
- The PowerBuilder system classpath. This path resides in a Windows registry key installed when you install PowerBuilder. It contains classes required at design time for Java-related PowerBuilder features such as JSP targets and JDBC connectivity.
- The PowerBuilder user classpath. This is the path that you specify on the Java page of the System Options dialog box.
- The system CLASSPATH environment variable.
- The current directory.

The runtime Java VM classpath

At runtime, you can use the GetJavaClasspath method to determine what classpath the Java VM is using. The Java VM uses the following classpath at runtime:

- A classpath added programmatically when the Java VM is started
- The PowerBuilder runtime static registry classpath
- The system CLASSPATH environment variable
- The current directory

For more information about the Java classpath at runtime, see “Java support” on page 786.

Classes required by servers

The classpath contains the classes required by EJB clients for EAServer 4.2. If you are using a different J2EE server, you need to add additional classes required by the application server to the system CLASSPATH. For example:

- For WebLogic, *weblogic.jar*. This file is installed in *wlserver6.1\lib* or *weblogic700\server\lib* on the server.
- For WebSphere, JAR files installed on the server in *websphere\appserver\lib*.

For detailed information about the files required on the client by each application server, see the documentation for the server.

Examples

This example demonstrates the creation of an instance of the Java VM that specifies the *html\classes* folder in an EAServer installation as a class path:

```
// global variables javavm g_jvm,
// boolean gb_jvm_started
boolean isdebug
string classpath
```

```
if NOT gb_jvm_started then
  //create JAVAVM
  g_jvm = create javavm

// The Java package for the EJB is in the
// EAServer html/classes folder
  classpath = &
  "D:\Program Files\Sybase\EAServer\html\classes;"

  isdebug = true
  choose case g_jvm.createJavaVM(classpath, isdebug)
  case 0
    gb_jvm_started = true
  case -1
    MessageBox("Error", "Failed to load JavaVM")
  case -2
    MessageBox("Error", "Failed to load EJBLocator")
  end choose
end if
```

This additional code can be added to the previous example to create a record of the Java VM version and classpath used:

```
integer li_FileNum
string ls_classpath, ls_version, ls_string

li_FileNum = FileOpen("C:\temp\PBJavaVM.log", &
   LineMode!, Write!, LockWrite!, Append!)

ls_classpath = i_jvm.getjavaclasspath()
ls_version = i_jvm.getjavavmversion()
ls_string = String(Today()) + " " + String(Now())
ls_string += " Java VM Version: " + ls_version
ls_string += " ~r~n" + ls_classpath + "~r~n"

FileWrite(li_FileNum, ls_string)
FileClose(li_filenum)
```

Connecting to the server

The EJBConnection class is used to connect to an EJB server and locate an EJB. It has four methods: ConnectToServer, DisconnectServer, Lookup, and GetEJBTransaction.

To establish a connection to the server, you need to execute the PowerScript statements required to perform these operations:

1. Declare an instance of the EJBConnection class.
2. Set properties for the EJBConnection object.
3. Use the CREATE statement to instantiate the EJBConnection object.
4. Invoke the ConnectToServer method to establish a connection to the server.
5. Check for errors.

Class path requirements

To connect to the application server and create an EJB object, the system CLASSPATH environment variable or the *classpath* argument of createJavaVM must contain the location of the EJB stub files, either a directory or a JAR file. The application server you are using might also require that some classes or JAR files be available on the client computer and added to the class path. For more information, see “The Java VM classpath in the development environment” on page 601.

Setting the initial context

The string used to establish the initial context depends on the EJB server. The following table shows sample string values. See the documentation for your server for more information.

Server	INITIAL_CONTEXT_FACTORY value
EAServer	com.sybase.ejb.InitialContextFactory
WebLogic	weblogic.jndi.WLInitialContextFactory
WebSphere	com.ibm.websphere.naming.WsnInitialContextFactory

Example

The following script shows a connection to EAServer. It sets connection properties to create an initial context, to identify the host name and port number of the server, and to identify the user ID and password.

IIOPS
IIOPS connections are not currently supported.

Then, the script creates an instance of the EJBConnection object, invokes the ConnectToServer method to establish a connection to the server, and checks for errors:

```
ejbconnection conn
string properties[]

properties[1]="javax.naming.Context.INITIAL_CONTEXT_FACTORY=
com.sybase.ejb.InitialContextFactory"
properties[2]="javax.naming.Context.PROVIDER_URL=iiop://myejbserver:9000"
properties[3]="javax.naming.Context.SECURITY_PRINCIPAL=admin"
properties[4]="javax.naming.Context.SECURITY_CREDENTIALS="

conn = CREATE ejbconnection
TRY
  conn.connectToServer(properties)
CATCH (exception e)
  MessageBox("exception", e.getmessage())
END TRY
```

Disconnecting from the server

When your application has finished using the EJB server, it should disconnect from the server:

```
conn.disconnectserver()
```

Invoking component methods

After a connection to the server has been established and a proxy object or objects created, the client application can begin using the EJB components. To invoke an EJB component method, you need to execute the PowerScript statements required to perform these operations:

1. Use the lookup method of EJBConnection to access the component's home interface.
2. Invoke the create or findByPrimaryKey method on the home interface to create or find an instance of the component and get a reference to the component's remote interface.
3. Invoke the business methods on the remote interface.

This procedure relies on the *pbejbclient110.jar* file, which is included in the Java VM classpath automatically at design time and runtime by the *pbjvm110.dll*.

Using the *lookup* method

The lookup method takes three string arguments: the name of the proxy for the home interface, the JNDI name of the EJB component, and the fully qualified home interface name of the EJB component.

The home interface name is the fully qualified class name of the EJB home interface. For example, if the class's location relative to the Java naming context is ejbsample, the home interface name is `ejbsample.HelloEJBHome`.

The following example shows the invocation of the lookup method for HelloEJB on WebLogic.

```
HelloEJBHome homeobj

homeobj = conn.lookup("HelloEJBHome",
  "ejbsample.HelloEJB", "ejbsample.HelloEJBHome")
```

Lookup is case sensitive
Lookup in EJB servers is case sensitive. Make sure that the case in the string you specify for the arguments to the lookup method matches the case on the server.

Creating or finding an instance of an EJB

A session bean is created in response to a client request. A client usually has exclusive use of the session bean for the duration of that client session. An entity bean represents persistent information stored in a database. A client uses an entity bean concurrently with other clients. Since an entity bean persists beyond the lifetime of the client, you must use a primary key class name to find an instance of the entity bean if one exists or create a new instance if it does not.

For a session bean, you use the proxy object's create method to create the instance of the EJB. The create method can throw CreateException and RemoteException. Assuming that you have obtained a reference to the home interface in homeobj, create is used in the same way on all EJB servers:

```
HelloEJB beanobj
try
  beanobj = homeobj.create()
catch (remoteexception re)
  MessageBox("Remote exception", re.getmessage())
catch (createexception ce)
  MessageBox("Create exception", ce.getmessage())
end try
```

For an entity bean, you provide a primary key. The FindByPrimaryKey method can throw FinderException and RemoteException. In this example, the key is the ID of a specific customer that is passed as an argument to the function:

```
try
  beanobj = homeobj.findByPrimaryKey(customerID)
catch (remoteexception re)
  MessageBox("Remote exception", re.getmessage())
catch (finderexception fe)
  MessageBox("Finder exception", fe.getmessage())
end try
```

Invoking EJB component methods

When the bean instance has been created or found, you can invoke its methods. For example:

```
string msg
msg = beanobj.displaymessage()
```

Creating an instance of a Java class

If the bean has a method that accepts a Java class as an argument, you use the CreateJavaInstance method of the JavaVM object to create it. For example, if the primary key in a call to the findByPrimaryKey method is a Java class, you would use the CreateJavaInstance method to create that class, and then use a PowerBuilder proxy to communicate with it.

In this example, the create method accepts a Java Integer class argument. PowerBuilder creates a proxy called java_integer (the prefix java_ is required to prevent a conflict with the PowerBuilder integer type). The call to CreateJavaInstance sets the value of that variable so you can call the EJB create method:

```
CustomerRemoteHome homeobj
CustomerRemote beanobj
java_integer jint_a

try
  homeobj = conn.lookup("CustomerRemoteHome", &
  "custpkg/Customer", "custpkg.CustomerRemoteHome" )
catch (Exception e)
  MessageBox( "Exception in Lookup", e.getMessage() )
  return
end try

try
  g_jvm.createJavaInstance(jint_a, "java_integer")
  jint_a.java_integer("8")
  beanobj = homeobj.create( jint_a, sle_name.text )
```

```
catch (RemoteException re)
  MessageBox( "Remote Exception", re.getMessage() )
  return
catch (CreateException ce)
  MessageBox( "Create Exception", ce.getMessage() )
  return
catch (Throwable t)
  MessageBox(" Other Exception", t.getMessage())
end try

MessageBox( "Info", &
  "This record has been successfully saved " &
  + "~r~ninto the database" )
```

Downcasting return values

When Java code returns a common Java object that needs to be downcast for use in Java programming, the Java method always sets the return value as Java.lang.Object. In a PowerBuilder EJB client proxy, java.lang.Object is mapped to the any datatype. At runtime, PowerBuilder gets the correct Java object and indexes the generated mapping structure to get the PowerBuilder proxy name. The any value is set as this proxy object. If the returned Java object can map to a PowerBuilder standard datatype, the any value is set as the PowerBuilder standard datatype.

Suppose the remote interface includes the method:

java.lang.Object account::getPrimaryKey()

and the home interface includes the method:

account accounthome::findByPrimaryKey(java.lang.String)

The return value java.lang.Object is really a java.lang.String at runtime. PowerBuilder automatically downcasts the return value to the PowerBuilder string datatype:

```
any nid
try
  account beanobj
  homeobj = conn.lookup("AccountHome", &
    ejb20-containerManaged-AccountHome, &
    examples.ejb20.basic.containerManaged.AccountHome)
  beanobj = homeobj.create("101", 0, "savings")
  nid = beanobj.getPrimaryKey()
  accounts = homeobj.findByPrimaryKey(string(nid))
catch (exception e)
  messagebox("exception", e.getmessage())
end try
```

Dynamic casting

There are two scenarios in which a Java object returned from a call to an EJB method can be represented by a proxy that does not provide the methods you need:

- If the class of a Java object returned from an EJB method call is dynamically generated, PowerBuilder uses a proxy for the first interface implemented by the Java class.
- The prototype of an EJB method that actually returns *someclass* can be defined to return a class that *someclass* extends or implements. For example, a method that actually returns an object of type java.util.ArrayList can be defined to return java.util.Collection.java.util.ArrayList, which inherits from java.util.AbstractList, which inherits from java.util.AbstractCollection, which implements java.util.Collection. In this case, PowerBuilder uses a proxy for java.util.Collection.

The DynamicCast method allows you to cast the returned proxy object to a proxy for the interface you require, or for the actual class of the object returned at runtime so that the methods of that object can be used.

You can obtain the actual class of the object using the GetActualClass method. You can also use the DynamicCast method with the GetSuperClass method, which returns the immediate parent of the Java class, and the GetInterfaces method, which writes a list of interfaces implemented by the class to an array of strings.

For example, given the following class:

```
public class java.util.LinkedList extends java.util.AbstractSequentialList
implements java.util.List, java.lang.Cloneable, java.io.Serializable
```

GetActualClass returns java.util.LinkedList, GetSuperClass returns java.util.AbstractSequentialList, and GetInterfaces returns 3 and writes three strings to the referenced string array: java.util.List, java.lang.Cloneable, and java.io.Serializable.

Java collection classes

EJB proxy generation generates Java common collection classes such as Enumeration, Iterator, Vector, and so forth. PowerBuilder can manipulate these collection classes in the same way as a Java client.

For example, suppose the home interface includes the following method with the return value java.util.Enumeration:

```
Enumeration accounthome:: findNullAccounts ()
```

The following code shows how a PowerBuilder EJB client can manipulate the enumeration class through the PowerBuilder proxy:

```
Enumeration enum
try
  enum = homeobj.findNullAccounts()
  if (not enum.hasMoreElements())  then
  msg = "No accounts found with a null account type"
  end if
catch (exception e)
  messagebox("exception", e.getmessage())
end try
```

Exception handling

Errors that occur in the execution of a method of an EJB component are mapped to exception proxies and thrown to the calling script. The methods of all the classes in *pbejbclient110.pbx* can also throw exceptions when, for example, connection to the server fails or the component cannot be located or created.

Building EJB proxy projects generates the proxies for the home and remote interfaces, proxies for any Java classes referenced by the EJB, proxies for ancestor classes, and proxies for any exceptions that can be thrown by the EJB and its supporting classes. The following exception proxies are among those that may display in the System Tree:

Proxy name	Java object name
createexception	javax.ejb.CreateException
ejbexception	javax.ejb.EJBException
finderexception	javax.ejb.FinderException
remoteexception	java.rmi.RemoteException
removeexception	javax.ejb.RemoveException

Catching exceptions

A client application can handle communications errors in a number of ways. For example, if a client connects to a server and tries to invoke a method for an object that does not exist, the client can disconnect from the server, connect to a different server, or retry the operation. Alternatively, the client can display a message to the user and give the user the opportunity to control what happens next.

When an error occurs, if the client connects to a new server to retry the operation, it must instantiate the remote object on the new server before invoking a method of the remote object.

In the following example, the script simply displays a message box when a specific exception occurs:

```
// function char getChar() throws RemoteException
try
  conn.connectToServer(properties)
  mappinghome = conn.lookup("pbEjbMappingHome",
   "pbEjbTest/pbEjbMappingBeanSL",
   "pbejb.pbEjbMappingHome")
  mapping = mappinghome.create()
  ret = mapping.getChar()
  messagebox("char from EJB", ret)
catch (remoteexception re)
  messagebox("remoteexception", re.GetMessage())
catch (createexception ce)
  messagebox("createexception", ce.GetMessage())
end try
```

Unhandled exceptions

If no exception handler exists, or if the existing exception handlers do not handle the exception, the SystemError event on the Application object is executed. If the SystemError event has no script, an application error occurs and the application is terminated.

Client-managed transactions

EJB client applications can control transactions on the server using the EJBTransaction object. This object has methods that enable the client to begin, commit, or roll back a transaction. The client can also get the status of a transaction, change its timeout value, or modify the transaction so that it cannot be committed.

The EJBTransaction methods map directly to the methods of the javax.transaction.UserTransaction interface, which is documented in the JTA Specification on the Sun Java Web site at http://java.sun.com/products/jta.

Beginning and ending transactions

Clients can obtain access to the methods of the EJBTransaction class by calling the getEJBTransaction method of the EJBConnection class:

```
ejbconnection conn
ejbtransaction trans
string properties[]

conn = create ejbconnection
TRY
  conn.connectToServer(properties)
  trans = conn.getEJBTransaction()
CATCH (exception e)
  messagebox("exception", e.getmessage())
END TRY
```

If an EJBTransaction instance is obtained successfully, you use its begin method to start the transaction and its commit or rollback methods to end it:

```
TRY
  // Start the transaction
  trans.begin()
  // Create a component and call methods to be executed
  // within the transaction
  ...
  // Commit the transaction
  trans.commit();
CATCH (exception e)
  messagebox("exception", e1.getmessage())
  trans.rollback()
END TRY
```

Getting information about the transaction

GetStatus returns an integer that indicates whether the transaction is active, has been marked for rollback, is in the prepare phase or commit phase, or has been committed or rolled back.

Setting a timeout period for transactions

A calling thread can specify a timeout period after which a transaction will be rolled back. This example sets the timeout period to 3 minutes (180 seconds):

```
trans.SetTimeout(180)
trans.Begin()
```

Debugging the client

The createJavaVM method of the JavaVM class takes a boolean value as a second argument. If this second argument is "true", execution information, including class loads, are logged to the file *vm.out* in the directory where the application resides:

```
// global variable: JavaVM g_jvm
string classpath
boolean isdebug

classpath = "d:\tests\ejbsample;"
isdebug = true
g_jvm.createJavaVM(classpath, isdebug)
```

PART 7

Developing Web Applications

This part presents tools and techniques for developing Web applications with PowerBuilder.

CHAPTER 30 Web Application Development with PowerBuilder

About this chapter

This chapter provides an overview of the techniques you can use to develop Web applications with PowerBuilder.

Contents

Building Web applications

PowerBuilder provides several tools that you can use to build Web applications. This section provides a brief overview of these tools and points to where you can find more information.

Appeon for PowerBuilder
Appeon for PowerBuilder is a product that deploys existing PowerBuilder client/server applications to the Web. For more information, see the Sybase web site at http://www.sybase.com.

.NET Web Forms applications and components

The PowerBuilder .NET Web Forms solution employs ASP.NET technology. It has a three-tier architecture, with the browser client as the front end, and the PowerBuilder components on the IIS server as the middle tier. The database tier remains unchanged.

Moving an existing application from client-server architecture to three-tier Web architecture typically requires a significant effort in modifying the application code and the tolerance of various functionality restrictions due to constraints of the Web environment. The PowerBuilder .NET Web Forms solution is intended to greatly ease the deployment of existing client-server applications to the Web and to allow you to use your PowerBuilder skills to create new Web applications.

PowerBuilder includes targets for creating .NET assemblies and .NET Web service applications from PowerBuilder nonvisual objects.

For more information, see *Deploying Applications and Components to .NET*.

Web services

Web services are loosely defined as the use of Internet technologies to make distributed software components talk to each other without human intervention. The software components might perform such business logic as getting a stock quote, searching the inventory of a catalog on the Internet, or integrating the reservation services for an airline and a car rental agency. You can reach across the Internet and use preexisting components, instead of having to write them for your application.

A PowerBuilder application can act as a client consuming a Web service that is accessed through the Internet. Through use of SOAP and WSDL, a collection of functions published remotely as a single entity can become part of your PowerBuilder application. A Web service accepts and responds to requests sent by applications or other Web services. You can also use Web services in a JSP target.

For more information about Web services, see Chapter 31, “Building a Web Services Client,” and the manual *Working with JSP Targets*.

JSP targets

JavaServer Pages (JSP) technology provides a quick, easy way to create Web pages with both static and dynamic content. JSPs are text-based documents that contain static markup, usually in HTML or XML, as well as Java content in the form of scripts and/or calls to Java components. JSPs extend the Java Servlet API and have access to all Java APIs and components.

You can use JSPs in many ways in Web-based applications. As part of the J2EE application model, JSPs typically run on a Web server in the middle tier, responding to HTTP requests from clients, and invoking the business methods of Enterprise JavaBeans (EJB) components on a transaction server.

JSP pages built with PowerBuilder support version 1.2 of the JavaServer Pages specification, version 2.3 of the Java Servlet specification, and JDK 1.3 and later. PowerBuilder also supports custom tag libraries that use the JSP 1.2 format. You can choose to deploy a JSP target as a Web application to EAServer, Apache Tomcat, or any JSP 1.2 server for which you can configure command line deployment capabilities.

For more information about JSP targets, see *Working with JSP Targets*.

Web DataWindow

The Web DataWindow is a thin-client DataWindow implementation for Web applications. It provides most of the data manipulation, presentation, and scripting capabilities of the PowerBuilder DataWindow without requiring any PowerBuilder DLLs on the client.

The Web DataWindow uses the services of several software components that can run on separate computers:

- Web DataWindow server component running in an application or transaction server
- Dynamic page server
- Web server
- Web browser
- Database

The server component is a nonvisual user object that uses a DataStore to handle retrieval and updates and generate HTML. You can use the generic component provided with PowerBuilder or a custom component.

There are several ways you can take advantage of the capabilities of the Web DataWindow:

- **Web DataWindow design-time control** In a JSP target, you can use the Web DataWindow design-time control to generate server-side scripts that access the Web DataWindow component. *This is the easiest way to use the Web DataWindow.*
- **Hand coding against the Web Target object model** In a JSP target, you can write server-side scripts that use the Web Target object model to access the Web DataWindow component. The Web Target object model provides a set of predefined objects and methods that simplify coding against the Web DataWindow component.
- **Hand coding against the Web DataWindow component itself** You can write server-side scripts that access the Web DataWindow component directly.
- **Writing your own HTML generator** Using a sample PBL provided with PowerBuilder as a starting point, you can create your own HTML generator that provides the methods you need for your application.

For more information about the Web DataWindow, see the *DataWindow Programmer's Guide*.

DataWindow Web control for ActiveX

The DataWindow Web control for ActiveX is a fully interactive DataWindow control for use with Internet Explorer. It implements all the features of the PowerBuilder DataWindow except rich text.

The DataWindow Web control for ActiveX supports data retrieval with retrieval arguments and data update. You can use edit styles, display formats, and validation rules. Most of the PowerBuilder methods for manipulating the DataWindow are available. Several functions that involve file system interactions are not supported, allowing the Web ActiveX to be in the *safely scriptable* category of ActiveX controls.

Included with the DataWindow Web control is the DataWindow Transaction Object control for making database connections that can be shared by several DataWindow Web controls.

The Web ActiveX is provided as a CAB file, which allows the client browser to install and register the control. When the user downloads a Web page that refers to the CAB file, the browser also downloads the CAB file if necessary, unpacks it, and registers the control.

For more information about the DataWindow Web control for ActiveX, see the *DataWindow Programmer's Guide*.

DataWindow plug-in

The DataWindow plug-in displays Powersoft reports (PSRs) that have already been generated and stored on the Web server. Because the PSR files have already been generated, database access is not necessary. The plug-in supports only read-only access—users can view, print, and save the reports, but they cannot modify them.

The plug-in works with any browser that supports Netscape plug-ins, including Netscape Navigator. (Microsoft Internet Explorer 5.5 Service Pack 2 and later versions do not support Netscape plug-ins.) Only the DataWindow plug-in DLL is required on the client browser.

For more information, see Chapter 32, "Using the DataWindow Plug-in."

PowerBuilder window plug-in

The PowerBuilder window plug-in runs a PowerBuilder application that displays a PowerBuilder window in an HTML page on a client workstation. It works with any browser that supports Netscape plug-ins, including Netscape Navigator (Microsoft Internet Explorer 5.5 Service Pack 2 and later versions do not support Netscape plug-ins). A secure version of the plug-in ensures that PowerBuilder applications downloaded over the Internet will not damage a client system or access information on a client workstation.

An application in the PowerBuilder window plug-in can execute most PowerBuilder functionality as long as the application begins with a child window. The application can run any PowerBuilder windows, display information, accept data input, and update databases. Database access is initiated on the client workstation using the client's connections.

A major disadvantage of the plug-in is that it requires a fat client—the browser client requires the PowerBuilder runtime DLLs or shared libraries and the PowerBuilder window plug-in DLL or shared library.

For more information, see Chapter 33, "Using the PowerBuilder Window Plug-in."

PowerBuilder window ActiveX

The PowerBuilder window ActiveX lets you provide a graphical interface inside HTML pages when using a Web browser that supports ActiveX. It provides all the capabilities of the window and DataWindow plug-ins, plus the ability to use JavaScript or VBScript to access a subset of a PowerBuilder child window's events and functions. It includes methods you can call to invoke functions and events in the child window contained in the window ActiveX control.

An application in the PowerBuilder window ActiveX can execute most PowerBuilder functionality as long as the application begins with a child window. The application can run any PowerBuilder windows, display information, accept data input, and update databases. Database access is initiated on the client workstation using the client's connections.

Like the window plug-in, the major disadvantage of the window ActiveX is that it requires a fat client—the browser client requires the PowerBuilder runtime DLLs and the PowerBuilder window ActiveX.

For more information, see Chapter 34, "Using the PowerBuilder Window ActiveX."

CHAPTER 31 Building a Web Services Client

About this chapter

This chapter describes how to use Web services in a PowerBuilder application. Reference information for the objects described in this chapter is in the *PowerBuilder Extension Reference* and in the online Help.

Contents

About Web services

Web services allow you to use preexisting components (available on the Internet or on a local network) instead of writing new business logic to perform common tasks invoked by the applications that you develop. Web services originated when the Simple Object Access Protocol (SOAP) was introduced. SOAP leverages Extensible Markup Language (XML) and usually employs Hypertext Transfer Protocol (HTTP) as the transport. Invoking Web services through SOAP requires serialization and deserialization of datatypes, and the building and parsing of SOAP messages.

Part of the value of Web services comes from the Web Services Description Language (WSDL), which enables a service to be self-describing. WSDL defines an XML grammar for describing Web services as collections of communication endpoints capable of exchanging messages. WSDL service definitions provide documentation for distributed systems and serve as a recipe for automating the details involved in applications communication.

Web services that are described in WSDL files can be registered on a Universal Description, Discovery, and Integration (UDDI) Web site. You can search UDDI registry sites from PowerBuilder at design time and find the services you need for your application.

With SOAP, WSDL, and UDDI, using third-party components is easier because interfaces between applications become standardized across disparate platforms.

PowerBuilder supports the following Web services standards:

- SOAP 1.1 or later
- WSDL 1.1 or later
- HTTP or HTTPS

You can access Web services from a PowerScript target or a JSP target. For information on accessing Web services in JSP applications that you create with PowerBuilder, see the chapter on JSP targets in the *Working with JSP Targets* book or the JSP page authoring topic in the online Help.

Producing a Web service
PowerBuilder provides tools for developing custom class (nonvisual) user objects and deploying them as EAServer components and exposing them as Web services. You can deploy a component to an EAServer host running on Windows and UNIX operating systems. For more information, see Chapter 24, “Building an EAServer Component.”

About building a Web services client

A PowerBuilder application can act as a client consuming a Web service that is accessed through the Internet. Using SOAP and WSDL, a collection of functions published remotely as a single entity can become part of your PowerBuilder application. A Web service accepts and responds to requests sent by applications or other Web services.

Invoking Web services through SOAP requires serialization and deserialization of data types, and the building and parsing of XML-based SOAP messages. Using objects from an extension file or dynamic library that installs with PowerBuilder, the Web services client proxy performs these tasks for you—thereby eliminating the need to have extensive knowledge of the SOAP specification and schema, the XML Schema specification, or the WSDL specification and schema.

Choosing a Web service engine

PowerBuilder lets you choose between the .NET Web service engine and the EasySoap Web service engine to construct SOAP requests and parse the SOAP messages returned from a Web service.

Using the .NET Web service engine

Generating a .NET assembly

The .NET Web service engine supports the latest Web service standards. To use this engine, you must have the *wsdl.exe* Web service tool on the development machine. This tool is required to parse WSDL files and generate C# code for a .NET assembly. The *wsdl.exe* file installs with the .NET SDK. It is not required on deployment machines, although deployment machines must have the .NET Framework to consume a Web service that depends on the .NET Web service engine.

If you select the .NET Web service engine in the Web Service Proxy wizard, the wizard generates a .NET assembly (DLL) in addition to a proxy object. To use the Web service at runtime, you must deploy the wizard-generated DLL along with your application.

You can also select the .NET Web service engine in the Project painter for a new Web service proxy. If you select the .NET Web service engine on the Web Service tab of the Properties dialog box for the Web Service Proxy Generator, PowerBuilder attempts to generate an assembly DLL after you click Apply or OK. You cannot use the Properties dialog box to change the Web service engine for a proxy that you already generated with the Web Service Proxy wizard.

Naming the DLL

You can name the DLL generated by the Web Service Proxy wizard or by the Project painter in the Proxy Assembly Name text box. You do not need to include the DLL extension. The name of the wizard-generated assembly is *Web_service*.DLL, where *Web_service* is the name you provide in the Proxy Assembly Name field. If you do not provide a name, the assembly takes the name of the Web service to be consumed by the DLL. The assembly is generated in the current target directory.

Deploying the DLL

You must deploy the DLL created for your Web service project to the directory where you deploy the client executable. You must also copy the *Sybase.PowerBuilder.WebService.Runtime.dll* and the *Sybase.PowerBuilder.WebService.RuntimeRemoteLoader.dll* system assemblies to this directory.

.

Extension objects

Although you use the same SOAP connection and exception-handling objects for the .NET Web service engine as for the EasySoap Web service engine, the objects that reference the .NET Web service engine require a different extension file or library.

The methods available on the SoapConnection object depend on which extension file or library you are using and on which Web service engine you are using. The methods for a .NET Web service engine allows you to include security information in the SOAP client header.

For more information, see "Importing objects from an extension file."

Using the EasySoap Web service engine

If you decide not to use the .NET SOAP engine, PowerBuilder uses the EasySoap Web service engine. Earlier releases of PowerBuilder supported the EasySoap Web service engine only. Unlike the .NET Web service engine, the EasySoap engine does not support the XML-type array datatype or header sections in SOAP message envelopes. The EasySoap Web service engine is retained for backward compatibility and for use with targets deployed to UNIX machines.

You set the Web service engine that you want to use on the first page of the Web Service Proxy Wizard or on the Web Service tab of the Property sheet for a Web service project. The Use .NET Engine check box is selected by default for new Web service projects. You must clear the check box if you are developing a Web service application that you intend to deploy to UNIX machines.

Assigning firewall settings to access a Web service

When you add a Web service at design time and your development machine is behind a firewall, you must assign proxy server settings to connect to the Internet.

Table 31-1 displays the design-time proxy server settings that you can enter on the Firewall Settings page of the PowerBuilder System Options dialog box. To enter runtime proxy server settings, you must use the SoapConnection SetProxyServer or the SetProxyServerOptions methods.

For information about the SetProxyServer or the SetProxyServerOptions methods, see the *PowerBuilder Extension Reference* in the online Help.

Table 31-1: Design-time firewall settings

Firewall setting	Description
Proxy host	Name of the proxy server that you use to access Web pages
Port	The port used for connecting to the proxy server
User name	User name for accessing the proxy server
Password	Password for the user accessing the proxy server

PowerBuilder uses the values you enter for the proxy server settings only if you also select the Use Above Values as System Defaults check box on the Firewall Setting page. The type of engine you select for consuming a Web service can also affect the settings that PowerBuilder uses to connect to the Internet at design time.

.NET Web service engine If the development machine is located behind a firewall but you do not select the Use Above Values as System Defaults check box, PowerBuilder attempts to connect to the Internet using settings entered in the Internet Options dialog box of the Internet Explorer browser. The selections you make on the Firewall Setting page have no effect if the development machine is not located behind a firewall.

EasySoap Web service engine If you do not select the Use Above Values as System Defaults check box, PowerBuilder assumes that the development machine is not behind a firewall and makes no attempt to use settings from the Internet Options dialog box of the Internet Explorer browser. If you select the Use Above Values as System Defaults check box, but the development machine is not located behind a firewall, the Web service invocation can fail.

Importing objects from an extension file

Invoking Web services through SOAP requires serialization and deserialization of data types, and the building and parsing of XML-based SOAP messages.

The *pbwsclient110.pbx* file contains objects for the .NET Web service engine that enable you to perform these tasks without extensive knowledge of the SOAP specification and schema, the XML Schema specification, or the WSDL specification and schema. You can use these objects after you import the extension file into a PowerBuilder Web service application.

If you use the EasySoap Web service engine, you can import the *pbsoapclient110.pbx* file or the *pbwsclient110.pbx* file into your PowerBuilder applications. However, the *pbwsclient110.pbx* file requires the .NET 2.0 Framework on design-time and runtime machines, even if you are not using the .NET Web service engine. Both extension files contain the same objects, and you use these objects and their methods in similar ways.

Using a PBD file

In earlier releases of PowerBuilder, instead of importing an extension file, you needed to add a *PBD* file to the application library list. Although this is no longer necessary, the setup program installs *PBD* files (containing the same SoapConnection and SoapException objects as the extension files) in the *Sybase\Shared\PowerBuilder* directory. You can use the *pbwsclient110.pbd* or the *pbsoapclient110.pbd* instead of importing object definitions from the *pbwsclient110.pbx* or *pbsoapclient110.pbx* file.

To add definitions from a PowerBuilder extension file to an application library, right-click the library in the System Tree and select Import PB Extensions from the pop-up menu. Browse to the *Sybase\Shared\PowerBuilder* directory and select the extension file that you want to use.

After you import the *PBWSClient110.pbx* or the *PBSoapClient110.pbx* file to your application, the following objects display in the System Tree:

Object	Description
soapconnection	Used to connect to a SOAP server
soapexception	Used to catch exceptions thrown from soapconnection

When you create a Web service client application, you must deploy the extension file that you use along with the client executable to a directory in the application's search path. You can use the Runtime Packager tool to automatically include the extension files required by your Web service applications.

Generating Web service proxy objects

Creating a Web service proxy object

To create a new Web service proxy, select the Web Service Proxy Wizard icon from the Projects page in the New dialog box. The Web Service Proxy Wizard helps you create the proxy so you can use the Web service in PowerScript. If you select the EasySoap Web service engine, one proxy is created for each port.

In the wizard you specify:

- Which Web service engine you want to use
- Which WSDL file you want to access
- Which service within the WSDL file you want to select
- Which port or ports you want to use (EasySoap engine only)
- What prefix you want to append to a port name (EasySoap) and include in the proxy name (EasySoap and .NET engines)
- Which PowerBuilder library you want to deploy the proxy to

You can also select the Web Service Proxy icon from the Projects page in the New dialog box. The Web Service Proxy icon opens the Project painter for Web services so that you can create a project, specify options, and build the proxy library. The new project lists the Web service (and, for the EasySoap engine, the ports for which proxies will be generated) and specifies the name of the output library that will contain the generated proxy objects.

Whether you create the Web service project through the wizard or in the painter, the final step is to build the proxy objects by clicking the Build icon on the painter bar or selecting Design>Deploy project from the menu bar.

Circular references
Generation of a Web service proxy from a WSDL file that contains a circular reference is not supported in PowerBuilder. An example of such a “circular reference” is a structure that includes itself as a child class member.

Using the UDDI browser in the wizard

PowerBuilder provides live access to Universal Description, Discovery, and Integration (UDDI) registries for both PowerScript and JSP targets. The UDDI service is an industry-wide effort to bring a common standard for business-to-business integration. It defines a set of standard interfaces for accessing a database of Web services.

The UDDI browser is incorporated in the Web Service Proxy wizard as well as in the JSP Web Service Proxy wizard. You open UDDI search pages by clicking the Search From UDDI button on the Select WSDL File page of these wizards or on the Web Service page of the properties dialog box for a Web Service Proxy Generator project. The UDDI Search page has search fields and options listed in Table 31-2.

Table 31-2: UDDI search fields and options

Search field or option	Description
UDDI profile	Editable drop-down list for the name of a UDDI operator. You can associate a UDDI profile with a query URL. The drop-down list allows you to select predefined profiles for the Microsoft and IBM public UDDI registries.
Query URL	Text box that displays the URL for the Web service registry in which you want to find a Web service. If you selected a predefined profile in the UDDI Profile drop-down list, the URL associated with that profile displays in the text box. You can also enter a query URL and associate the URL with a profile name by clicking the Save Profile button.
Search For	Text box for entering the key word you want to use in a UDDI search.
In	Drop-down list for "Service Names" (default) or "Business Names."
Exact Match	Check box option. If selected, limits search to the current value in the Search For drop-down list.
Case Sensitive	Check box option. If selected, limits the search to the capitalization used by the current value in the Search For drop-down list.
Sort	Radio button option. Sorts search results in ascending or descending order.
Maximum Rows	Spin button option. Limits the number of search results returned to the number that you enter in this spin button control.

The next wizard page in the UDDI search depends on whether you are searching a key word in business names or service names:

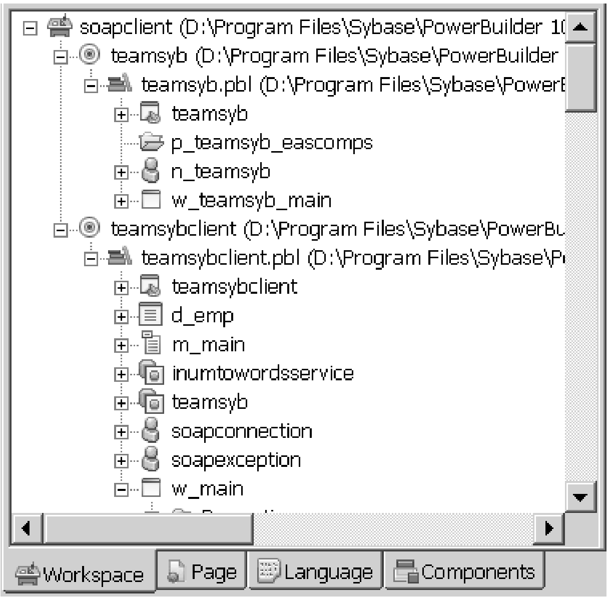
- **For a business name search** The Select Business wizard page returns a list of business names and descriptions that meet your search criteria. After you select a business name and click Next, a list of service names is returned on the Select Service wizard page, along with a service description and WSDL file name for each service listed.
- **For a service name search** The Select Service wizard page returns a list of service names along with a business name, service description, and WSDL file name for each service listed.

After you select a service on the Select Service page of a wizard, the UDDI search is complete and you continue your selections on the remaining pages of the wizard.

The Web Service page of the properties dialog box for a Web Service proxy object displays the WSDL file selection that you made in the Web Service Proxy wizard. It also allows you to modify that search through a UDDI search wizard that contains the same search options and search result lists as the UDDI search pages in the Web Service Proxy wizard.

Generated proxies

The generated proxies display in the System Tree. You can expand the proxy nodes to display the signatures of the methods.


soapclient (D:\Program Files\Sybase\PowerBuilder 1(
teamsyb (D:\Program Files\Sybase\PowerBuilder
teamsyb.pbl (D:\Program Files\Sybase\PowerE
teamsyb
p_teamsyb_eascomps
n_teamsyb
w_teamsyb_main
teamsybclient (D:\Program Files\Sybase\PowerBu
teamsybclient.pbl (D:\Program Files\Sybase\P
teamsybclient
d_emp
m_main
inumtowordsservice
teamsyb
soapconnection
soapexception
w_main
Workspace
Page
Language
Components

Aliases for XML methods

PowerBuilder is not case sensitive, whereas XML, SOAP, C#, and .NET are. To ensure that PowerScript code can call XML methods correctly, each method in the proxy uses an alias. The string that follows `alias for` contains the name and the signature of the corresponding XML or SOAP method in case-sensitive mode.

For example:

```
function real getquote(string ticker) alias for
getQuote(xsd:string symbol)#
return xsd:float StockPrice@urn:xmethods-delayed-
quotes@SoapAction
```

Web services across time zones

When an application consumes a Web service that uses the date, time, or datetime datatypes, it is possible that the service implementation processes and returns different data for application users who access the service from different time zones. This is typically the result of design considerations of the Web service and not the result of precision differences or translation errors between the Web service and the application that calls it.

Datatype mappings for EasySoap Web service engine

The Web service proxy generator maps datatypes between XML and PowerBuilder if you use the EasySoap Web engine, and between XML, C#, .NET, and PowerBuilder if you use the .NET Web service engine. All XML data types are based on schemas from the World Wide Web Consortium Web site at http://www.w3.org/1999/XMLSchema and at http://www.w3.org/2001/XMLSchema.

Table 31-3 shows the datatype mappings between XML and PowerScript. If you use the .NET Web service engine, datatypes are converted to C#, then to .NET datatypes. (Table 31-4 and Table 31-5 show datatype mappings used with the .NET Web service engine.)

Table 31-3: Datatype mappings between XML and PowerBuilder

XML Type	PowerScript Type
boolean	boolean
byte (-128 to 127) or short	int
unsignedByte (0 to 255) or unsignedShort	uint
int	long
unsignedInt	ulong
long (-9223372036854775808 to 9223372036854775807), unsignedLong (0 to 9223372036854775807), integer (-9223372036854775808 to 9223372036854775807), nonNegativeInteger (0 to 9223372036854775807), negativeInteger (-1 to -9223372036854775808), nonPositiveInteger (0 to -9223372036854775808), or positiveInteger (1 to 9223372036854775807)	longlong
decimal (-999999999999999999 to 999999999999999999)	decimal
float	real
double	double

XML Type	PowerScript Type
gYear, gYearMonth, gMonthDay, gDay, anyURI, QName, NOTATION, string, normalizedSting, token, or datatypes derived from token **About normalizedString, token, and derived datatypes** A normalized string does not contain carriage return, line feed, or tab characters. A token is similar to similar to a normalizedString, but does not contain leading or trailing spaces or an internal sequence of two or more spaces. Datatypes that derive from token include language, Name, NCName, NMTOKEN, NMTOKENS, ID, IDREF, IDREFS, ENTITY, ENTITIES.	string
date	date
time	time
dateTime	datetime
base64, base64Binary, or hexBinary	blob

Datatype mappings for .NET Web service engine

When you use the .NET Web Service engine, PowerBuilder converts the XML from WSDL files to C# code and compiles it in a .NET assembly. Table 31-4 displays datatype mappings for these conversions.

Table 31-4: Datatype mappings for the .NET Web service engine

XML type	C# type	.NET type
int	int	System.Int32
unsignedInt	uint	System.UInt32
boolean	bool	System.Boolean
unsignedByte	Byte	System.Byte
short	short	System.Int16
unsignedShort	ushort	System.UInt16
long	long	System.Int64
unsignedLong	ulong	System.UInt64
Decimal	Decimal	System.Decimal
Float	Float	System.Float
Double	Double	System.Double
Datetime, Date, and Time	System.DateTime	System.DateTime
hexBinary and hex64Binary	Byte [ ]	System.Byte [ ]

XML type	C# type	.NET type
nonNegativeInteger, negativeInteger, nonPositiveInteger, positiveInteger, gYear, gMonth, gMonthDay, gDay, duration, anyURI, QName, NOTATION, normalizedString, token, language, NMTOKEN, NMTOKENS, Name, NCName,ID, IDREF, IDREFS, ENTITY, ENTITIES, and String	String	System.String
AnyType	Object	System.Object

Table 31-5 displays the datatype mapping between C# datatypes and PowerBuilder.

Table 31-5: Datatype mappings between C# and PowerBuilder

C# type	PowerScript type
byte	byte
sbyte	int
short	int
int	long
long	longlong
ushort	uint
uint	ulong
ulong	longlong
float	real
double	double
object	any
char	uint
string	string
decimal	decimal
bool	boolean
System.DateTime	datetime

Arrays of arrays

Unlike XML, PowerBuilder can support only unbounded one-dimensional arrays. If an array in a WSDL file is bounded and one-dimensional, PowerBuilder automatically converts it to an unbounded array. If an array in a WSDL file is multidimensional, the return type is invalid and cannot be used.

In function prototypes, PowerBuilder displays an array type as a PowerBuilder any type. You must declare an array of the appropriate type to hold the return value.

Connecting to a SOAP server

You use the SoapConnection object to connect to the SOAP server that hosts the Web service that you want to access. The SetOptions method on a SoapConnection object lets you set options such as the user ID and password for an HTTPS connection. For .NET Web services, you can also use authentication methods such as SetBasicAuthentication, SetCertificateFile and UseWindowsAuthentication.

Using multiple Web services in the same application
If you connect to multiple Web services that have different authentication requirements, you must instantiate multiple SoapConnection objects and set the appropriate values in the SetOptions method or in the other authentication methods of each connection object.

You use the CreateInstance method to create the client proxy instance to access the Web service.

For more information on SoapConnection object methods, see the *PowerBuilder Extension Reference* in the online Help.

Example

The following script creates a connection to a Web service on a SOAP server using the EasySoap Web service engine. It sets the connection properties using an endpoint defined in the CreateInstance method. If the endpoint is not defined in the CreateInstance method, a default URL stored in the proxy would be used. The script uses the SetSoapLogFile method to specify a log file. It displays a return value in a message box.

```
SoapConnection conn // Define SoapConnection
syb_currencyexchangeport proxy_obj // Declare proxy
long rVal, lLog
real amount

//Define endpoint. You can omit it, if you want to use
//the default endpoint inside proxy
string str_endpoint
```

```
str_endpoint = "http://services.xmethods.net:80/soap"
conn = create SoapConnection  //Instantiated connection

lLog = conn.SetSoapLogFile ("C:\mySoapLog.log")
// Set trace file to record soap interchange data,
// if string is "", disables the feature

rVal = Conn.CreateInstance(proxy_obj, &
   "syb_currencyexchangeport", str_endpoint)

// Create proxy object
try
   amount = proxy_obj.getrate("us","japan")
   // Invoke service
   messagebox("Current Exchange Rate", "One US Dollar"&
   + " is equal to " + string(amount) + " Japanese Yen")
catch ( SoapException e )
   messagebox ("Error", "Cannot invoke Web service")
   // error handling
end try
destroy conn
```

Invoking the Web service method

SoapConnection is used to create the Soap_proxy object with connection options that you set using SoapConnection object methods. Once a proxy object for a Web service is created, the client application can begin accessing the Web service. To invoke a Web service method, the proxy object must contain the following information:

- End point of service, obtained from a WSDL file
- Name space definition used in the SOAP method call
- Any structure definition, when applicable
- An instance variable for each returned structure array, since all returned arrays are any
- One or more SOAP methods and corresponding alias strings

Exception handling

Errors that occur in the execution of a method of a Web service are converted to SoapException objects and thrown to the calling script. The methods of the SoapConnection object in *PBWSClient110.pbx* and *PBSoapClient110.pbx* can also throw SoapException objects when, for example, connection to the server fails, or the Web service cannot be located or created.

Catching exceptions

A client application can handle communications errors in a number of ways. For example, if a client connects to a server and tries to invoke a method for an object that does not exist, the client can disconnect from the server, connect to a different server, or retry the operation. Alternatively, the client can display a message to the user and give the user the opportunity to control what happens next.

When an error occurs, if the client connects to a new server to retry the operation, it must instantiate the remote object on the new server before invoking a method of the remote object.

Unhandled exceptions

If no exception handler exists, or if the existing exception handlers do not handle the exception, the SystemError event on the Application object is executed. If the SystemError event has no script, an application error occurs and the application is terminated.

Using the UDDI Inquiry API

The UDDIProxy PowerBuilder extension class enables you to search UDDI registries for a Web service that you want to access. For a description of this extension class and its methods, see the *PowerBuilder Extension Reference* or the online Help.

Example code

The following is example code using all the methods in the UDDIProxy class. It searches an IBM UDDI registry by service name (Weather) and business name (IBM), using the same search options (case sensitivity and a maximum of 5 rows returned):

```
uddiproxy proxy
int ret
proxy = create uddiproxy
ret = proxy.setinquiryurl
  ("http:/www-3.ibm.com/services/uddi/inquiryapi")
ret = proxy.setoption (false, true, 0, 5)
int count, count2
```

```
string businessName[], businessDescription[]
string businessKey []
string servicename[], servicedescription[]
string servicekey [], wsdl [ ]
ret = proxy.findService(“Weather”,count,serviceName, &
  serviceDescription, serviceKey, businessName, wsdl)
int i, j
FOR i = 1 TO count
   messagebox(servicename[i], &
     servicedescription[i]+servicekey[i]+wsdl[i])
NEXT

proxy.findbusiness(“IBM”, count, businessName, &
  businessDescription, businessKey)
FOR i = 1 TO count
   messagebox(businessName[i], &
     businessDescription[i] + businessKey[i])
  proxy.getbusinessdetail (businessKey [i], count2, &
     servicename, servicedescription, servicekey, wsdl)
  FOR j = 1 TO count2
   messagebox(servicename[j], &
     servicedescription[j]+servicekey[j]+wsdl[j])
  NEXT
NEXT
destroy proxy
```

Troubleshooting UDDI API calls

You can turn on logging to track down any failures on method calls to the UDDIProxy object. The PowerBuilder Java service class path must include the *log4j.properties* configuration file to turn on logging. The following is an example of a log configuration file for a UDDI search:

```
#log4j.debug=true
#log all level
#log4j.rootCategory=DEBUG, lf5
#only log com.sybase.powerbuilder.uddi
log4j.category.com.sybase.powerbuilder.uddi=DEBUG,
    dest2, lf5
#dest1
#log4j.appender.dest1=org.apache.log4j.ConsoleAppender
#log4j.appender.dest1.layout=
    org.apache.log4j.PatternLayout
#log4j.appender.dest1.layout.ConversionPattern=
    %-5p: %-5r: %-5c: %l: %m%n
#dest2
log4j.appender.dest2=org.apache.log4j.FileAppender
log4j.appender.dest2.layout=
    org.apache.log4j.PatternLayout
```

```
log4j.appender.dest2.layout.ConversionPattern=
    %-5p: %l: %m%n
log4j.appender.dest2.File=c:/mylog.txt
#lf5
log4j.appender.lf5=
    org.apache.log4j.RollingFileAppender
log4j.appender.lf5.File=c:/mylog.lf5
log4j.appender.lf5.layout=
    org.apache.log4j.PatternLayout
log4j.appender.lf5.layout.ConversionPattern=
    [slf5s.start]%d{DATE}[slf5s.DATE]%n\
    %p[slf5s.PRIORITY]%n%x[slf5s.NDC]
    %n%t[slf5s.THREAD]%n\%c[slf5s.CATEGORY]
    %n%l[slf5s.LOCATION]%n%m[slf5s.MESSAGE]%n%n
log4j.appender.lf5.MaxFileSize=500KB
```

CHAPTER 32 Using the DataWindow Plug-in

About this chapter

This chapter describes how to plan and deploy a Powersoft report (PSR) in a Web page using the DataWindow plug-in.

Contents

Before you begin

This chapter assumes that you are familiar with HTML and URLs and how a Web browser obtains pages from a Web server, and that you have access to a Web server.

About the DataWindow plug-in

The DataWindow plug-in lets you display a Powersoft report (PSR) on a Web page viewed in a browser that supports Netscape plug-ins.

A PSR file contains a report definition (source and object) as well as the data contained in the report when the PSR file was created. Because its data is saved with it, a PSR file does not require a database connection, but the data is static and cannot be refreshed.

Supported browsers

The DataWindow plug-in requires use of a Web browser that supports Netscape plug-ins, such as the Netscape browser and Microsoft Internet Explorer versions 3 through 5.5 Service Pack 1.

Microsoft Internet Explorer 5.5 Service Pack 2 and later versions do not support Netscape plug-ins. If you are using the Enterprise or Professional edition of PowerBuilder, you can use the DataWindow Web control for ActiveX to display a PSR in Internet Explorer. For more information, see the *DataWindow Programmer’s Guide*.

Security

There are no security issues for the DataWindow plug-in. It does not run any local applications and it does not write local files unless the user explicitly chooses to save the report locally.

The DataWindow plug-in displays PSR files only. PSR files are read-only, so there is no need for a secure version of the DataWindow plug-in.

Restrictions

The PSR *cannot* use the RichText presentation style.

How the DataWindow plug-in works

The DataWindow plug-in displays a PSR with its data and formatting as it was previewed and saved from the Report painter, DataWindow painter, DataWindow control, or DataStore. The PSR displays all the formatting in the DataWindow or report object, including rotated text, colored text, shading, or edit styles like check boxes or radio buttons.

The data is a report—there is no data entry or database access.

A pop-up menu in the Web browser lets the client print the report or save it in several formats. If clients save the report as a local PSR file, they can view it in InfoMaker, which has more tools for searching, filtering, and sorting the data.

The DataWindow plug-in implements the Netscape plug-in API and requires a browser that supports this API (for information, see “Supported browsers” on page 641).

Client-server interactions, in detail

Table 32-1 describes in detail what happens between the client and server when the user views an HTML document containing a DataWindow plug-in.

Table 32-1: Client-server transactions for the DataWindow plug-in

Step	What the client does	How the server responds
1	The Web browser requests the HTML document from the server	The server sends a header identifying the document’s MIME type (text/html)
2	The browser receives the MIME type and prepares to receive an HTML document	The server sends the HTML document
3	The browser receives the HTML document and displays it	—
4	The browser recognizes the Embed element, reserves space for the plug-in on the page, and requests the PSR file from the server	The server sends a header identifying the PSR’s MIME type (application/data window)

Step	What the client does	How the server responds
5	The browser receives the MIME type and prepares to receive a PSR file	The server sends the PSR file
6	The Web browser receives the PSR file	—
7	The browser searches its Plugins directory for the DLL that corresponds to the MIME type (application/data window)	—
8	The Web browser loads the plug-in DLL and displays the PSR file	—

Requirements

Each client that will browse pages containing DataWindow plug-ins needs the DataWindow plug-in DLL installed on the local machine.

Name and location

The name of the DataWindow plug-in DLL is *NPDWE110.DLL*. PowerBuilder installs it in the *PowerBuilder 11.0\Internet Tools\Plugins* directory.

Installing and configuring the DataWindow plug-in

The DataWindow plug-in is not installed on your computer if you select the Typical setup type when you install PowerBuilder. You can install the plug-in by running a custom installation of PowerBuilder and checking the Web Plugins box on the Select Components page. Click the Change button on that page to select the plug-ins and controls you want to install. This section describes setup tasks that you may need to perform *after* installing the plug-in.

Installation location

The DataWindow plug-in is installed in the *PowerBuilder 11.0\Internet Tools\Plugins* directory. The PowerBuilder installation program might also install a copy of the plug-in in your Web browser's *Plugins* directory if you have Netscape installed.

If you have no browser installed or the installation program did not find the browser, you need to install the Netscape Web browser and then copy or move the plug-ins to your browser's *Plugins* directory. Current versions of Microsoft Internet Explorer do not support plug-ins.

Developing and deploying a DataWindow plug-in

What you do

There are four main tasks involved in displaying a PSR in the DataWindow plug-in.

❖ **To display a PSR in the DataWindow plug-in:**

1. Save a PSR file.
2. Create an HTML page that embeds the DataWindow plug-in.
3. Configure the Web server by registering the appropriate content type (MIME type) and copying the HTML page and PSR files to appropriate directories.
4. On all client workstations, install the DataWindow plug-in DLL.

Resulting components

When everything is set up, the various computers will have the components listed in Table 32-2.

Table 32-2: Server and client configuration for DataWindow plug-in

Computer	Component
Server	MIME type application/data window registered for the file extension PSR
	HTML page with an Embed element for the PSR
	PSR file
Client	DataWindow plug-in DLL in the browser's *Plugins* directory and supporting DLLs

What's next

The rest of this chapter describes the four steps for displaying a PSR in the DataWindow plug-in:

- Saving a Powersoft report (PSR)
- Creating an HTML page
- Setting up the Web server
- Setting up users' workstations

Saving a Powersoft report (PSR)

Creating a PSR file

There are several ways in which you or a user can create a PSR file.

❖ **To create a PSR file:**

- In the PowerBuilder DataWindow painter or InfoMaker Report painter, do one of the following:
  - Select File>Save As File. In the Select a File Name dialog, specify a location for the file, a file name, and Powersoft Report as the file type.
  - Select File>Save Rows As. In the Save As dialog box, specify a location for the file, a file name, and Powersoft Report as the type.

Resources for the PSR

A DataWindow or report object can display bitmap objects and custom pointers. These external resources must be made available on the client workstation. They are not automatically downloaded from the server.

Paths for external resources must be valid on the client workstation. In the DataWindow or Report painter, you can edit paths so that they specify directories relative to the current directory instead of absolute paths that are unlikely to exist on the user's computer.

In an intranet environment, you can store external resources on a network drive that is accessible to all users. In Windows environments, you can use a network path to name the network drive instead of using a mapped drive letter.

OLE objects and custom controls

OLE servers and custom controls must be installed and registered on each client workstation.

Restrictions

You *cannot* use a PSR file in the DataWindow plug-in whose original DataWindow or report had the RichText presentation style.

What's next

After saving a PSR file, you need to create the HTML page that displays it.

Creating an HTML page

You include a Powersoft report (PSR) on a Web page with the Embed element. Element attributes specify the space allocated for the report and the report file name.

A sample Embed element might look like this:

```
<EMBED src=April_sales.psr WIDTH=370 HEIGHT=320>
```

Attributes of the Embed element

The Embed element is part of the HTML specification for plug-ins. For the DataWindow plug-in, you specify only standard HTML attributes.

HTML attributes HTML attributes name the file to be downloaded to the client and the space reserved for the plug-in on the Web page.

Table 32-3: HTML attributes for the DataWindow plug-in

HTML attribute	Value
SRC	A URL identifying the object to be downloaded. When the browser processes the Embed element, it requests the resource from the server and finds the DLL in its Plugins directory that handles the content type. For a DataWindow plug-in, the object is a PSR file.
WIDTH	The width of the viewing window in pixels.
HEIGHT	The height of the viewing window in pixels.

Sample page

Here is HTML code that includes the DataWindow plug-in in a sample page. Note the use of the Embed element to specify a PSR file.

Opening element

```
<HTML>
```

Document heading

```
<HEAD>
<TITLE>On Leave report</TITLE>
</HEAD>
<BODY>
```

H1 heading

```
<H1>On Leave report</H1>
```

Embed element in a paragraph

```
<P><EMBED src=DWB_att_crossout.psr WIDTH=680 HEIGHT =
   350></P>
```

Closing elements

```
</BODY>
</HTML>
```

Setting up the Web server

Setting up the Web server involves:

1 Specifying the MIME type

2 Putting the files on the server

Specifying the MIME type

Using the appropriate software for your Web server, register the MIME type for the DataWindow plug-in. For PowerBuilder 11.0, the MIME type is:

```
application/datawindow11.0
```

The file extension associated with the MIME type is PSR.

Your server documentation might use the term content type instead of MIME type.

Putting the files on the server

You must copy the PSR and HTML files to appropriate directories on your server.

Table 32-4: PSR and HTML file locations on server

Files	Location	What you do
HTML page	The HTML documents directory or subdirectory	Copy the HTML page to the directory specified in any URLs that link to the page.
PSR file named in the SRC attribute of the Embed element	The HTML documents directory or some other directory as appropriate	Copy the PSR file to the directory you specified in the Embed element attribute.

About URLs The URLs you specify in your HTML page are logical paths as defined by the Web server. For example, the system path for your PSR might be:

```
C:\WEBSITE\HTDOCS\PB\DWB_ATT_CROSSOUT.PSR
```

If htdocs is defined as the server’s documents directory, the logical path for the URL would be relative to the documents directory:

```
pb/dwb_att_crossout.psr
```

Setting up users’ workstations

After saving the PSR, creating an HTML page, and setting up the server, you need to make sure client workstations are set up to view the page containing the DataWindow plug-in.

Users who will view a Web page that includes a DataWindow plug-in need supporting software installed on their client workstations. They also need a connection to the Web server.

Required components

Each client workstation needs the components in Table 32-5 to view a Web page containing a DataWindow plug-in.

Table 32-5: Client requirements to view DataWindow plug-in

Component	Procedure
An Internet or intranet connection	Available within your corporation or from an Internet service provider.
A Web browser that supports Netscape plug-ins	Available from the browser vendor. Examples are: • Netscape Navigator Version 3.x or later. • Microsoft Internet Explorer Version 3.x through 5.5 Service Pack 1. Internet Explorer 5.5 Service Pack 2 and later versions do not support plug-ins.
DataWindow plug-in DLL	If it is not there already, copy the *NPDWE110.DLL* file from the *Internet Tools\Plugins* directory to the browser's *Plugins* directory.
Supporting DLLs	If it is not there already, copy the *PBSHR110.DLL* file from the *Shared\PowerBuilder* directory to the browser's *Plugins* directory. If they are not already available on the client workstation, install the following Microsoft runtime files in the client's Windows system directory: *MSVCR71.DLL*, *MSVCP71.DLL*, and *ATL71.DLL*.
Other files	If the original DataWindow object or report included bitmap objects or custom pointers from external files, you need to copy those files to the client workstation. The path on the client workstation must correspond to the path saved in the PSR. If the original included OLE objects or custom controls, you must install and register the OLE server or custom control on the client workstation. The path for an OLE object must be valid on the client.

Viewing the Web page and the PSR

While viewing a PSR in the DataWindow plug-in, the user can:

- Navigate through the data in the PSR using scroll bars
- Change values in rows and columns if the source of the PSR was an editable DataWindow object (but because there is no database connection, data cannot be updated in a database)
- Right-click on the PSR to display a pop-up menu for saving, printing, and navigating

InfoMaker users who save the report from the Web page as a local PSR file can open that PSR file in InfoMaker to search, filter, and sort the data.

CHAPTER 33 Using the PowerBuilder Window Plug-in

About this chapter

This chapter describes how to develop, test, and deploy a PowerBuilder application that is displayed as a plug-in application in a Web page.

Contents

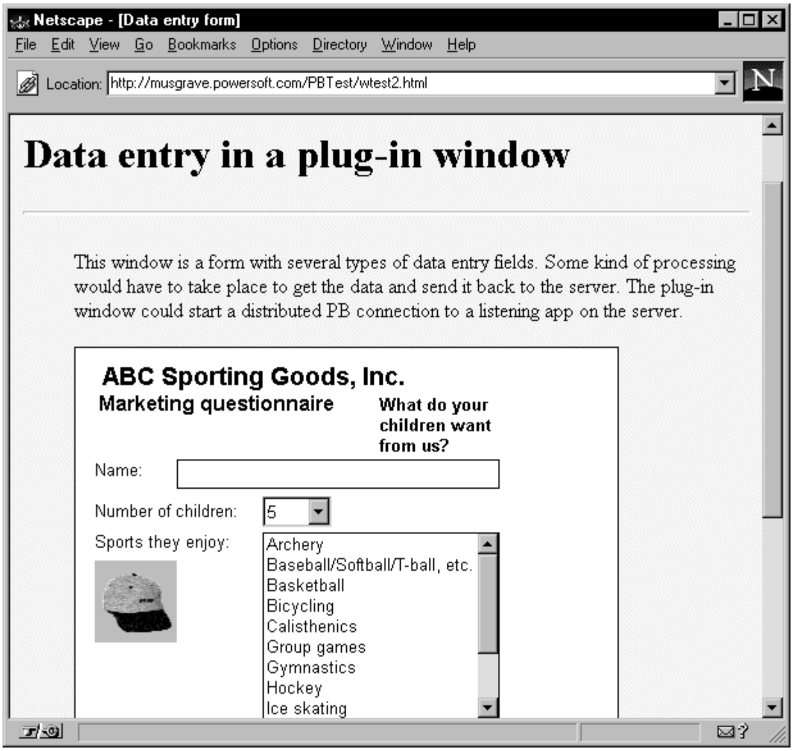
Before you begin

This chapter assumes you are familiar with HTML and URLs and how a Web browser obtains pages from a Web server, and that you have access to a Web server.

About the PowerBuilder window plug-in

The PowerBuilder window plug-in lets you display a PowerBuilder child window on a Web page viewed in a browser that supports Netscape plug-ins.

Internet Explorer
Microsoft Internet Explorer 5.5 Service Pack 2 and later versions do not support Netscape plug-ins. As a result, the PowerBuilder window plug-in cannot be used to display PSR reports on those browsers.



Netscape - [Data entry form]
File Edit View Go Bookmarks Options Directory Window Help
Location: http://musgrave.powersoft.com/PBTest/wtest2.html

Data entry in a plug-in window

This window is a form with several types of data entry fields. Some kind of processing would have to take place to get the data and send it back to the server. The plug-in window could start a distributed PB connection to a listening app on the server.

ABC Sporting Goods, Inc.

Marketing questionnaire

What do your children want from us?

Name:

Number of children: 5

Sports they enjoy:

Archery
Baseball/Softball/T-ball, etc.
Basketball
Bicycling
Calisthenics
Group games
Gymnastics
Hockey
Ice skating

Features

The PowerBuilder child window can include all the familiar controls, including DataWindows, OLE objects, ActiveX (OCX) controls, and tree controls. You can also open other (pop-up or response) windows from the child window.

As the user interacts with controls in the child and other windows, scripts for the controls' events are executed just as they are in standalone PowerBuilder applications. Database access by the plug-in application occurs locally using the client's defined database connections.

The objects in the application can be contained in one or more PowerBuilder Dynamic Libraries (PBDs).

Standard and secure versions

There are two versions of the PowerBuilder window plug-in: standard and secure.

Standard PowerBuilder window plug-in The standard PowerBuilder window plug-in displays a PowerBuilder child window in an HTML page. The standard window plug-in is implemented by *NPPBA110.DLL*.

Secure PowerBuilder window plug-in The secure PowerBuilder window plug-in is a secure version of the standard window plug-in. Using the secure version ensures that PowerBuilder applications downloaded over the Internet do not damage a client system or access information on a client workstation. The secure window plug-in is implemented by *NPPBS110.DLL*.

For considerations to keep in mind when using the secure version of the PowerBuilder window plug-in, see "Using the secure PowerBuilder window plug-in" on page 655.

Supported browsers

The PowerBuilder window plug-in requires use of a browser that supports Netscape plug-ins, such as Netscape Navigator Version 3.x or later. Microsoft Internet Explorer 5.5 Service Pack 2 and later versions do not support Netscape plug-ins.

Security

The standard PowerBuilder window plug-in is a nonsecure application, meaning that it can access local files and run local applications. These types of processing are often undesirable in an uncontrolled Web environment but might be perfectly acceptable on a corporate intranet where access is controlled.

If security is important at your site, consider using the secure version of the PowerBuilder window plug-in to build your application.

Some events always execute in secure mode
As of PowerBuilder 7, the application Open event and some of the Constructor events for controls execute in secure mode even when you use the standard window plug-in. If your application logic depends on the ability to perform a nonsecure activity in these events, such as opening a file for writing, post an event message to yourself. These are processed once the plug-in returns to standard mode.

For more about the secure PowerBuilder window plug-in, see "Using the secure PowerBuilder window plug-in" on page 655.

What kinds of applications make good plug-ins?

HTML forms provide some user interaction by means of a limited user interface. The PowerBuilder window plug-in takes you beyond HTML. You can present a fully developed application window with a rich user interface design to a Web page. You can access data sources defined on the client workstation.

Examples

Your application might be:

- A data analysis window with master and detail DataWindows
- A user interface design that lets the user make choices using TreeView, ListView, and PictureListBox controls
- A data entry form that processes data on a server
- A data entry form that uses the client's defined database connection (either network or local) to process the data on the client machine
- A window that uses PowerBuilder's OLEControl control to display an ActiveX control

How the PowerBuilder window plug-in works

The PowerBuilder window plug-in displays a PowerBuilder child window inside a fixed space reserved on the Web page. The user can interact with the controls on the page, and the PowerBuilder scripts for the window and its controls can execute any PowerBuilder code. When the user switches to another Web page, the PowerBuilder window is closed and the PowerBuilder DLLs are unloaded from memory.

You include the plug-in on an HTML page using the HTML Embed element. The Embed element names one or more PBDs that contain PowerBuilder objects and the name of the child window object that is displayed on the page.

The PowerBuilder window plug-in implements the Netscape plug-in API and requires a browser that supports this API (for information, see "Supported browsers" on page 651.)

Client-server interactions, in detail

Table 33-1 describes in detail what happens between the client and server when the user views an HTML document with a PowerBuilder window plug-in:

Table 33-1: Client/server interaction for the Window plug-in

Step	What the client does	How the server responds
1	The Web browser requests the HTML document from the server	The server sends a header identifying the document's MIME type (text/html)
2	The browser receives the MIME type and prepares to receive an HTML document	The server sends the HTML document
3	The browser receives the HTML document and displays it	—

Step	What the client does	How the server responds
4	The browser recognizes the Embed element, reserves space for the plug-in on the page, and requests the PBD file from the server	The server sends a header identifying the PBD's MIME type, which can be one of the following: • **Standard window plug-in** application/vnd.powerbuilder11.0 • **Secure window plug-in** application/vnd.powerbuilder11.0-s
5	The browser receives the MIME type and prepares to receive a PBD file	The server sends the PBD file
6	The browser receives the PBD file	—
7	The browser searches its Plugins directory for the DLL that corresponds to the MIME type (see server step 4)	—
8	The browser loads the plug-in DLL	—
9	The plug-in looks for and loads the PowerBuilder deployment DLLs	—
10	If the Embed element includes LIBRARY attributes, then the client requests the specified PBD files	The server sends a header identifying the PBD's MIME type (see server step 4)
11	The browser receives the MIME type and prepares to receive additional PBD files	The server sends the PBD file
12	PowerBuilder displays the child window specified in the Embed element	—
13	The child window executes its Open script	—
14	If a script calls the CommandParm function, PowerBuilder queries the browser for the value of the COMMANDPARM attribute in the Embed element	—

Requirements

The PowerBuilder window plug-in uses the PowerBuilder runtime DLLs as well as the plug-in DLL to provide a full range of PowerBuilder functionality.

Each client that will browse pages containing PowerBuilder window plug-ins needs supporting software installed on the local machine:

- PowerBuilder runtime DLLs
- Window plug-in DLL, placed in the Web browser's *Plugins* directory

The PowerBuilder window plug-in is especially useful in an intranet application where you have control over the setup of client machines.

For details about setting up client machines on each supported platform, see "Setting up users' workstations" on page 668.

Name

The name of the PowerBuilder window plug-in DLL depends on whether you are using the standard or secure version:

Table 33-2: PowerBuilder window plug-in DLLs

Component	**Name**
Standard PowerBuilder window plug-in	*NPPBA110.DLL*
Secure PowerBuilder window plug-in	*NPPBS110.DLL*

Installing and configuring the PowerBuilder window plug-ins

The PowerBuilder window plug-ins are not installed on your computer if you select the Typical setup type when you install PowerBuilder. You can install the plug-ins by running a custom installation of PowerBuilder and checking the Web Plugins box on the Select Components page. Click the Change button on that page to select the plug-ins and controls you want to install.

This section describes setup tasks that you may need to perform *after* installing the plug-ins.

Installation location

The PowerBuilder window plug-in DLLs are installed in the *PowerBuilder 11.0\Internet Tools\Plugins* directory. The PowerBuilder installation program may also install a copy of the plug-ins in your Web browser's *Plugins* directory if you have Netscape installed.

If you have no browser installed or the installation program did not find the browser, you need to install Netscape and then copy or move the plug-ins to the browser's *Plugins* directory.

PowerBuilder deployment DLLs

The PowerBuilder window plug-in must know the location of the PowerBuilder runtime DLLs (*PBVM110.DLL*, *PBDWE110.DLL* and so forth). If the Setup program finds your browser, it will make the appropriate registry modifications, but if it cannot find your browser, you must make the modifications. You can do this in one of two ways:

- Add the directory for the PowerBuilder runtime DLLs to the application path key for the browser's executable in the Windows registry
- Add the directory for the PowerBuilder runtime DLLs to the system path

Using the secure PowerBuilder window plug-in

How to use it

You follow the same basic steps to develop and deploy applications using the secure PowerBuilder window plug-in as you do when using the standard PowerBuilder window plug-in. (These steps are described in "Developing and deploying a PowerBuilder window plug-in application" on page 656.)

The only difference is that the secure PowerBuilder window plug-in uses a special version of the standard window plug-in DLL. The secure version of the window plug-in is *NPPBS110.DLL*.

Restricted functionality

Using the secure PowerBuilder window plug-in severely restricts the PowerBuilder application running on the client workstation, denying access to the client system except for printing. For this reason, using the secure window plug-in might not be appropriate in all situations.

The types of activities listed in Table 33-3 are restricted when you use the secure PowerBuilder window plug-in.

Table 33-3: Secure window plug-in restrictions

Activity	Restriction
External functions	Calling an external function causes an execution error
Certain PowerScript functions	Calling a restricted PowerScript function causes an execution error
Database connection	Calling PowerScript functions that result in database access causes an execution error
Internet access	Applications that use the secure PowerBuilder window plug-in can establish an Internet connection only to the current Web server
E-mail	Calling PowerScript Mail functions causes an execution error

Activity	Restriction
OLE	Calling PowerScript OLE functions causes an execution error
Dynamic Data Exchange (DDE)	Calling PowerScript DDE functions causes an execution error

Developing and deploying a PowerBuilder window plug-in application

Basic steps

There are four main tasks involved in developing a plug-in application.

❖ **To create and deploy a PowerBuilder window plug-in application:**

1. Create, test, and build the PowerBuilder application.
2. Create an HTML page that embeds your PowerBuilder application window.
3. Configure the Web server by registering the appropriate content type (MIME type) and copying the HTML page and PBD files for the application to appropriate directories.
4. On all client workstations, install the standard or secure PowerBuilder window plug-in DLL and the PowerBuilder deployment DLLs.

Resulting components

When everything is set up, the various computers have the components in Table 33-4.

Table 33-4: Server and client configuration for PowerBuilder window plug-in

Computer	Component
Server	MIME type registered. For instructions, see “Specifying the MIME type” on page 667.
	HTML page with an Embed element for the PBD.
	One or more PBDs containing your Application objects.
Client	PowerBuilder window plug-in DLL in the browser’s Plugins directory.
	PowerBuilder deployment DLLs installed using the Deployment Kit for your platform.
	Directory for the PowerBuilder deployment DLLs listed in the system path.
	Any additional software required by the plug-in application, such as database connection software or OLE servers and custom controls.

What’s next

The rest of this chapter describes the four steps for creating and deploying a PowerBuilder window plug-in application:

- Creating the PowerBuilder application
- Creating an HTML page
- Setting up the server
- Setting up users’ workstations

Creating the PowerBuilder application

The starting point of your PowerBuilder window plug-in application is a child window displayed in a Web page. In the window you can include controls and write scripts for events. Your scripts can open other windows, read and write files, and run other programs on the client machine.

Design choices for plug-in applications

The application you design for use as a plug-in can be much the same as other PowerBuilder applications you develop, but there are some restrictions and considerations to keep in mind in the following areas:

- Window management
- Objects
- Scripts and variables
- Data access
- External files

Window management

Initial child window Your initial window needs to be a child window that lives in the browser frame.

You *can*:

- Include a title bar on the child window (but you *should not* use a control menu, maximize box, or minimize box on that window)
- Open pop-up and response windows from the child window (but not main or MDI windows)

You *cannot*:

- Have a menu for the child window
- Open another child window from the initial window

Closing windows When the client browses to another Web page, the child window on the current Web page is closed, but other windows remain open unless your application closes them. You must close them in the child window's Close or CloseQuery events.

Do not try to stop the child window from being closed in the CloseQuery event by setting a return value. You cannot prevent the browser from changing to another page and removing the window from view. If the user returns to the page, another instance of your application is started.

Objects

Objects in your PBDs Your plug-in application has access to all objects in the PBDs. This includes functions, structures, and user objects.

System objects Your plug-in application has access to system objects that PowerBuilder instantiates, such as the SQLCA Transaction object and the Message object.

Application object You can use the optional APPLICATION attribute of the Embed element to specify the name of your PBD's Application object. This gives your plug-in application access to the Application object's Open and Close events.

If you do *not* specify the APPLICATION attribute:

- Your plug-in application does not have access to the Application object, and thus events like SystemError and Idle are not available. You cannot treat variables and functions defined in an Application object as global.
- Any scripts in the Application object are used only during testing within PowerBuilder. You cannot do any application setup in the Application object's scripts.

For information about specifying the APPLICATION attribute, see "Attributes of the Embed element" on page 664.

Scripts and variables

Global variables If you use the APPLICATION attribute of the Embed element to specify your PBD's Application object, your plug-in application has access to global variables and global functions used in the application.

If you do *not* specify the APPLICATION attribute, the plug-in application cannot use global variables. You must define all variables as instance, shared, or local.

For information about specifying the APPLICATION attribute, see "Attributes of the Embed element" on page 664.

Referencing the initial window You cannot reference the initial child window by name in your scripts, because PowerBuilder does not create a variable to hold the instance of it. (By contrast, when you instantiate a window yourself by coding the Open function, you always place the instance in a variable that you can then reference.)

Thus, the following code produces an error at runtime (because the window variable w_mychild does not exist):

```
// This code produces a runtime error:
w_mychild.title = "The initial window"
```

But you *can* code:

```
// In the child window itself:
this.title = "The initial window"
// or
title = "The initial window"
```

```
// In controls of the child window:
parent.title = "The initial window"
```

Scripts for application setup All application setup must be done in the child window, including connecting to the DBMS. The first scriptable events to occur are the constructors for the controls in the window. Then the Open event for the window occurs.

The Activate event does not occur for a child window; do not put application setup code there.

Data access

If your application accesses a DBMS, each client must have a connection to the data source. The connection must be defined on the client's machine, not the server. The data source might be a local or a network DBMS.

For information about how to connect to a DBMS from the client machine, see *Connecting to Your Database*.

The constructor events for controls occur before the Open event of the window. If you connect to the DBMS in the window's Open event, the constructor events cannot get data. You can get data for controls in the window's Open event, or you can post events from the constructor events or the window's Open event.

Paths for external resources

Paths you specify for files must be valid on the client workstation.

If your application uses images as external files, the images must be available at the path specified in the PowerBuilder object. Instead of using external files, you can build image resources into PBDs, as described in "Building the dynamic libraries" on page 662.

If your application reads or writes local files, the path for those files must be valid on each client's machine.

If a path refers to a network drive by mapped drive letter, all clients must use the same drive letter that the application uses. As an alternative, specify the server name in the path instead.

For example, both of these paths are valid when *o:* is mapped to *\\marketing\drive*, but the second path, which uses a server name, is more likely to *remain* valid.

```
O:\pbapps\connect.bmp
\\marketing\drive\pbapps\connect.bmp
```

Defining the starting window in the Window painter

You use the Window painter to create the starting window, as you do for any window you create in PowerBuilder.

❖ **To create your application's starting (or only) window:**

1. Create a new window object in PowerBuilder.
2. On the window's General tab of the Properties view, set the window type to child.
3. Add other controls as needed.
4. Write scripts for events of the window and controls.

❖ **To convert an existing application to run as a plug-in:**

1. Change the type of the opening window to child.
2. If you do *not* plan to specify the APPLICATION attribute of the Embed element, do both of the following:

    - Remove references to global variables.
    - Move application setup code from the application's Open event or MDI frame events to the child window's Open event.

    For information about specifying the APPLICATION attribute, see "Attributes of the Embed element" on page 664.

3. Depending on the application's design, you may need to redesign how it opens other windows.

About child windows

Child windows cannot have menus. They are never considered the active window—and therefore the Activate event is never triggered. They can have title bars and can be minimized, maximized, and resized. In the plug-in environment, the child window is always restricted to the space allotted by the WIDTH and HEIGHT attributes specified on the Web page, as follows:

- Maximizing causes the window to fill the space allotted by the WIDTH and HEIGHT attributes.
- Minimizing displays the window's icon and title at the bottom of the plug-in's allotted space.
- If the child window is resizable, the user can drag the borders to make the window smaller (but not larger) than the allotted space.

As a result, it is not useful to allow minimizing, maximizing, or resizing of the child window.

Testing the application in PowerBuilder

Before you try your application in a client browser, you can test it in PowerBuilder by defining a main window that opens the child window.

❖ **To test your PowerBuilder window plug-in application:**

1. Create a new window object whose type is main (the default).
2. Write a script for the Open event that opens the plug-in's starting child window:

   ```
   Open(w_child)
   ```

3. For convenience, you can set appropriate window sizes in the Window painter:
   - Make the main window large enough to display the child
   - Position the child window in the upper-left corner (use the Position fields in the Other tab of the Properties view)
4. Run the test window by clicking the Preview button or selecting File>Run/Preview.

Debugging in PowerBuilder
To use the PowerBuilder debugger, you need to run an application instead of a single window. Define an Application object with a script that opens the main window. Then you can use the Run or Debug command to test the application.

Building the dynamic libraries

The *User's Guide* describes how to build dynamic runtime libraries (PBDs). The procedure is the same for a plug-in application. This section highlights the choices you need to make for building a PBD for a plug-in application.

Remember that in the Web environment, file size is important.

Organizing objects in PBLs

Before you build your application, you should use the System Tree or Library painter to organize the objects your application uses in PBLs, which are the sources for the plug-in application's PBDs. The following suggestions can help you optimize the resulting libraries:

- To minimize file size, include only objects the application uses; remove any objects that are not needed.

- Include any objects that are dynamically created, such as DataWindow objects used in DataStores or assigned dynamically to DataWindow controls.
- Include ancestor objects.

Using PowerBuilder resource (PBR) files

Several controls can use external files for images. These include PictureListBox, TreeView, Picture, and PictureButton controls, pointers, and bitmap objects in DataWindow objects. If your application uses external files for the images, it is unlikely that the client has the same images on the same system path.

Instead of finding some way to install the pictures on client machines, you can use one or more PowerBuilder resource (PBR) files so that the images are built into the PBDs. The resulting PBDs are larger but self contained.

Because you are building a dynamic runtime library, not an executable, you do not need to include DataWindow objects in the PBR file. All PowerBuilder objects in the source PBL are included in the resulting PBD.

Images and other resources on a network
Instead of building the files into the PBD, you can put the files in a generally accessible network directory, but the path to the files must be identical to the path named in the PowerBuilder objects. This means that in the Windows environment, each client must use the same drive letter to map the network drive, or you can specify the server name in the path.

❖ **To define a PBR file:**

1 Open the PowerBuilder File Editor (Shift+F6) or some other text editor and create a new file with the extension PBR. (You can add the File Editor icon to the toolbar.)

2 List each image or other resource on its own line. List the path and file name exactly as it is named in the object property sheet or script.

Shortcut
Look at the object’s Properties view and use Ctrl+C to copy the file name to the clipboard, then paste it into the editor.

3 Save the file.

If your application includes several PBLs, each with objects using their own resources, you should create a PBR file for each PBL. The PBR file will list file names for resources that are used in one PBL.

Building the PBDs

Build your runtime libraries (PBDs) in the Project or Library painter as described in the *User's Guide*. Keep in mind:

- **Deselect Machine code.** Your plug-in application must have PBDs, not machine code DLLs.
- **Specify a PBR file for each PBD.** If the objects in the PBL use resources that you have listed in a PBR file, put the PBR name in the Resource File Name text box.

For instructions on defining a PBR file and building your runtime libraries (PBDs), see the chapter on creating an executable in the *User's Guide*.

Creating an HTML page

After creating and building the PowerBuilder PBDs for your plug-in application, you need to create the HTML page that displays it.

You include a PowerBuilder window on a Web page with the Embed element. Element attributes specify the space allocated for the window, the name of the PBD, and the name of the child window in the PBD.

A sample Embed element might look like this:

```
<EMBED SRC=plugin_tree.pbd WIDTH=370 HEIGHT=320
WINDOW=w_emp_by_dept>
```

Attributes of the Embed element

The Embed element is part of the HTML specification for plug-ins. It defines several standard attributes, and PowerBuilder defines additional attributes.

HTML attributes

HTML attributes name the file to be downloaded to the client and the space reserved for the plug-in on the Web page.

Table 33-5: HTML Embed element attributes

HTML attribute	Value
SRC	A URL identifying the object to be downloaded. When the browser processes the Embed element, it requests the resource from the server and finds the DLL that handles the content type in its Plugins directory. For a PowerBuilder window plug-in, the object is a PBD containing the child window that starts the application.
WIDTH	The width of the viewing window in pixels.
HEIGHT	The height of the viewing window in pixels.

The WIDTH and HEIGHT attributes define the maximum width and height of the child window. If the child window is resizable, the user can make it smaller than the specified size, but not larger.

PowerBuilder attributes

The PowerBuilder attributes for the Embed element let you identify the window object that starts your application, additional libraries, parameters to pass to the application, and the name of your Application object.

Table 33-6: PowerBuilder Embed element attributes

PowerBuilder attribute	Value
WINDOW	The class name of the child window in the PBD.
LIBRARY (optional)	A URL specifying an additional PBD that contains objects that your application needs. You must specify the *full URL* for the PBD, rather than a relative one. You can specify more than one LIBRARY attribute. Specify a LIBRARY attribute for each additional PBD your application needs. Do not specify a LIBRARY attribute for the file specified for SRC. For an example of HTML code that uses the LIBRARY attribute, see “Embed element with additional attributes” on page 667.

PowerBuilder attribute	Value
COMMANDPARM (optional)	A string that you want to pass to your window. To access this string from within your window, call the CommandParm function. For an example of HTML code that uses the COMMANDPARM attribute, see “Embed element with additional attributes” on page 667.
APPLICATION (optional)	The name of the Application object in the PBD. This gives your plug-in application access to the Application object’s Open and Close events, as well as to global variables and global functions used in the application. When you use the APPLICATION attribute, the Application object’s Open and Close events execute by default, and cannot be overridden. Make sure the Application object’s Open event does *not* open the child window specified by the WINDOW attribute. If it does, your application will fail. For an example of HTML code that uses the APPLICATION attribute, see “Embed element with additional attributes” on page 667.

Sample page

Here is HTML code that includes the PowerBuilder window plug-in showing a window with master and detail DataWindow controls. Note the use of the Embed element to specify the PowerBuilder window plug-in.

Opening element

```
<HTML>
```

Document head

```
<HEAD>
<TITLE>Master-Detail window</TITLE>
</HEAD>
<BODY>
```

A small image

```
<IMG SRC="undercon.gif" BORDER=0 HEIGHT=38 WIDTH=40>
```

H1 heading

```
<H1>Master-detail in a PB window plug-in</H1>
```

Horizontal rule

```
<P><HR></P>
```

Paragraph

```
<P>This window accesses the PB demo database to display
a list of departments. When the user clicks a department
row, the second DataWindow displays the employees in
that department.</P>
```

Embed element in a paragraph

```
<P><EMBED SRC=plugin_tree.pbd WIDTH=370 HEIGHT=320
WINDOW=w_emp_by_dept>
</P>
```

Link to site's home page

```
<LI>Back to <A
HREF="http://www.mycompany.com/index.html">Home
Page</A> </LI>
```

Closing elements

```
</BODY>
</HTML>
```

Embed element with additional attributes

If your plug-in application uses additional libraries, a COMMANDPARM string, and the APPLICATION attribute to provide access to the PBD's Application object, the Embed element might look like this:

```
<EMBED SRC=plugin_tree.pbd WIDTH=370 HEIGHT=320
WINDOW=w_emp_by_dept
LIBRARY=http://www.mycompany.com/pb/extra1.pbd
LIBRARY=http://www.mycompany.com/pb/extra2.pbd
COMMANDPARM="Eastern region"
APPLICATION=plugin_tree>
```

Setting up the server

After defining the HTML page that displays your plug-in application, you need to set up the Web server.

Specifying the MIME type

Using the appropriate software for your Web server, register the **MIME type** for the PowerBuilder window plug-in. Table 33-7 shows the MIME types you can use.

Table 33-7: PowerBuilder window plug-in MIME types

For this plug-in	Register this MIME type
Standard PowerBuilder window plug-in	application/vnd.powerbuilder11.0
Secure PowerBuilder window plug-in	application/vnd.powerbuilder11.0-s

The file extension associated with the MIME type is *PBD*.

Your server documentation might use the term **content type** instead of MIME type.

Putting the files on the server

You need to copy PowerBuilder libraries and the HTML file to appropriate directories on your server.

Table 33-8: PBD and HTML file locations on server

Files	Location	What you do
HTML page	The HTML documents directory	Copy the HTML page to the directory specified in any URLs that link to the page
PBD files named in the SRC and LIBRARY attributes of the Embed element	The HTML documents directory or some other directory as appropriate	Copy the PBD files to the directories you specified in the Embed element attributes

About URLs The URLs you specify in your HTML page are logical paths as defined by the Web server. On Windows, for example, the system path for your PBD might be:

```
C:\WEBSITE\HTDOCS\PB\PLUGIN_APP.PBD
```

If *htdocs* is defined as the server's documents directory, you can use the following relative path when specifying the SRC attribute of the Embed element. This path is relative to the documents directory:

```
pb/plugin_app.pbd
```

For the LIBRARY attribute, do not specify a relative URL for the PBD. You must specify a full URL. For example:

```
http://www.mycompany.com/pb/plugin_app.pbd
```

Setting up users' workstations

After building the plug-in application, creating its HTML page, and setting up the server, you need to make sure client workstations are set up to view the plug-in application.

To view a Web page that includes a PowerBuilder window plug-in application, users need supporting software installed on their client workstations. They also need a connection to the Web server.

Required components

Each client workstation needs the components in Table 33-9 to view a Web page containing a PowerBuilder window plug-in application.

Table 33-9: Client requirements to view PowerBuilder window plug-in

Component	Details
An Internet or intranet connection	Available within your corporation or from an Internet service provider.
A Web browser that supports Netscape plug-ins	Available from the browser vendor. Examples are: • Netscape Navigator Version 3.x or later. • Microsoft Internet Explorer Version 3.x through 5.5 Service Pack 1. Internet Explorer 5.5 Service Pack 2 and later versions do not support plug-ins.
PowerBuilder deployment DLLs	Install the PowerBuilder runtime DLLs as described in "PowerBuilder runtime files" on page 775. The PowerBuilder runtime DLLs belong in the application directory or in a directory on the system path. The PowerBuilder window plug-in DLL must know the location of the PowerBuilder runtime DLLs. To accomplish this, you can add the directory for the PowerBuilder deployment DLLs to the system path, or you can add the directory to the application path key for the browser (in the Windows registry).
Standard or secure PowerBuilder window plug-in DLL	If it is not there already, copy the *NPPBA110.DLL* file (standard) or *NPPBS110.DLL* file (secure) from the *Internet Tools\Plugins* directory to the browser's *Plugins* directory.
Other files	If you did not include image resources in the PBDs, copy them to the paths specified in the object properties if they are not there already. If the plug-in application connects to a database, set up client software for the DBMS. For how to connect to a DBMS from the client machine, see *Connecting to Your Database*.

Viewing the Web page and plug-in application

When the required software is installed, the user can view the Web page with the plug-in application.

When the user navigates to the URL of the Web page:

- The text on the page displays with a reserved space for the PowerBuilder window
- The client downloads the PBDs from the server
- The browser displays the child window within the Web page
- The user interacts with controls in the window and scripts for events execute
- The user navigates away from the page, which closes the window and causes the PowerBuilder DLLs to be unloaded from memory

CHAPTER 34 Using the PowerBuilder Window ActiveX

About this chapter

This chapter describes how to use the PowerBuilder window ActiveX.

Contents

About the PowerBuilder window ActiveX

The PowerBuilder window ActiveX lets you display a PowerBuilder child window on Web pages viewed in a browser that supports ActiveX.

Features

The PowerBuilder window can include all the familiar controls, including DataWindows, OLE objects, OCX (ActiveX) controls, and TreeView controls. You can also open other (pop-up or response) windows from the child window.

As the user interacts with controls in the windows, scripts for the controls' events are executed just as they are in standalone PowerBuilder applications. Additionally, you can call PowerBuilder functions and react to PowerBuilder events by coding VB Script or JavaScript within the HTML page.

Database access by the PowerBuilder window ActiveX application occurs using the client's locally defined database connections.

The objects in the application can be contained in one or more PowerBuilder Dynamic Libraries (PBDs).

Supported browsers

The PowerBuilder window ActiveX requires use of a browser that supports ActiveX, such as Microsoft Internet Explorer.

Kinds of applications that work with the PowerBuilder window ActiveX

HTML forms provide some user interaction by means of a limited user interface. The PowerBuilder window ActiveX takes you beyond HTML. You can present a fully developed application window with a rich user interface design to a Web page. You can access data sources defined on the client workstation.

Examples

Your application might be:

- A data analysis window with master and detail DataWindows
- A user interface design that lets the user make choices using TreeView, ListView, and PictureListBox controls
- A data entry form that processes data on a server
- A data entry form that uses the client's defined database connection (either network or local) to process the data on the client machine
- A window that uses PowerBuilder's OLEControl control to display an ActiveX control

How the PowerBuilder window ActiveX works

The PowerBuilder window ActiveX displays a PowerBuilder child window inside a fixed space reserved on the Web page. The user can interact with the controls on the page, and the PowerBuilder scripts for the window and its controls can execute any PowerBuilder code. When the user switches to another Web page, the PowerBuilder window is closed and the PowerBuilder DLLs are unloaded from memory.

The PowerBuilder window ActiveX is included on an HTML page using the HTML Object element. It names one or more PBDs that contain PowerBuilder objects, the name of the child window object that is displayed on the page, and (optionally) the PowerBuilder Application object.

Security

Unsecured applications can access local files and can run local applications. These types of processing are often undesirable in an uncontrolled Web environment but may be perfectly acceptable on a corporate intranet where access is controlled. There are two versions of the PowerBuilder window ActiveX: one secure (*PBRXS110.OCX*) and the other unsecure (*PBRX110.OCX*). The secure version is extremely restricted, with no access to the client workstation.

Some events always execute in secure mode

As of PowerBuilder 7, the application Open event and some of the Constructor events for controls execute in secure mode even when you use the unsecure version of the window ActiveX. If your application logic depends on the ability to perform an activity that is not secure in these events, such as opening a file for writing, post an event message to the application or control. This is processed once the window ActiveX returns to standard mode.

Requirements

The PowerBuilder window ActiveX uses the PowerBuilder virtual machine as well as the ActiveX itself to provide a full range of PowerBuilder functionality.

Each client that is used to browse pages with the PowerBuilder window ActiveX needs supporting software installed on the local machine:

- PowerBuilder virtual machine (*PBVM110.DLL* and supporting files)
- PowerBuilder window ActiveX

The PowerBuilder window ActiveX is especially useful in an intranet application where you have control over the setup of client machines.

Installing and configuring the PowerBuilder window ActiveX

The PowerBuilder window ActiveX controls are not installed on your computer if you select the Typical setup type when you install PowerBuilder. You can install the ActiveX controls by running a custom installation of PowerBuilder and checking the Web Plugins box on the Select Components page. Click the Change button on that page to select the plug-ins and controls you want to install.

This section describes setup tasks that you may need to perform *after* installing the window ActiveX.

Installation location

The PowerBuilder window ActiveX files (both the secure and unsecure versions) are installed in the *\Sybase\Shared\PowerBuilder* directory.

Register the ActiveX

For you to use the PowerBuilder window ActiveX for development, it must be registered on your machine. You can check to see if it is registered by using the PowerBuilder Browser. Open the Browser, select the OLE tab, and expand the OLE Custom Controls item. If the ActiveX is registered, it is listed as the PowerBuilder Window Control or PowerBuilder Secure Window Control in the tree view. Expand the Class Information node to verify that the current version of the control is registered.

If the ActiveX is not registered on your machine, you can register it from within PowerBuilder or by using the MS-DOS regsvr32 command.

❖ **To register the ActiveX within PowerBuilder:**

1. From within PowerBuilder, open a new or existing application, then open a new or existing window.
2. Select Insert>Control>OLE from the menu.

   The Insert Object dialog displays.
3. Select the Insert Control tab and click the Register New button.

   The Browse dialog displays.
4. Browse to the location of the OCX in the System directory, select either *PBRX110.OCX* or *PBRXS110.OCX*, and click Open.

   If the OCX fails to register, you receive an error message.

❖ **To register the ActiveX using the MS-DOS regsvr32 command:**

- Run the MS-DOS regsvr32.exe command with the complete path to the OCX as an argument. For example:

```
regsvr32.exe C:\Windows\System32\pbrx110.ocx
```

Developing and deploying a PowerBuilder window ActiveX application

There are four main tasks involved in developing a PowerBuilder window ActiveX application.

❖ **To create and deploy a PowerBuilder window ActiveX application:**

1. Create, test, and build the PowerBuilder application.
2. Create an HTML page that includes your PowerBuilder application window.

3 Configure the Web server by copying the HTML page and PBD files for the application to appropriate directories.

4 On all client workstations, install and register the PowerBuilder window ActiveX control and install the PowerBuilder runtime DLLs.

Resulting components

When everything is set up, the various computers have the components listed in Table 34-1.

Table 34-1: Server and client configuration for PowerBuilder window ActiveX application

Computer	Component
Server	HTML page with an Object element for the window (or report) and PBD
	PowerBuilder window ActiveX installed and registered (optional)
	One or more PBDs containing your child windows and other PowerBuilder objects
Client	PowerBuilder window ActiveX installed and registered
	PowerBuilder runtime DLLs installed as described in Chapter 41, “Deploying Applications and Components.”
	Directory for the PowerBuilder runtime DLLs listed in the system path
	Microsoft DLLS: *MFC42.DLL* *MSVCRT.DLL* *URL.DLL* *URLMON.DLL*
	Any additional software required by the PowerBuilder window ActiveX application, such as database connection software or OLE servers and ActiveX controls

Creating the PowerBuilder application

The starting point of your PowerBuilder window ActiveX application is a child window, which is displayed within a Web page. In the window, you can include controls and write scripts for events. Your scripts can open other windows, read and write files, and run other programs on the client machine.

Designing the application

The application you design for use with the PowerBuilder window ActiveX can be much the same as other PowerBuilder applications you develop; however, there are some restrictions and considerations to keep in mind. This section discusses:

- Window management
- Scripts and variables
- Data access
- External files

Window management

Initial child window Your initial window needs to be a child window that lives in the browser frame.

You *cannot*:

- Have a menu for the child window
- Open another child window from the initial window

You *can*:

- Include a title bar on the child window (but you *should not* use a control menu, maximize box, or minimize box on that window)
- Open pop-up and response windows from the child window (but not main or MDI windows)

Closing windows When the client browses to another Web page, the child window on the current Web page is closed, but other windows remain open unless your application closes them. You must close them in the child window's Close or CloseQuery events.

Do not try to stop the child window from being closed in the CloseQuery event by setting a return value. You cannot prevent the browser from changing to another page and removing the window from view. If the user returns to the page, another instance of your application is started.

Objects

Objects in your PBDs Your PowerBuilder window ActiveX application has access to all objects in the PBDs. This includes functions, structures, and user objects.

System objects Your PowerBuilder window ActiveX application has access to system objects that PowerBuilder instantiates, such as the SQLCA Transaction object and the Message object.

Application object Your PowerBuilder window ActiveX application can access global variables, which are properties of the Application object.

PowerBuilder requires that you have a current Application object during development, but only the application Open event and global variables are used in your application. Any other scripts in the Application object are used only during testing within PowerBuilder.

Scripts and variables

Scripts for application setup If you specify the PBAPPLICATION parameter in the HTML, the PowerBuilder window ActiveX executes the application Open event before opening the child window. You can use this event to establish a database connection and initialize global variables.

The HTML specifies the window to open
Do not open windows from within the application Open event.

If you do not specify the PBAPPLICATION parameter in the HTML, all application setup must be done in the child window, including connecting to the DBMS. The first scriptable events to occur are the constructors for the controls in the window, and then the Open event for the window occurs.

Note that the Activate event does not occur for a child window, so do not do application setup there.

Global variables The PowerBuilder window ActiveX application can access global variables.

Referencing the initial window You cannot reference the initial child window by name in your scripts because PowerBuilder does not create a variable to hold the instance of it. (By contrast, when you instantiate a window yourself by coding the Open function, you always place the instance in a variable that you can then reference.)

As a result, the following code produces an error at runtime (because the window variable w_mychild does not exist):

```
// This code produces a runtime error:
    w_mychild.title = "The initial window"
```

But you can code:

```
// In the child window itself:
    this.title = "The initial window"
    // or
    title = "The initial window"
```

```
// In controls of the child window:
   parent.title = "The initial window"
```

Data access

If your application accesses a DBMS, each client must have a connection to the data source. The connection must be defined on the client's machine, not the server. The data source can be a local or a network DBMS.

For information about how to connect to a DBMS from the client machine, see *Connecting to Your Database*.

Remember that the Constructor events for controls occur before the Open event of the window. That means that if you connect to the DBMS in the window's Open event, the Constructor events cannot get data. You can get data for controls in the window's Open event, or you can post events from the Constructor events or the window's Open event.

Paths for external resources

Paths you specify for files must be valid on the client workstation.

If your application uses images as external files, the images must be available at the path specified in the PowerBuilder object. Instead of using external files, you can build image resources into PBDs.

If your application reads or writes local files, the path for those files must be valid on each client's machine.

In Windows environments, if a path refers to a network drive by mapped drive letter, all clients must use the same drive letter that the application uses. As an alternative, specify the server name in the path instead.

For example, both of these paths are valid when *O:* is mapped to *\\marketing\drive*. However, the second path that uses a server name is more likely to *remain* valid:

```
O:\pbapps\connect.bmp
\\marketing\drive\pbapps\connect.bmp
```

Defining the starting window in the Window painter

You create the starting window for your application in PowerBuilder.

❖ **To create your application's starting window:**

1. Create a new window object in PowerBuilder.
2. On the window's General tab of the Properties view, set the window type to Child.

3 Add other controls as desired.

4 Write scripts for events of the window and controls.

❖ **To convert an existing application to run as a PowerBuilder window ActiveX:**

1 Change the type of the opening window to child.

2 Move application setup code from the application's Open event or MDI frame events to the child window's Open event.

Depending on the application's design, you may need to redesign how it opens other windows.

About child windows

Child windows cannot have menus. They are never considered the active window; therefore, the Activate event is never triggered. They can have title bars and can be minimizable, maximizable, and resizable. However, in the PowerBuilder window ActiveX environment, the child window is always restricted to the space allotted by the WIDTH and HEIGHT attributes specified on the Web page:

- Maximizing causes the window to fill the space allotted by the WIDTH and HEIGHT attributes.
- Minimizing displays the window's icon and title at the bottom of the ActiveX control's allotted space.
- If the child window is resizable, the user can drag the borders to make the window smaller (but not larger) than the allotted space.

As a result, it is not useful to allow minimizing, maximizing, or resizing of the child window.

Testing the application in PowerBuilder

Before you try your application in a client browser, you can test it in PowerBuilder by defining a main window that opens the child window.

❖ **To test your PowerBuilder window ActiveX application:**

1 In PowerBuilder, create a new window object whose type is main (the default).

2 Write a script for the Open event that opens the child window:

```
Open(w_child)
```

3. For convenience, you can set appropriate window sizes in the Window painter:
    - Make the main window large enough to display the child
    - Position the child window in the upper-left corner (you set the position on the Position tab of the property sheet)
4. Run the test window by clicking the Run Window icon.

Debugging in PowerBuilder
To use the PowerBuilder debugger, you need to run an application instead of a single window. Define an Application object with a script that opens the main window. Then you can use the Run or Debug commands to test the application.

Organizing objects in PBLs

In the Web environment, file size is important. Clients are downloading your application each time they run it (unless it happens to be cached because they ran it recently).

Before you build your application, use the Library painter to organize the objects your application uses in PBLs, which are the sources for the PowerBuilder window ActiveX application's PBDs. To minimize file size, include only objects the application uses. Remove any objects that are not needed. You must:

- Include any objects that are dynamically created, such as DataWindows used in DataStores or assigned dynamically to DataWindow controls and Proxy objects
- Include ancestor objects

Using PowerBuilder resource (PBR) files

Several controls can use external files for images. These include PictureListBox, TreeView, Picture, and PictureButton controls, pointers, and bitmap objects in DataWindow objects. If your application uses external files for the images, it is unlikely that the client has the same images on the same system path.

Instead of finding some way to install the pictures on client machines, you can use one or more PowerBuilder resource (PBR) files so that the images are built into the PBDs. The resulting PBDs are larger but self-contained.

Because you are building a PBD, not an executable, you do not need to include DataWindow objects in the resource file. All PowerBuilder objects in the source PBL are included in the resulting PBD.

Images and other resources on a network
Instead of building the files into the PBD, you can put the files in a generally accessible network directory, but the path to the files must be identical to the path named in the PowerBuilder objects. In the Windows environment, this means each client must use the same drive letter to map the network drive (or you can specify the server name in the path).

❖ **To define a PBR file:**

1 Open the File Editor (Shift+F6) or some other text editor and create a new file with the extension *PBR*.

2 List each image or other resource on its own line. List the path and file name exactly as they are shown in the object's Properties view or script.

Shortcut
Look at the object's Properties view and use Ctrl+C to copy the file name to the clipboard. Then paste it into the editor.

3 Save the file.

If your application includes several PBLs, each with objects using their own resources, you should create a PBR file for each PBL. The PBR file lists file names for resources that are used in one PBL.

Building the PBDs

Build your dynamic libraries (PBDs) using the System Tree or the Project or Library painter. Do the following:

- **Deselect Machine code** Your PowerBuilder window ActiveX application must have PBDs, not machine code DLLs.
- **Specify a PBR file for each PBD** If the objects in the PBL use resources that you have listed in a PBR file, put the PBR name in the Resource File Name text box.

For instructions on defining a PBR file and building your dynamic libraries (PBDs), see the chapter on creating executables in the *User's Guide*.

Creating an HTML page

After creating and building the PowerBuilder PBDs for your PowerBuilder window ActiveX application, you need to create the HTML page that displays it.

To include a PowerBuilder window on a Web page, you use the Object element. Element attributes specify the class ID for the PowerBuilder window ActiveX, the space allocated for the window, the name of the PBD, the name of the child window in the PBD, the library list, and the version of PowerBuilder.

A sample Object element might be:

```
<OBJECT NAME="PBRX1" WIDTH=225 HEIGHT=83
CLASSID="CLSID:BBBB1304-BBBB-1000-8000-080009AC61A9">
<PARAM NAME="_Version" VALUE="65536"></PARAM>
<PARAM NAME="_ExtentX" VALUE="5962"></PARAM>
<PARAM NAME="_ExtentY" VALUE="2164"></PARAM>
<PARAM NAME="_StockProps" VALUE="0"></PARAM>
<PARAM NAME="PBWindow" VALUE="w_helloworld"></PARAM>
<PARAM NAME="LibList"
VALUE="http://www.company.com/rknnt.pbd;"></PARAM>
<PARAM NAME="PBApplication" VALUE="hello"></PARAM>
<PARAM NAME="PBVersion" VALUE="100"></PARAM>
</OBJECT>
```

Attributes of the Object element

The Object element is part of the HTML specification for ActiveX controls. It defines several standard attributes, and PowerBuilder defines additional attributes.

HTML attributes

HTML attributes specify the class ID, the name, and the space reserved for the PowerBuilder window ActiveX on the Web page.

Table 34-2: HTML attributes of Object element

HTML attribute	Value
NAME	Name of the object when referenced in code or when submitted as part of a form.
CLASSID	The class ID of the registered ActiveX control. The syntax is: CLSID:*class_id* To find the class ID value of the registered ActiveX control, you can use the PowerBuilder Browser's OLE tab. Expand the OLE Custom Controls item, then expand the PowerBuilder Window Control or Secure Window Control item. The class ID is the value of the GUID item of the Class Information branch.
CODEBASE	A URL identifying the location of the OCX or CAB file to be downloaded if the client machine does not contain the PowerBuilder window ActiveX. The client machine must still have the PowerBuilder virtual machine and any other required DLLs on the system path.
WIDTH	The width of the viewing window in pixels.
HEIGHT	The height of the viewing window in pixels.

The WIDTH and HEIGHT attributes define the maximum width and height of the child window. If the child window is resizable, the user can make it smaller than the specified size, but not larger.

Param elements

To specify properties for the PowerBuilder window ActiveX, include Param elements. Param elements let you identify the window object that starts your application, the Application object, additional libraries, and additional parameters to pass to the PowerBuilder window ActiveX. Table 34-3 lists the PowerBuilder-specific Param elements.

Table 34-3: PowerBuilder window ActiveX Param elements

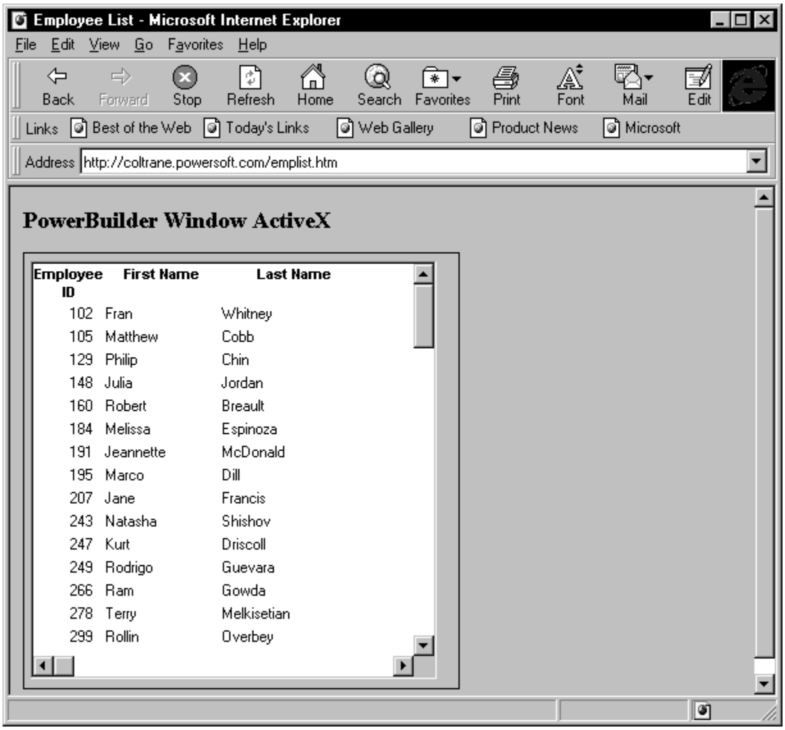
Property	Value
PBWINDOW	The class name of the child window in the PBD.
LIBLIST	A list of PowerBuilder dynamic libraries (PBD files) required by the application. Separate multiple entries with a semicolon.
PBAPPLICATION (optional)	The PowerBuilder Application object.
PBVERSION	The version of the PowerBuilder DLLs (for example, 110).

Property	Value
DISPLAYERRORS (optional)	Boolean indicating whether to display execution errors.
COMMANDPARM (optional)	A string you want to pass to your window. To access this string from within your window, call the CommandParm function.

Coding the Object element
To minimize coding errors, use an ActiveX-aware HTML editor (such as the JSP targets HTML editor, the ActiveX Control Pad, or Front Page) when coding an Object element and its parameters.

Basic page

This sample page includes the PowerBuilder window ActiveX:



Address http://coltrane.powersoft.com/emplist.htm

PowerBuilder Window ActiveX

Employee ID	First Name	Last Name
102	Fran	Whitney
105	Matthew	Cobb
129	Philip	Chin
148	Julia	Jordan
160	Robert	Breault
184	Melissa	Espinoza
191	Jeannette	McDonald
195	Marco	Dill
207	Jane	Francis
243	Natasha	Shishov
247	Kurt	Driscoll
249	Rodrigo	Guevara
266	Ram	Gowda
278	Terry	Melkisetian
299	Rollin	Overbey

Here is the HTML code that produces this page (note the use of the Object element to specify the PowerBuilder window ActiveX):

```
<HTML>
<HEAD>
<TITLE>Employee List</TITLE>
</HEAD>
<BODY>
<H1>PowerBuilder window ActiveX</H1>
<P>
<OBJECT ID="PBRX1" NAME="PBRX1" WIDTH=357 HEIGHT=269
CLASSID="CLSID:BBBB1304-BBBB-1000-8000-080009AC61A9">
<PARAM NAME="_Version" VALUE="65536">
<PARAM NAME="_ExtentX" VALUE="9440">
<PARAM NAME="_ExtentY" VALUE="7112">
<PARAM NAME="_StockProps" VALUE="0">
<PARAM NAME="PBWindow" VALUE="w_emplist">
<PARAM NAME="LibList"
VALUE="http://www.company.com/rknnt.pbd;">
<PARAM NAME="PBApplication" VALUE="rknnt">
<PARAM NAME="PBVersion" VALUE="110">
</OBJECT>
</BODY>
</HTML>
```

Client-side scripting

You can interact with the window displayed in a PowerBuilder window ActiveX by adding JavaScript or VBScript to the HTML page. You can:

- Code event handlers that respond to events that occur in the window
- Call PowerScript functions to obtain pointer information, print, set redraw, or set a timer
- Call the InvokePBFunction function to invoke a user-defined window function
- Call the TriggerPBEvent function to trigger a user event on the window

Viewing ActiveX properties, events, and functions
When the window ActiveX is registered on your machine, you can use the PowerBuilder Browser's OLE tab to see the list of window ActiveX properties, events, and functions.

Coding event handlers

Your HTML page can contain JavaScript or VBScript event handlers for the PowerBuilder window ActiveX.

Coding example assumptions
The following code examples assume that the HTML page includes a Form, named buttonForm, which contains several Input elements: passedFlags, passedXPos, and passedYPos.

❖ To code JavaScript event handlers for the PowerBuilder window ActiveX:

1 Insert the PowerBuilder window ActiveX into the HTML page, specifying all necessary properties:

```
<OBJECT NAME="PBRX1" WIDTH=225 HEIGHT=83
CLASSID="CLSID:BBBB1304-BBBB-1000-8000-
080009AC61A9">
<PARAM NAME="_Version" VALUE="65536">
<PARAM NAME="_ExtentX" VALUE="5962">
<PARAM NAME="_ExtentY" VALUE="2164">
<PARAM NAME="_StockProps" VALUE="0">
<PARAM NAME="PBWindow" VALUE="w_helloworld">
<PARAM NAME="LibList"
VALUE="http://www.company.com/rknnt.pbd;">
<PARAM NAME="PBApplication" VALUE="rknnt">
<PARAM NAME="PBVersion" VALUE="110">
</OBJECT>
```

2 Within the heading of the HTML page, code a function to be called when the event occurs.

The following sample function simply displays the arguments to the Clicked event.

```
function wasClicked(flags, xpos, ypos) {
   document.buttonForm.passedFlags.value = flags;
   document.buttonForm.passedXPos.value = xpos;
   document.buttonForm.passedYPos.value = ypos;
}
```

3 Within the body of the HTML page, code an event handler that calls the function when the event occurs:

```
<SCRIPT LANGUAGE="JavaScript" FOR="PBRX1"
Event="Clicked(flags, xpos, ypos)">
<!--
wasClicked(flags, xpos, ypos);
```

```
-->
</SCRIPT>
```

Coding style
Alternatively, you can omit the function call, placing all code within the event handler, as shown next.

❖ To code VBScript event handlers for the PowerBuilder window ActiveX:

1 Insert the PowerBuilder window ActiveX into the HTML page, specifying all necessary properties:

```
<OBJECT NAME="PBRX1" WIDTH=225 HEIGHT=83
CLASSID="CLSID:BBBB1304-BBBB-1000-8000-
080009AC61A9">
<PARAM NAME="_Version" VALUE="65536">
<PARAM NAME="_ExtentX" VALUE="5962">
<PARAM NAME="_ExtentY" VALUE="2164">
<PARAM NAME="_StockProps" VALUE="0">
<PARAM NAME="PBWindow" VALUE="w_helloworld">
<PARAM NAME="LibList"
VALUE="http://www.company.com/rknnt.pbd;">
<PARAM NAME="PBApplication" VALUE="rknnt">
<PARAM NAME="PBVersion" VALUE="110">
</OBJECT>
```

2 Within the body of the HTML page, code an event handler that processes the event.

This sample function simply displays the arguments to the Clicked event:

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub PBRX1_Clicked(flags, xpos, ypos)
   document.buttonForm.passedFlags.value = flags
   document.buttonForm.passedXPos.value = xpos
   document.buttonForm.passedYPos.value = ypos
end sub
-->
</SCRIPT>
```

Calling PowerScript functions

The PowerBuilder window ActiveX allows you to call certain PowerScript functions on the window displayed in the Active control:

- **PointerX** Returns the distance from the left edge of the window to the pointer.
- **PointerY** Returns the distance from the top of the window to the pointer.

- **Print** Prints the window.
- **SetRedraw** Turns on or off automatic redrawing of the window after every change.
- **Timer** Causes the window's Timer event to occur repeatedly at the specified interval.

For more information on these functions, see the *PowerScript Reference.*

As with all ActiveX controls, the PowerBuilder window ActiveX provides an AboutBox function, which you can call to see information about the control.

Coding example assumptions
The following coding examples assume that you have written scripts to be invoked by the PowerBuilder window's Timer event.

❖ To use JavaScript to call a PowerScript function:

1 Code a function to call the PowerScript function.

This example calls the PowerScript Timer function:

```
<SCRIPT LANGUAGE="JavaScript">
<!--
var cumSeconds = 0;
function setPBTimer( f ) {
var li_return
var li_interval
li_interval = parseInt(f.timerInterval.value);
li_return = PBRX1.Timer(li_interval);
if (li_return != 1) {
   alert("Set Timer failed");
}
}
//-->
/SCRIPT
```

2 Code a function, anchor, or button that calls the function.

This example uses a button on a form to call the function defined above, which resets the timer interval:

```
<FORM NAME="clockForm">
<P>Timer Interval:
<INPUT TYPE=Text NAME="timerInterval" Size="5">
<P><INPUT TYPE=BUTTON VALUE="Set Timer"
ONCLICK="setPBTimer(this.form)">
</FORM>
```

❖ To use VBScript to call a PowerScript function:

1 Code a function to call the PowerScript function. This example calls the PowerScript Timer function:

```
<SCRIPT LANGUAGE="VBScript">
<!--
dim cumSeconds
cumSeconds = 0
Sub pbSetTime()
  dim li_return
  dim li_interval
  li_interval = clockForm.timerInterval.value
  li_return = pbrx1.Timer(li_interval)
  if li_return  1 THEN
    msgBox "Set Timer failed"
  end if
  end sub
//-->
</SCRIPT>
```

2 Code a function, anchor, or button that calls the function.

This example uses a button on a form to call the function defined above, which resets the timer interval:

```
<FORM NAME="clockForm">
<P>Timer Interval:
<INPUT TYPE=Text NAME="timerInterval" Size="5">
<HR>
<P>Mirror of PB Time:
<INPUT TYPE=Text NAME="pbTime" Size="8">
<HR>
<P>INPUT TYPE=BUTTON VALUE="Set Timer" onClick="call
pbSetTime()">
</FORM>
```

Calling user-defined functions

The PowerBuilder window ActiveX provides the InvokePBFunction function, which you can use to call a user-defined window function.

VBScript and JavaScript differ
If your user-defined functions contain arguments and you are using JavaScript, you must use SetArgElement to specify the arguments; you cannot specify the arguments explicitly in the InvokePBFunction function.

- ❖ **To code JavaScript that invokes a user-defined function:**

1. Define window functions as needed.

   The following example assumes that in the PowerBuilder window, you have defined the function of_arg that takes a string as a parameter.

2. Code a JavaScript function that calls the InvokePBFunction function, specifying the user-defined function to invoke.

   This example initializes arguments and calls the of_args window function:

   ```
   function invokeFunc(f) {
     var retcd;
     var rc;
     var numargs;
     var theFunc;
     var theArg;
     retcd = 0;
     numargs = 1;
     theArg = f.textToPB.value;
     PBRX1.SetArgElement(1, theArg);
     theFunc = "of_args";
     retcd = PBRX1.InvokePBFunction(theFunc, numargs);
     rc = parseInt(PBRX1.GetLastReturn());
     if (rc != 1) {
       alert("Error. Empty string.");
     }
     PBRX1.ResetArgElements();
   }
   ```

3. Code a function, anchor, or form button that invokes your JavaScript function. For example:

   ```
   <FORM>
   <P>Copy this text to PowerBuilder:
   <INPUT TYPE=Text NAME="textToPB" SIZE="20">
   <P><INPUT TYPE=BUTTON VALUE="Invoke Func"
   ONCLICK="invokeFunc(this.Form)">
   </FORM>
   ```

Defining arguments in JavaScript
When coding in JavaScript, define function and event arguments by calling the SetArgElement function.

❖ To code VBScript that invokes a user-defined function:

1. Define window functions as needed.

   The following example assumes that in the PowerBuilder window you have defined the function of_arg that takes a string as a parameter.

2. Code a VBScript function that calls the InvokePBFunction function, specifying the user-defined function to invoke.

   This example initializes arguments and calls the of_args window function:

   ```
   Sub invokeFunction()
     Dim retcd
     Dim myForm
     Dim args(1)
     Dim rc
     Dim numargs
     Dim theFunc
     Dim rcfromfunc
     retcd = 0
     numargs = 1
     rc = 0
     theFunc = "of_args"
     Set myForm = Document.buttonForm
     args(0) = buttonForm.textToPB.value
     retcd = PBRX1.InvokePBFunction(theFunc,
       numargs, args)
     rc = PBRX1.GetLastReturn()
     if rc  1 then
       msgbox "Error. Empty string."
     end if
     PBRX1.ResetArgElements()
   end sub
   ```

3. Code a function, anchor, or form button whose click invokes your VBScript function. For example:

   ```
   <FORM NAME="buttonForm">
   <P><INPUT TYPE=Text NAME="textToPB" SIZE="20">
   <P><INPUT TYPE=BUTTON VALUE="Invoke Function"
   ONCLICK="call invokeFunction()">
   </FORM>
   ```

Calling user events

The PowerBuilder window ActiveX provides the TriggerPBEvent function, which you can use to call a user event on the window.

❖ To code JavaScript that triggers a user event:

1 Define user events on the window as needed.

The following example assumes that in the PowerBuilder window you have defined the user event ue_args that takes a string as an argument.

2 Code a function to call the user event. This example initializes arguments and calls the ue_args window event:

```
function triggerEvent(f) {
  var retcd;
  var rc;
  var numargs;
  var theEvent;
  var theArg;
  retcd = 0;
  numargs = 1;
  theArg = f.textToPB.value;
  PBRX1.SetArgElement(1, theArg);
  theEvent = "ue_args";
  retcd = PBRX1.TriggerPBEvent(theEvent, numargs);
  rc = parseInt(PBRX1.GetLastReturn());
  if (rc != 1) {
    alert("Error. Empty string.");
  }
  PBRX1.ResetArgElements();
}
```

3 Code a function, anchor, or form button that calls the TriggerPBEvent function. For example:

```
<FORM>
<P><INPUT TYPE=Text NAME="textToPB" SIZE="20">
<P><INPUT TYPE=BUTTON VALUE="Trigger Event"
ONCLICK="triggerEvent(this.Form)">
</FORM>
```

❖ To code VBScript that triggers a user event:

1 Define user events on the window, as needed.

The following example assumes that in the PowerBuilder window you have defined the user event ue_args that takes a string as an argument.

2 Code a function to call the user event. This example initializes arguments and calls the ue_args window function:

```
Sub TrigEvent( )
  Dim retcd
  Dim myForm
  Dim args(1)
  Dim rc
  Dim numargs
  Dim theEvent
  retcd = 0
  numargs = 1
  rc = 0
  theEvent = "ue_args"
  Set myForm = Document.buttonForm
  args(0) = buttonForm.textToPB.value
  retcd = PBRX1.TriggerPBEvent(theEvent,
       numargs, args)
  rc = PBRX1.GetLastReturn()
  if rc  1 then
    msgbox "Error. Empty string."
  end if
  PBRX1.ResetArgElements()
end sub
```

3 Code a function, anchor, or form button whose click invokes your VBScript function. For example:

```
<FORM NAME="buttonForm">
<P><INPUT TYPE=Text NAME="textToPB" SIZE="20">
<P><INPUT TYPE=BUTTON VALUE="Trigger Event"
ONCLICK="call TrigEvent()">
</FORM>
```

Events for the PowerBuilder window ActiveX

The PowerBuilder window ActiveX can respond to certain events occurring within the child window. These are outbound events, which execute first within the PowerBuilder window and then in the PowerBuilder window ActiveX. You can add JavaScript or VB Script code that responds to these events. The events are listed in Table 34-4.

Table 34-4: PowerBuilder window ActiveX events

Event	Occurs
Activate	Just before the window becomes active
Clicked	When the user clicks in an unoccupied area of the window (any area with no visible, enabled object)
Close	When the window is closed
Deactivate	When the window becomes inactive
DoubleClicked	When the user double-clicks in an unoccupied area of the window (any area with no visible, enabled object)
Hide	Just before the window is hidden
Key	When the user presses a key and the insertion point is not in a line edit
MouseDown	When the user presses the left mouse button in an unoccupied area of the window (any area with no visible, enabled object)
PBMouseMove	When the pointer is moved within the window
PBMouseUp	When the user releases the left mouse button in an unoccupied area of the window (any area with no visible, enabled object)
RButtonDown	When the user presses the right mouse button in an unoccupied area of the window (any area with no visible, enabled object)
Resize	When the user or a script opens or resizes a window
Show	When a script executes the Show function for this window (the event occurs just before the window is displayed)
SystemKey	When the insertion point is not in a line edit and the user presses Alt or Alt plus another key
Timer	When a specified number of seconds elapses after the Timer function has been called

Setting up the server

Setting up the server involves placing files on the server.

You need to copy PowerBuilder libraries, the PowerBuilder window ActiveX module, and the HTML file to appropriate directories on your server.

Table 34-5: PBD and HTML file locations on server

Files	Location
HTML page	The HTML documents directory. Copy the HTML page to the directory specified in any URLs that link to the page.
PBD files named in the LibList attributes of the Param element	A directory named in the application path for the Web server application or any directory on the system path.
PowerBuilder window ActiveX module	The HTML documents directory or some other directory as specified in the Object element's CODEBASE attribute.

Setting up users' workstations

Users who need to view a Web page that includes a PowerBuilder window ActiveX require supporting software installed on their client workstations and a connection to the Web server.

Table 34-6: Client requirements to view PowerBuilder window ActiveX

Component	Details
An Internet or intranet connection	Available within your corporation or from an Internet service provider.
A Web browser that supports ActiveX	Available from the browser vendor, for example Microsoft Internet Explorer.
PowerBuilder runtime DLLs	Install the runtime files as described in "PowerBuilder runtime files" on page 775. The PowerBuilder runtime DLLs belong in the application directory or in a directory on the system path. The PowerBuilder window ActiveX *must be able to access* the PowerBuilder virtual machine (and any other required runtime DLLs). To accomplish this, you can add the directory for the PowerBuilder runtime DLLs to the system path, or you can place the runtime DLLs in the *Windows\System32* directory.

Component	Details
PowerBuilder window ActiveX module	Copy the file *PBRX110.OCX* to the client workstation and register it. Alternatively, you can add the CODEBASE attribute to the Object element, which causes the browser to download and register it when the page is loaded.
Other files	If you did not include image resources in the PBDs, copy them to the paths specified in the object properties if they are not there already. If the PowerBuilder window ActiveX application connects to a database, set up client software for the DBMS. For how to connect to a DBMS from the client machine, see *Connecting to Your Database*.

Viewing the Web page and PowerBuilder window ActiveX application

When the required software is installed, the user can view the Web page with the PowerBuilder window ActiveX application.

When the user navigates to the URL of the Web page:

- The text on the page displays with a reserved space for the PowerBuilder window
- The client downloads the PBDs from the server
- The browser displays the child window within the Web page
- The user interacts with controls in the window and scripts for events execute
- The user navigates away from the page, which closes the window and causes the PowerBuilder DLLs or shared libraries to be unloaded from memory

PART 8

General Techniques

This part describes techniques for handling internationalization, printing, accessibility requirements, and the Windows registry. It explains how to build styles and actions for use in InfoMaker.

CHAPTER 35 Internationalizing an Application

About this chapter

This chapter describes some of the issues that arise when you develop and deploy applications for multiple languages.

Contents

Developing international applications

When you develop an application for deployment in multiple languages, you can take advantage of the Unicode support built into PowerBuilder. You also need to focus on two phases of the development process:

- The first is the **internationalization** phase, when you deal with design issues before you begin coding the application.
- The second is the **localization** phase, which starts once the development phase of an internationalized application is complete, when you deal with the translation and deployment of your application you enter the.

Using Unicode

Unicode is a character encoding scheme that enables text display for most of the world's languages. Support for Unicode characters is built into PowerBuilder. This means that you can display characters from multiple languages on the same page of your application, create a flexible user interface suitable for deployment to different countries, and process data in multiple languages.

About Unicode

Before Unicode was developed, there were many different encoding systems, many of which conflicted with each other. For example, the same number could represent different characters in different encoding systems. Unicode provides a unique number for each character in all supported written languages. For languages that can be written in several scripts, Unicode provides a unique number for each character in each supported script.

For more information about the supported languages and scripts, see the Unicode Web site at http://www.unicode.org/onlinedat/languages-scripts.html.

Encoding forms

There are three Unicode encoding forms: UTF-8, UTF-16, and UTF-32. Originally UTF stood for Unicode Transformation Format. The acronym is used now in the names of these encoding forms, which map from a character set definition to the actual code units that represent the data, and to the encoding schemes, which are encoding forms with a specific byte serialization.

- UTF-8 uses an unsigned byte sequence of one to four bytes to represent each Unicode character.
- UTF-16 uses one or two unsigned 16-bit code units, depending on the range of the scalar value of the character, to represent each Unicode character.
- UTF-32 uses a single unsigned 32-bit code unit to represent each Unicode character.

Encoding schemes

An encoding scheme specifies how the bytes in an encoding form are serialized. When you manipulate files, convert blobs and strings, and save DataWindow data in PowerBuilder, you can choose to use ANSI encoding, or one of three Unicode encoding schemes:

- UTF-8 serializes a UTF-8 code unit sequence in exactly the same order as the code unit sequence itself.
- UTF-16BE serializes a UTF-16 code unit sequence as a byte sequence in big-endian format.
- UTF-16LE serializes a UTF-16 code unit sequence as a byte sequence in little-endian format.

UTF-8 is frequently used in Web requests and responses. The big-endian format, where the most significant value in the byte sequence is stored at the lowest storage address, is typically used on UNIX systems. The little-endian format, where the least significant value in the sequence is stored first, is used on Windows.

Unicode support in PowerBuilder

PowerBuilder uses UTF-16LE encoding internally. The source code in PBLs is encoded in UTF-16LE, any text entered in an application is automatically converted to Unicode, and the string and character PowerScript datatypes hold Unicode data only. Any ANSI or DBCS characters assigned to these datatypes are converted internally to Unicode encoding.

Support for Unicode databases

Most PowerBuilder database interfaces support both ANSI and Unicode databases.

A Unicode database is a database whose character set is set to a Unicode format, such as UTF-8 or UTF-16. All data in the database is in Unicode format, and any data saved to the database must be converted to Unicode data implicitly or explicitly.

A database that uses ANSI (or DBCS) as its character set can use special datatypes to store Unicode data. These datatypes are NChar, NVarChar, and NVarChar2. Columns with one of these datatypes can store Unicode data, but data saved to such a column must be converted to Unicode explicitly.

For more specific information about each interface, see *Connecting to Your Database*.

String functions

PowerBuilder string functions, such as Fill, Len, Mid, and Pos, take characters instead of bytes as parameters or return values and return the same results in all environments. These functions have a “wide” version (such as FillW) that is obsolete and will be removed in a future version of PowerBuilder because it produces the same results as the standard version of the function. Some of these functions also have an ANSI version (such as FillA). This version is provided for backwards compatibility for users in DBCS environments who used the standard version of the string function in previous versions of PowerBuilder to return bytes instead of characters.

You can use the GetEnvironment function to determine the character set used in the environment:

```
environment env
getenvironment(env)

choose case env.charset
case charsetdbcs!
    // DBCS processing
    ...
case charsetunicode!
    // Unicode processing
    ...
```

```
case charsetansi!
    // ANSI processing
    ...
case else
    // Other processing
    ...
end choose
```

Encoding enumeration

Several functions, including Blob, BlobEdit, FileEncoding, FileOpen, SaveAs, and String, have an optional *encoding* parameter. These functions let you work with blobs and files with ANSI, UTF-8, UTF-16LE, and UTF-16BE encoding. If you do not specify this parameter, the default encoding used for SaveAs and FileOpen is ANSI. For other functions, the default is UTF-16LE.

The following examples illustrate how to open different kinds of files using FileOpen:

```
// Read an ANSI File
Integer li_FileNum
String s_rec
li_FileNum = FileOpen("Employee.txt")
// or:
// li_FileNum = FileOpen("Emplyee.txt", &
//    LineMode!, Read!)
FileRead(li_FileNum, s_rec)

// Read a Unicode File
Integer li_FileNum
String s_rec
li_FileNum = FileOpen("EmployeeU.txt", LineMode!, &
   Read!, EncodingUTF16LE!)
FileRead(li_FileNum, s_rec)

// Read a Binary File
Integer li_FileNum
blob bal_rec
li_FileNum = FileOpen("Employee.imp", Stream Mode!, &
   Read!)
FileRead(li_FileNum, bal_rec)
```

Initialization files

The SetProfileString function can write to initialization files with ANSI or UTF16-LE encoding on Windows systems, and ANSI or UTF16-BE encoding on UNIX systems. The ProfileInt and ProfileString PowerScript functions and DataWindow expression functions can read files with these encoding schemes.

Exporting and importing source

The Export Library Entry dialog box lets you select the type of encoding for an exported file. The choices are ANSI/DBCS, which lets you import the file into PowerBuilder 9 or earlier, HEXASCII, UTF8, or Unicode LE.

The HEXASCII export format is used for source-controlled files. Unicode strings are represented by hexadecimal/ASCII strings in the exported file, which has the letters HA at the beginning of the header to identify it as a file that might contain such strings. You cannot import HEXASCII files into PowerBuilder 9 or earlier.

If you import an exported file from PowerBuilder 9 or earlier, the source code in the file is converted to Unicode before the object is added to the PBL.

External functions

When you call an external function that returns an ANSI string or has an ANSI string argument, you must use an ALIAS clause in the external function declaration and add `;ansi` to the function name. For example:

```
FUNCTION int MessageBox(int handle, string content,
string title, int showtype)
LIBRARY "user32.dll" ALIAS FOR "MessageBoxA;ansi"
```

The following declaration is for the “wide” version of the function, which uses Unicode strings:

```
FUNCTION int MessageBox(int handle, string content,
string title, int showtype)
LIBRARY "user32.dll" ALIAS FOR "MessageBoxW"
```

If you are migrating an application from PowerBuilder 9 or earlier, PowerBuilder replaces function declarations that use ANSI strings with the correct syntax automatically.

Setting fonts for multiple language support

The default font in the System Options and Design Options dialog boxes is Tahoma.

Setting the font in the System Options dialog box to Tahoma ensures that multiple languages display correctly in the Layout and Properties views in the Window, User Object, and Menu painters and in the wizards.

If the font on the Editor Font page in the Design Options dialog box is not set to Tahoma, multiple languages *cannot* be displayed in Script views, the File and Source editors, the ISQL view in the DataBase painter, and the Debug window.

You can select a different font for printing on the Printer Font tab page of the Design Options dialog box for Script views, the File and Source editors, and the ISQL view in the DataBase painter. If the printer font is set to Tahoma and the Tahoma font is not installed on the printer, PowerBuilder downloads the entire font set to the printer when it encounters a multilanguage character. If you need to print multilanguage characters, specify a printer font that is installed on your printer.

To support multiple languages in DataWindow objects, set the font in every column and text control to Tahoma.

The default font for print functions is the system font. Use the PrintDefineFont and PrintSetFont functions to specify a font that is available on users' printers and supports multiple languages.

PBNI

The PowerBuilder Native Interface is Unicode based. PBNI extensions must be compiled using the _UNICODE preprocessor directive in your C++ development environment.

Your extension's code must use TCHAR, LPTSTR, or LPCTSTR instead of char, char*, and const char* to ensure that it works correctly in a Unicode environment. Alternatively, you can use the MultiByteToWideChar function to map character strings to Unicode strings. For more information about enabling Unicode in your application, see the documentation for your C++ development environment.

Unicode enabling for Web services

In a PowerScript target, the PBNI extension classes instantiated by Web service client applications use Unicode for all internal processing. However, calls to component methods are converted to ANSI for processing by EasySoap, and data returned from these calls is converted to Unicode.

In a JSP target, the authoring tool (HTML editor) is Unicode-enabled so you can input text in multiple languages on a single page. When you type in the editor, the text is saved in UTF-16LE encoding. However, JSP files with other encoding schemes can still be imported in the editor. Text with these encodings is automatically converted to UTF-16LE.

XML string encoding

The XML parser cannot parse a string that uses an eight-bit character code such as windows-1253. For example, a string with the following declaration cannot be parsed:

```
string ls_xml
ls_xml += '<?xml version="1.0" encoding="windows-
1253"?>'
```

You must use a Unicode encoding value such as UTF16-LE.

Internationalizing the user interface

When you build an application for international deployment, there are two user interface design issues you should consider:

- The physical design of the user interface
- The cultural standards of your application's audience

Physical design

The physical design of the user interface should include:

- Windows and objects with the flexibility to accommodate expanded string lengths required when the text in menu items, lists, and labels is translated

  For example, you could inherit a window from an English language ancestor window, and change the language for a localized deployment. Generally, you can accommodate the text for most languages if you allow for a menu item, list, or label size that is 1.3 times the length of an English text string.

- Windows that can be easily used in RightToLeft versions of Windows

Cultural awareness

The cultural design of your user interface requires you to be cognizant of what is and is not acceptable or meaningful to your audience.

For example, an icon of a hand displaying an open palm might mean stop in one culture but indicate an unacceptable gesture in another. Similarly, although the color yellow signifies caution in some cultures, in other cultures it signifies happiness and prosperity.

Localizing the product

PowerBuilder provides resources for international developers that include localized runtime files and the Translation Toolkit. The localized files become available after the general release of a new version of PowerBuilder.

Localized runtime files

Localized runtime files are provided for French, German, Italian, Spanish, Dutch, Danish, Norwegian, and Swedish. You can install localized runtime files in the development environment or on the user's machine. If you install them on the development machine, you can use them for testing purposes.

The localized PowerBuilder runtime files handle language-specific data at runtime. They are required to display standard dialog boxes and user interface elements, such as day and month names in spin controls, in the local language. They also provide the following features:

- **DayName function manipulation** The DayName function returns a name in the language of the runtime files available on the machine where the application is run.
- **DateTime manipulation** When you use the String function to format a date and the month is displayed as text (for example, the display format includes “mmm”), the month is in the language of the runtime files available when the application is run.
- **Error messages** PowerBuilder error messages are translated into the language of the runtime files.

Localized PFC libraries

The PFC is now available on the Sybase CodeXchange Web site in the PFC project at http://pfc.codeXchange.sybase.com/.

In order to convert an English language PFC-based application to another language such as Spanish, you need multiple components. You need to test the application on a computer running the localized version of the operating system with appropriate regional settings. You must also obtain or build localized PFC libraries and install the localized PowerBuilder runtime files. When you deploy the application, you must deploy it to a computer running a localized version of the operating system, and you must deploy the localized runtime files.

You can translate the PFC libraries with the Translation Toolkit. Localized PFC libraries are the same as the original PFC libraries except that strings that occur in windows, menus, DataWindow objects, dialog boxes, and other user interface elements, and in runtime error messages, are translated into the local language. These include, for example, day and month names in the Calendar service. All services remain otherwise the same. In a Spanish PFC application, error messages displayed by the PFC are in Spanish, month names in the Calendar service are in Spanish, column headers in DataWindow objects and Menu items are in Spanish, and so on.

The Translation Toolkit adds a string in the format `%LANGUAGE%` to the comment associated with every object that contains a translated string. For example, if you look at a PFC library that has been translated into Spanish in the List view in the Library painter, you will notice the string `%SPANISH%` at the beginning of the comment for many objects.

The dictionaries used to translate the PFC libraries into each language are provided with the Translation Toolkit. You can use the dictionaries to translate the rest of your application into a local language using the Translation Toolkit, and you can view the dictionary in a text editor to see which strings have been translated.

The localized PFC libraries work in coordination with the localized runtime files, regional settings, and the localized operating system.

Regional settings

PowerBuilder always uses the system's regional settings, set in the Windows Control Panel, to determine formats for the Date and Year functions, as well as date formats to be used by the SaveAs function. The use of these regional settings is independent of the use of PowerBuilder localized runtime files or PFC libraries.

The regional settings are also used to determine behavior when using Format and Edit masks. For more information, see the section on defining display formats in the *User's Guide*.

Localized operating system

The localized operating system is required for references to System objects, such as icons and buttons, that are referenced using enumerated types in PowerBuilder, such as *OKCancel!*, *YesNo!*, *Information!*, and *Error!*. These enumerated types rely on API calls to the local operating system, which passes back the appropriate button, icon or symbol for the local language. For example, if you use the *OKCancel!* argument in a MessageBox function, the buttons that display on the message box are labeled OK and Cancel if the application is *not* running on a localized operating system.

About the Translation Toolkit

The Translation Toolkit is a set of tools designed to help you translate PowerBuilder applications into other languages. It includes a standalone translator tool that is used by the person or group translating the text of the application.

When you use the Toolkit to create a project, a copy of each of your application's source libraries is created for each project. The application's original source libraries are not changed.

How the Toolkit works

You work with the **phrases** (one or more words of text) in an application. These phrases are in the application's object properties, controls, and scripts.

You use the tools to:

- Extract phrases from the project libraries
- Present the phrases for translation
- Substitute translated phrases for the original phrases in the project libraries

Using the translated project libraries, you use PowerBuilder to build the translated application.

For more information, see the online Help for the Translation Toolkit.

CHAPTER 36 Building Accessible Applications

About this chapter

This chapter provides information about guidelines and requirements for making applications accessible to users with disabilities. It explains what features PowerBuilder offers to support the creation of accessible applications, and it includes pointers to additional sources of information.

Contents

Understanding accessibility challenges

When designing and developing software applications and Web pages that you want to make accessible to people with disabilities, there are four general types of impairments you need to consider:

- Visual
- Hearing
- Mobility
- Cognitive or learning

Visual impairments

Application users who are blind require text equivalents for all graphic images and videos available to the sighted user. The text needs to convey content that is conceptually equivalent to the information provided in graphical form, so that assistive technologies such as screen and braille readers can make the information fully accessible. All user interface (UI) elements must have text or menu equivalents, and blind users need keyboard equivalents for entering input that a sighted user would enter with a mouse.

To accommodate users who are color blind, you should avoid using color as the sole means of conveying information. Using fill patterns in addition to colors in graphs and other images is one strategy for supplementing information conveyed by color. Auditory cues can serve as an alternative way of presenting warnings or other content signaled by color only.

By enabling high contrast support, you can allow color-blind users and users with low vision to adjust default system colors and fonts to make areas of a window or Web page easier to distinguish. Users with low vision also use hardware or software magnifiers to enlarge the pixels on a display, and they depend on alternate text to get some of the information presented in images.

Hearing impairments

Users who are deaf or hard of hearing require visual representations of auditory information. You might need to provide alternate visual cues in your application for audible warnings, for example. Blinking text is one alternative, though the blink rate must be within a certain range to avoid causing problems for users with seizure disorders. Audio tracks require transcripts, and videos might require closed captioning.

Technology to assist with hearing impairments includes voice recognition products that can convert auditory information to text or sign language. Important also are TTY/TDD modems that connect computers with telephones and convert typed ASCII text output to Baudot code, which is what deaf individuals commonly use to communicate over the telephone.

Limited mobility

Users with limited mobility often have difficulty handling hardware and media, but input is typically their biggest challenge. Depending on the disability, mobility-impaired users might need to use voice recognition or an on-screen keyboard with an electronic switch, tracking ball, or joy stick. They might enter input at a slower pace, which means that timers and response times should be adjustable. Systems with built-in intelligence can provide cues to cut down the amount of input required. For Windows applications, the FilterKeys feature is available to slow the keyboard repeat rate, and the Windows StickyKeys feature allows users to enter multiple keystrokes such as Ctrl/Alt/Delete as key sequences.

Cognitive impairments

Reading difficulties, an inability to process visual or auditory information, problems with text input, and short-term memory problems can all affect a user’s access to the content of software and Web applications. Use of clear, simple language, enforcement of consistent design, and presentation of the same information in redundant format, such as both audio and video, can all help users with cognitive impairments to access information. Providing adjustable response times is important to those whose comprehension is slower than normal. Making content available to screen readers to reinforce visual representation is another strategy for aiding comprehension of people with cognitive impairments.

General suggestions

For Web display, it is important to use elements for all markup instead of manipulating text features such as font size directly. Visual appearance should not be the only indicator of function for text elements. Element markup allows assistive technologies such as screen readers to announce text elements such as headings by their function.

Good design for accessibility benefits not only those with disabilities, but users in general. By enforcing a consistent interface design, using simple language, ensuring ease of navigation, and providing the same information in a variety of ways, you can make your applications more usable for everyone.

For more information

For general information about making Web sites accessible, see the World Wide Web Consortium Web site at http://www.w3.org/ and the Utah State University WebAim Web site at http://www.webaim.org.

For information on how your users can adjust various browsers for better legibility, and for ways to accommodate vision impairments in general, see the Lighthouse International Web site at http://www.lighthouse.org/.

Accessibility requirements for software and Web applications

Organizations that want to make their applications accessible to the disabled might have to comply with several sets of slightly different regulations and guidelines, depending on the countries in which their products will be sold or used.

Section 508

Section 508, enacted in 1998, is an extension of the U.S. Government's Rehabilitation Act. Section 508 requires that all electronic and information technology that U.S. Government agencies develop, procure, maintain, and use must be accessible to members of the general public who have disabilities. Many individual states in the U.S. have adopted these requirements as well. Organizations that offer software applications for sale to the U.S. Federal government and many state governments, as well as companies that use or sell accessibility aids, must comply with these regulations to ensure that their products qualify for purchase.

WCAG 1.0

The Section 508 guidelines are based on the accessibility guidelines published in May 1999 by the World Wide Web Consortium. These are known as the Web Content Accessibility Guidelines (WCAG) version 1.0. The WCAG 1.0 is the common basis for most accessibility guidelines and the standard for government enforcement of regulations in many countries today. These guidelines have three priority levels. Priority 1 deals with features essential for access to Web content; Priority 2 defines practices that make Web sites more usable and comprehensible in general, and especially to those using accessibility tools; Priority 3 describes enhanced usability features that make use of the newest technology.

Section 508 includes most of the Priority 1 WCAG recommendations, several from Priorities 2 and 3, and also a few other requirements that are not in the WCAG. The WCAG recommends that organizations strive to meet the Priority 1 and 2 guidelines.

French legislation

The French government has also enacted legislation requiring Web accessibility for those with disabilities and published criteria for conformance called AccessiWeb. AccessiWeb includes three levels, Bronze, Silver, and Gold, that correspond roughly to the three priority levels of the WCAG, but AccessiWeb promotes many level 2 and 3 requirements to higher levels and includes more detail than some of the WCAG recommendations.

U.K. legislation

The United Kingdom has passed legislation called the Disability Discrimination Act that requires Web sites targeting British residents to be accessible to those with disabilities. Enforcement of the U.K. law currently is based on the WCAG 1.0 Priority 1 and 2 guidelines.

Other countries

Many other countries have enacted legislation requiring government or general-use Web sites to be accessible to the disabled. Several of these countries explicitly require compliance with Priorities 1 and 2 of the WCAG 1.0, but a few require only Priority 1 compliance. Many other countries without legislated requirements use the WCAG standards in practice.

WCAG 2.0

The WCAG standards are currently being updated with the intention that they will become a universally accepted set of international guidelines for Web accessibility. WCAG 2.0 will focus on general principles that set out the characteristics Web sites must have to be accessible to users with disabilities. Separate documents will spell out the technical requirements so that these can be updated easily as technology changes without requiring updates to the general principles.

For more information

For information about the accessibility requirements of the U.S. Federal Government for software applications and Web sites, see the Guide to the Section 508 Standards for Electronic and Information Technology Accessibility Standards at http://www.access-board.gov/sec508/guide/ and the standard at http://www.access-board.gov/sec508/508standards.htm.

For the generally accepted international recommendations for Web accessibility, see the WCAG guidelines at http://www.w3.org/TR/WCAG10/. For the new guidelines under development, see the WCAG 2.0 guidelines at http://www.w3.org/TR/WCAG20/.

For the Web accessibility criteria adopted by the French government, see the AccessiWeb criteria at http://www.accessiweb.org/fr/Label_Accessibilite/criteres_accessiweb/92_accessiweb_lineaire/.

Creating accessible software applications with PowerBuilder

MSAA standard

PowerBuilder provides the infrastructure and properties needed to build accessibility features into your Windows and Web applications. Its features allow applications to conform generally to Microsoft Active Accessibility (MSAA) Version 2. MSAA is a Windows standard that defines the way accessibility aids obtain information about user interface elements and the way programs expose information to the aids.

PowerBuilder standard controls support all required Microsoft Active Accessibility properties as listed in the following table:

Table 36-1: MSAA properties and PowerBuilder support

Microsoft Active Accessibility property	PowerBuilder property support
Name	objectname.AccessibleName Some controls support the Name setting through the Text or Title property. For all controls, Name is customizable through the AccessibleName property.
Role	objectname.AccessibleRole Customizable through the AccessibleRole property.
State	Default Active Accessibility support
Location	Default Active Accessibility support
Parent	Default Active Accessibility support
ChildCount	Default Active Accessibility support
Keyboard Shortcut	Default Active Accessibility support for “&” access key of the Text property Also, PowerBuilder Accelerator property setting if applicable to the control.
DefaultAction	Default Active Accessibility support (For example, a selected check box has a default action of uncheck.)
Value	Default Active Accessibility support (For example, a selected check box has the value checked.)
Children	Default Active Accessibility support (For example, items in a list box.)
Focus	Default Active Accessibility support
Selection	Default Active Accessibility support
Description	objectname.AccessibleDescription Customizable through the AccessibleDescription property.
Help	Not supported
HelpTopic	Not supported

Visual controls

For PowerBuilder visual controls that inherit from DragObject, you can manipulate the IAccessible Name, Role, and Description properties of each control by using PowerBuilder dot notation or the Other page in the Properties view of the painters. You can also manipulate the IAccessible property KeyboardShortcut using PowerBuilder properties wherever the ampersand in text property and accelerator property are supported. Other IAccessible properties are set automatically using Active Accessibility default support. (For example, location is automatically updated with absolute screen coordinates for Windows controls at runtime.)

The following table lists PowerBuilder visual controls that inherit from DragObject and their default accessible roles:

Table 36-2: PowerBuilder visual controls and their default roles

PowerBuilder visual controls	AccessibleRole enumerated value
Animation	animationrole!
CheckBox	checkbuttonrole!
CommandButton	pushbuttonrole!
DataWindow	clientrole!
DropDownListBox	comboboxrole!
DropDownPictureListBox	comboboxrole!
EditMask	textrole!
Graph	diagramrole!
GroupBox	groupingrole!
HProgressBar, VProgressBar	progressbarrole!
HScrollBar, VScrollBar	scrollbarrole!
HTrackBar, VTrackBar	sliderrole!
ListBox	listrole!
ListView	listrole!
MonthCalendar	clientrole!
MultiLineEdit	textrole!
Picture	graphicrole!
PictureButton	pushbuttonrole!
PictureHyperLink	linkrole!
PictureListBox	listrole!
RadioButton	radiobuttonrole!
RichTextEdit	clientrole!
SingleLineEdit	textrole!
StaticHyperLink	linkrole!
StaticText	statictextrole!

PowerBuilder visual controls	AccessibleRole enumerated value
Tab control	clientrole!
Tab page	clientrole!
TreeView	outlinerole!

The OLEControl control is set to pushbuttonrole! by default. You need to set this role depending on content.

DataWindow control

PowerBuilder implements the MSAA standard for the DataWindow custom control and its children.

The AccessibleName and AccessibleDescription properties take string values. The AccessibleRole property takes the value of the AccessibleRole enumerated variable.

There are some limitations regarding accessibility support in the DataWindow:

- For the navigation function accNavigate, spatial navigation (navigation by keyboard based on screen location) is not supported. Logical navigation, where keyboard navigation follows a logical tab sequence, is supported only for columns in the detail band. Columns that have a tab value set to 0 so that users cannot update them cannot be accessed from the keyboard.
- The Label, N-Up, OLE 2.0, and RichText DataWindow styles are not supported.
- Support for OLE objects, OLE database columns, and nested reports in DataWindows is limited.

PowerBuilder cannot provide accessibility for control content. This must be provided by the control vendor.

Examples

The following statements set the IAccessible properties for a command button in a Window:

```
cb_1.accessiblename = "Delete"
cb_1.accessibledescription = "Deletes selected text"
cb_1.accessiblerole = pushbuttonrole!
```

The following statement sets the AccessibleName property of a button in a DataWindow object:

```
dw_1.Object.b_1.accessiblename = "Update"
```

The following statements set the AccessibleRole property for a button in a DataWindow object to 43 (the number associated with PushButtonRole!) and return the property to a string variable:

```
string ls_data
```

```
dw_1.Object.b_1.AccessibleRole = 43
ls_data = dw_1.Describe("b_1.AccessibleRole")
```

Deployment

When you deploy an accessible application, you must deploy the *pbacc110.dll* file.

For more information

For more information, see the Microsoft general accessibility Web site at http://www.microsoft.com/enable and the MSDN library site at http://msdn.microsoft.com/library/default.asp?url=/nhp/default.asp?contentid=28000544. Also helpful is the WebAim Web site at http://www.webaim.org.

About VPATs

A Voluntary Product Accessibility Template (VPAT) is a table designed to help U.S. Federal officials make preliminary assessments of accessibility compliance for products offered to the government for sale. A VPAT lists the criteria for compliance with accessibility requirements for various types of products and provides columns where you can indicate and comment on how your product meets them.

VPATs are available for software applications and operating systems, Web-based Internet information and applications, and other types of products. Even if you do not need to fill out a VPAT, reviewing the template for your type of product can give you a clearer understanding of the requirements of Section 508 for software and Web applications.

For more information

To view the various VPATs, see the Information Technology Industry Council Web site at http://www.itic.org/policy/vpat.html.

To see samples of completed VPATs for IT products, see the Sybase accessibility site at http://www.sybase.com/accessibility.

Testing product accessibility

The MSAA 2.0 Software Development Kit (SDK) includes several tools for verifying the MSAA compliance of your application. They include AccExplorer, Accessible Event Watcher, and Object Inspector. These tools are available on the

Microsoft Web site at http://msdn.microsoft.com/library/default.asp?url=/downloads/list/accessibility.asp.

To test the user experience of your application for those with disabilities directly, you can use various methods. For example, try using a text-only browser; enter input using only the keyboard; use the application with a screen reader such as JAWS, Window-Eyes, Hal, or Supernova.

Several commercial applications are also available for testing Web sites for compliance with Section 508 and the WCAG 1.0.

For more information

For a checklist for testing WCAG 1.0 compliance, see the appendix to the WCAG 1.0 on the W3C Web site at http://www.w3.org/TR/1999/WAI-WEBCONTENT-19990505/full-checklist. The W3C Web site also lists and evaluates tools for testing accessibility.

CHAPTER 37 Printing from an Application

About this chapter

This chapter describes how to use predefined functions to create printed lists and reports.

Contents

Printing functions

PowerScript provides predefined functions that you can use to generate simple and complex lists and reports. Using only three functions, you can create a tabular report in your printer's default font. Using additional functions, you can create a report with multiple text fonts, character sizes, and styles, as well as lines and pictures.

Table 37-1 lists the functions for printing.

Table 37-1: PowerScript printing functions

Function	Description
Print	There are five Print function formats. You can specify a tab in all but two formats, and in one you can specify two tabs.
PrintBitMap	Prints the specified bitmap.
PrintCancel	Cancels the specified print job.
PrintClose	Sends the current page of a print job to the printer (or spooler) and closes the print job.
PrintDataWindow	Prints the specified DataWindow as a print job.
PrintDefineFont	Defines one of the eight fonts available for a print job.

Function	Description
PrintGetPrinter	Gets the current printer name.
PrintGetPrinters	Gets the list of available printers.
PrintLine	Prints a line of a specified thickness at a specified location.
PrintOpen	Starts the print job and assigns it a print job number.
PrintOval	Prints an oval (or circle) of a specified size at a specified location.
PrintPage	Causes the current page to print and sets up a new blank page.
PrintRect	Prints a rectangle of a specified size at a specified location.
PrintRoundRect	Prints a round rectangle of a specified size at a specified location.
PrintScreen	Prints the screen image as part of a print job.
PrintSend	Sends a specified string directly to the printer.
PrintSetFont	Sets the current font to one of the defined fonts for the current job.
PrintSetPrinter	Sets the printer to use for the next print function call. This function does not affect open jobs.
PrintSetSpacing	Sets a spacing factor to determine the space between lines.
PrintSetup	Calls the printer Setup dialog box and stores the user’s responses in the print driver.
PrintSetupPrinter	Displays the printer setup dialog box.
PrintText	Prints the specified text string at a specified location.
PrintWidth	Returns the width (in thousandths of an inch) of the specified string in the current font of the current print job.
PrintX	Returns the x value of the print cursor.
PrintY	Returns the y value of the print cursor.

For more information about printing functions, see the *PowerScript Reference*.

Printing basics

All printing is defined in terms of the print area. The print area is the physical page size less any margins. For example, if the page size is 8.5 inches by 11 inches, and the top, bottom, and side margins are all a half-inch, the print area is 7.5 inches by 10 inches.

Measurements

All measurements in the print area are in thousandths of an inch. For example, if the print area is 7.5 inches by 10 inches, then:

The upper-left corner is 0,0
The upper-right corner is 7500,0
The lower-left corner is 0,10000
The lower-right corner is 7500,10000

Print cursor

When printing, PowerBuilder uses a print cursor to keep track of the print location. The print cursor stores the coordinates of the upper-left corner of the location at which printing begins. PowerBuilder updates the print cursor (including tab position if required) after each print operation except PrintBitmap, PrintLine, PrintRectangle, or PrintRoundRect. To position text, objects, lines, and pictures when you are creating complex reports, specify the cursor position as part of each print function call.

Printing a job

PrintOpen must be the first function call in every print job. The PrintOpen function defines a new blank page in memory, specifies that all printing be done in the printer's default font, and returns an integer. The integer is the print job number that is used to identify the job in all other function calls.

PrintOpen is followed by calls to one or more other printing functions, and then the job is ended with a PrintClose (or PrintCancel) call. The functions you call between the PrintOpen call and the PrintClose call can be simple print functions that print a string with or without tabs, or more complex functions that add lines and objects to the report or even include a picture in the report.

Printing titles
To print a title at the top of each page, keep count of the number of lines printed, and when the count reaches a certain number (such as 50), call the PrintPage function, reset the counter, and print the title.

Here is a simple print request:

```
Int PrintJobNumber
// Start the print job and set PrintJobNumber to
// the integer returned by PrintOpen.
PrintJobNumber = PrintOpen()
// Print the string Atlanta.
Print(PrintJobNumber,"Atlanta")
// Close the job.
PrintClose(PrintJobNumber)
```

Using tabs

The Print function has several formats. The format shown in the previous example prints a string starting at the left edge of the print area and then prints a new line. In other formats of the Print function, you can use tabbing to specify the print cursor position before or after printing, or both.

Specifying tab values

Tab values are specified in thousandths of an inch and are relative to the left edge of the print area. If a tab value precedes the string in the Print call and no tab value follows the string, PowerBuilder tabs, prints, then starts a new line. If a tab value follows the string, PowerBuilder tabs after printing and does not start a new line; it waits for the next statement.

In these examples, Job is the integer print job number.

This statement tabs one inch from the left edge of the print area, prints Atlanta, and starts a new line:

```
Print(Job,1000,"Atlanta")
```

This statement prints Boston at the current print position, tabs three inches from the left edge of the print area, and waits for the next statement:

```
Print(Job,"Boston",3000)
```

This statement tabs one inch from the edge of the print area, prints Boston, tabs three inches from the left edge of the print area, and waits for the next statement:

```
Print(Job,1000,"Boston",3000)
```

Tabbing and the print cursor

When PowerBuilder tabs, it sets the x coordinate of the print cursor to a larger print cursor value (a specified value or the current cursor position). Therefore, if the specified value is less than the current x coordinate of the print cursor, the cursor does not move.

The first Print statement shown below tabs one inch from the left edge of the print area and prints `Sybase`, but it does not move to the next tab. (0.5 inches from the left edge of the print area is less than the current cursor position.) Since a tab was specified as the last argument, the first Print statement does not start a new line even though the tab was ignored. The next Print statement prints `Inc.` immediately after the `e` in `Sybase` (`Sybase Inc.`) and then starts a new line:

```
Print(Job,1000,"Sybase",500)
Print(Job," Inc.")
```

Stopping a print job

There are two ways to stop a print job. The normal way is to close the job by calling the PrintClose function at the end of the print job. The other way is to cancel the job by calling PrintCancel.

Using PrintClose

PrintClose sends the current page to the printer or spooler, closes the print job, and activates the window from which the printing started. After you execute a PrintClose function call, any function calls that refer to the job number fail.

Using PrintCancel

PrintCancel ends the print job and deletes any output that has not been printed. The PrintCancel function provides a way for the user to cancel printing before the process is complete. A common way to use PrintCancel is to define a global variable and then check the variable periodically while processing the print job.

Assume *StopPrint* is a boolean global variable. The following statements check the *StopPrint* global variable and cancel the job when the value of *StopPrint* is TRUE:

```
IntJobNbr
JobNbr = PrintOpen()
//Set the initial value of the global variable.
StopPrint = FALSE
//Perform some print processing.
```

```
Do While ...
.
.
.
// Test the global variable.
// Cancel the print job if the variable is TRUE.
// Stop executing the script.
      If StopPrint then
      PrintCancel(JobNbr)
      Return
      End If
Loop
```

Advanced printing techniques

Creating complex reports in PowerBuilder requires the use of additional functions but is relatively easy. You can use PowerScript functions to define fonts for a job, specify fonts and line spacing, place objects on a page, and specify exactly where you want the text or object to be placed.

Defining and setting fonts

The examples so far have used the default font for the printer. However, you can define as many as eight fonts for each print job and then switch among them during the job.

In addition, you can redefine the fonts as often as you want during the print job. This allows you to use as many fonts as you have available on your printer during a print job. Since there is a slight performance penalty for redefining fonts, you should define the fonts after the PrintOpen call and leave them unchanged for the duration of the print job.

To define a font, set an integer variable to the value returned by a call to the PrintDefineFont function and then use the PrintSetFont function to change the font in the job.

Example Assume that JobNum is the integer print job number and that the current printer has a font named Helv. The following statements define Helv18BU as the Helv font, 18 point bold and underlined. The definition is stored as font 2 for JobNum. The company name is printed in font 2:

```
IntJob, Helv18BU
JobNum = PrintOpen()
Helv18BU = PrintDefineFont(JobNum,2,"Helv",250,700, &
      Variable!,Swiss!,FALSE,TRUE)
```

```
PrintSetFont(JobNum,2)
Print(JobNum,"Sybase, Inc.")
```

For more information about PrintDefineFont and PrintSetFont, see the *PowerScript Reference*.

Setting line spacing

PowerBuilder takes care of line spacing automatically when you use the Print function. For example, after you print in an 18-point font and start a new line, PowerBuilder adds 1.2 times the character height to the Y coordinate of the print cursor.

The spacing factor 1.2 is not fixed. You can use the PrintSetSpacing function to control the amount of space between lines.

Examples This statement results in tight single-line spacing. (Depending on the font and the printer, the bottoms of the lowest characters may touch the tops of the tallest characters):

```
PrintSetSpacing(JobNum,1)
```

This statement causes one-and-a-half-line spacing:

```
PrintSetSpacing(JobNum,1.5)
```

This statement causes double spacing:

```
PrintSetSpacing(JobNum,2)
```

Printing drawing objects

You can use the following drawing objects in a print job.

- Lines
- Rectangles
- Round rectangles
- Ovals
- Pictures

When you place drawing objects in a print job, place the objects first and then add the text. For example, you should draw a rectangle inside the print area and then add lines and text inside the rectangle. Although the objects appear as outlines, they are actually filled (contain white space); if you place an object over text or another object, it hides the text or object.

Be careful: PowerBuilder does not check to make sure that you have placed all the text and objects within the print area. PowerBuilder simply does not print anything that is outside the print area.

Example These statements draw a 1-inch by 3-inch rectangle and then print the company address in the rectangle. The rectangle is at the top of the page and centered:

```
IntJob
JobNum = PrintOpen()
PrintRect(JobNum,2500,0,3000,1000,40)
Print(JobNum,2525,"")

Print(JobNum,2525,"25 Mountain Road")
Print(JobNum,2525,"Milton, MA 02186")
PrintClose(JobNum)
```

CHAPTER 38 Managing Initialization Files and the Windows Registry

About this chapter

This chapter describes how to manage preferences and default settings for PowerBuilder applications.

Contents

About preferences and default settings

Many PowerBuilder applications store user preferences and default settings across sessions. For example, many applications keep track of settings that control the appearance and behavior of the application, or store default parameters for connecting to the database. PowerBuilder applications can manage this kind of information in initialization files or in the Windows registry.

Database connection parameters

Often you need to set the values of the Transaction object from an external file. For example, you might want to retrieve values from your PowerBuilder initialization file when you are developing the application or from an application-specific initialization file when you distribute the application.

For information about database connection parameters in an initialization file, see “Reading values from an external file” on page 165.

For an example of how to save and restore database connection parameters in the Windows registry, see “Managing information in the Windows registry” on page 729.

Toolbar settings

PowerBuilder provides some functions you can use to retrieve information about your toolbar settings and also modify these settings. By using these functions, you can save and restore the current toolbar settings.

For more information, see “Saving and restoring toolbar settings” on page 72.

Other settings you may want to save

In addition to the database connection parameters and toolbar settings, you may want to store a variety of other application-specific settings. For example, you might want to keep track of user preferences for colors, fonts, and other display settings.

Managing information in initialization files

Functions for accessing initialization files

PowerBuilder provides several functions you can use to manage application settings in initialization files.

Table 38-1: PowerBuilder initialization file functions

Function	Description
ProfileInt	Obtains the integer value of a setting in a profile file
ProfileString	Obtains the string value of a setting in a profile file
SetProfileString	Writes a value in a profile file

For complete information about these functions, see the *PowerScript Reference*.

For how to use the ProfileString functions with the registry, see “Functions for accessing initialization files” on page 728.

The format of APP.INI

The examples below manage application information in a profile file called *APP.INI*. This file keeps track of user preferences that control the appearance of the application. It has a Preferences section that stores four color settings:

```
[Preferences]
WindowColor=Silver
BorderColor=Red
BackColor=Black
TextColor=White
```

Reading values

The following script retrieves color settings from the *APP.INI* file:

```
wincolor = ProfileString("app.ini", "Preferences", "WindowColor", "")
brdcolor = ProfileString("app.ini", "Preferences", "BorderColor", "")
bckcolor = ProfileString("app.ini", "Preferences", "BackColor", "")
txtcolor = ProfileString("app.ini", "Preferences", "TextColor", "")
```

Setting values

The following script stores color settings in the *APP.INI* file:

```
SetProfileString("app.ini", "Preferences", "WindowColor", wincolor)
SetProfileString("app.ini", "Preferences", "BorderColor", brdcolor)
SetProfileString("app.ini", "Preferences", "BackColor", bckcolor)
SetProfileString("app.ini", "Preferences", "TextColor", txtcolor)
```

Managing information in the Windows registry

Functions for accessing the Registry

PowerBuilder provides several functions you can use to manage application settings in the Windows registry.

Table 38-2: PowerBuilder registry setting functions

Function	Description
RegistryDelete	Deletes a key or a value in a key in the Windows registry.
RegistryGet	Gets a value from the Windows registry.
RegistryKeys	Obtains a list of the keys that are child items (subkeys) one level below a key in the Windows registry.
RegistrySet	Sets the value for a key and value name in the Windows registry. If the key or value name does not exist, RegistrySet creates a new key or value name.
RegistryValues	Obtains a list of named values associated with a key.

For the complete information for these functions, see the *PowerScript Reference*.

Overriding initialization files

You can use the ProfileString functions to obtain information from the registry instead of from an initialization file. Create a new key called *INIFILEMAPPING* at the following location:

HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion

To override the *WIN.INI* file, create a subkey in *INIFILEMAPPING* called *WIN.INI* with the following value:

```
#usr:software\microsoft\windows\currentversion\extensions
```

The examples that follow use the registry to keep track of database connection parameters. The connection parameters are maintained in the registry in the *MyCo\MyApp\database* branch under *HKEY_CURRENT_USER\Software*.

Reading values from the registry

The following script retrieves values for the default Transaction object from the registry.

```
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
   "dbms", sqlca.DBMS)
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "database", sqlca.database)
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "userid", sqlca.userid)
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "dbpass", sqlca.dbpass)
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "logid", sqlca.logid)
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "logpass", sqlca.logpass)
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    “servername", sqlca.servername)
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "dbparm", sqlca.dbparm)
```

Setting values in the registry

The following script stores the values for the Transaction object in the registry:

```
RegistrySet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "dbms", sqlca.DBMS)
RegistrySet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "database", sqlca.database)
RegistrySet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "userid", sqlca.userid)
RegistrySet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "dbpass", sqlca.dbpass)
RegistrySet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "logid", sqlca.logid)
RegistrySet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "logpass", sqlca.logpass)
RegistrySet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "servername", sqlca.servername)
RegistrySet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "dbparm", sqlca.dbparm)
```

CHAPTER 39

Building InfoMaker Styles and Actions

About this chapter

This chapter explains how to build styles in PowerBuilder and provide them to InfoMaker users.

Contents

About form styles

InfoMaker comes with built-in form styles with which users can build sophisticated forms. You can create your own form styles in PowerBuilder and provide them to InfoMaker users. With these custom form styles, you can enforce certain standards in your forms and provide extra functionality to your InfoMaker users. For example, you might want to:

- Include your organization’s logo in each form

  You can do this by creating custom form styles that have the logo in place.

- Reconfigure the toolbar that is provided with the built-in form styles

  You can do this by modifying a built-in form style and saving it as a custom form style.

- Use drag and drop in forms
- Include picture buttons, edit controls, and other controls in forms

Almost anything you can do in a PowerBuilder window you can do in a custom form style.

What a form style is

InfoMaker users use forms to maintain data. Users can view, add, delete, and update data in a form. Each form is based on a form style, which specifies:

- The way the data is presented (for example, in a freeform, grid, or master/detail presentation)
- The menu and toolbar that are available when users run a form
- Actions that users can attach to command buttons in the form

How form styles are constructed

You build form styles in PowerBuilder. A form style consists of:

- A window
- A menu

Figure 39-1: PowerBuilder form style

+
=
Window
Menu
Form style

About the window The window serves as the foundation of the form. It contains one or more DataWindow controls with special names. It is these DataWindow controls that are the heart of the form style. The user views and changes data in the form through the special DataWindow controls.

This chapter refers to the special DataWindow controls as the **central DataWindow controls**. You must name the central DataWindow controls using one of a set of supported names.

In addition to the central DataWindow controls, the window can contain any other controls that you can place in a window in PowerBuilder (such as CommandButtons, RadioButtons, user objects, and pictures).

About the menu When users run forms, they can pick items off a menu. You build the menu in the Menu painter and associate it with the window that the form style is based on.

When building the menu, you can specify which menu items should display in a toolbar when a form is run. The toolbar works like all PowerBuilder toolbars.

About actions Form styles contain actions that users can attach to command buttons in the form and that you can call in scripts.

Each public window function you define in the window for the form style is available as an action to users of the form style.

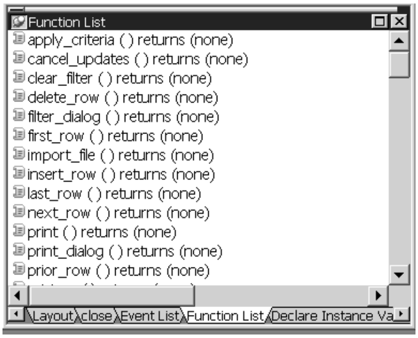
Looking at an example

For example, the built-in form style Freeform consists of:

- A window named w_pbstyle_freeform
- A menu named m_pbstyle_freeform

About w_pbstyle_freeform The window w_pbstyle_freeform contains a DataWindow control named dw_freeform and contains no other controls.

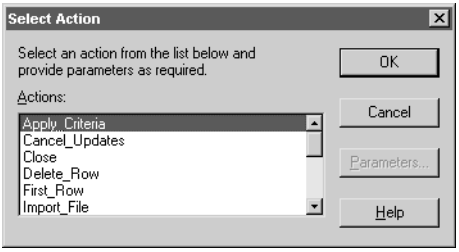
The PowerBuilder window defines many window-level functions:



Function List

```
apply_criteria ( ) returns (none)
cancel_updates ( ) returns (none)
clear_filter ( ) returns (none)
delete_row ( ) returns (none)
filter_dialog ( ) returns (none)
first_row ( ) returns (none)
import_file ( ) returns (none)
insert_row ( ) returns (none)
last_row ( ) returns (none)
next_row ( ) returns (none)
print ( ) returns (none)
print_dialog ( ) returns (none)
prior_row ( ) returns (none)
```

Each of these window functions is available as an action in InfoMaker to users of the Freeform form style:



Select Action

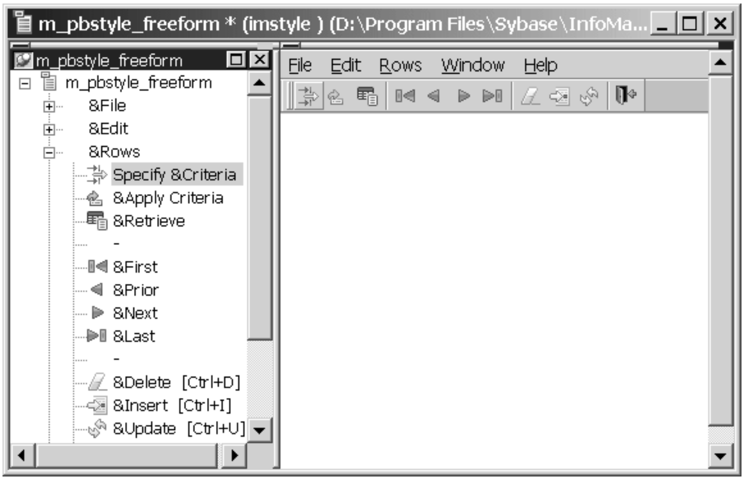
Select an action from the list below and provide parameters as required.

Actions:

Apply_Criteria
Cancel_Updates
Close
Delete_Row
First_Row
Import_File

About m_pbstyle_freeform The menu named m_pbstyle_freeform provides the menu items and toolbar items available to users when they run forms based on the Freeform style.

For example, m_pbstyle_freeform contains the item Specify Criteria on the Rows menu; the item also displays on the toolbar:



When InfoMaker users run the form, they can select Specify Criteria to enter selection criteria that are used in retrieving rows in the form.

Naming the DataWindow controls in a form style

Each form style you define contains one or more central DataWindow controls that are based on DataWindow controls in one of the built-in InfoMaker form styles.

The best way to understand the behavior of these DataWindow controls is to build forms in InfoMaker using each of the built-in styles. Then, when you want to build a form style, choose the DataWindow controls from the built-in style that matches the type of presentation you want in your form style.

For example, to create a basic freeform data entry form, base it on dw_freeform, the DataWindow control found in w_pbstyle_freeform.

When building your form style, you must assign one of the following names to the central DataWindow controls:

- dw_freeform
- dw_grid
- dw_master_12many

- dw_detail_12many
- dw_master_many21
- dw_detail_many21

Valid combinations You must use one of the four combinations of DataWindow controls in Table 39-1in a form style.

Table 39-1: PowerBuilder DataWindow controls

Use these DataWindow control names	To base your form style on this built-in style
dw_freeform *only*	Freeform.
dw_grid *and* dw_freeform	Grid. dw_grid is the central DataWindow control; dw_freeform shares the result set and serves as the background, allowing users to place computed fields anywhere in the form.
dw_master_12many *and* dw_detail_12many	Master Detail/One-To-Many.
dw_master_many21 *and* dw_detail_many21	Master Detail/Many-To-One.

Building and using a form style

❖ **To build and use a form style:**

1. Do one of the following:
    - Copy the window and menu from an existing form style to act as your starting point
    - Begin from scratch by creating a new window and placing in it one or two DataWindow controls that have the supported names
2. Save the window with a special comment that indicates that the window serves as the basis for a form style.
3. Enhance the form style by adding controls to the window, modifying the menu, defining window functions to serve as actions, and so on.
4. Copy all objects used in the form style (such as windows, user objects, and menus) to a library that will be defined as a style library for InfoMaker users.

5 Add the style library to the search path for InfoMaker users.

When InfoMaker users create a new form, the form style you defined displays in the New Form dialog box. Users can select the style to build a form based on the style you built.

The rest of this chapter describes these steps.

Modifying an existing style

The easiest way to get started building form styles is to copy an existing form style and work with it. By examining its structure and making small changes, you can quickly understand how form styles work.

❖ **To begin by modifying an existing form style:**

1 Open the Library painter in PowerBuilder.

2 Copy the window and menu that serve as the foundation for a form style to a library that is on your application’s library search path.

Starting from a built-in form style
The windows and menus that serve as the basis for the built-in form styles are in *IMSTYLE110.PBL*, which is shipped with InfoMaker and installed in the InfoMaker 11.0 directory. You can make a copy of this PBL and use it as the basis of your own form styles.

3 Open the window in the Window painter and select File>Save As from the menu bar to save it with a new name.

4 Give the window a new name.

You can use any name you want, except that names of windows that define form styles must be unique across all style libraries that are used by an InfoMaker user.

5 Define a special comment for the window (for instructions, see “Identifying the window as the basis of a style” on page 737).

6 Click OK to save the window.

7 Open the menu in the Menu painter and select File>Save As from the menu bar to save it with a new name.

8 Provide a new name and an optional comment, then click OK to save the menu.

You do not need to provide a comment for the menu, but it is a good idea to identify it as being used in the form style you are building.

9 Enhance the form style (for instructions, see "Completing the style" on page 738).

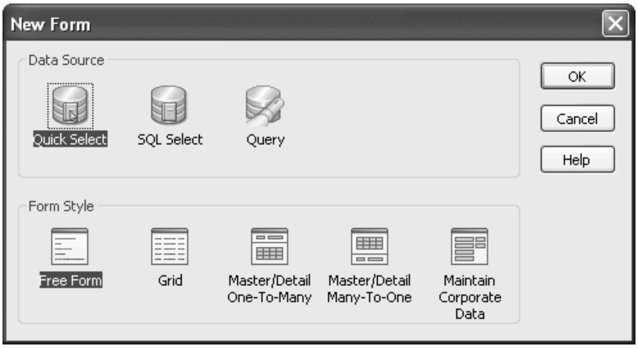
Identifying the window as the basis of a style

In order for InfoMaker to recognize that a window in a library serves as the basis for a form style, you must specify a comment for the window that starts with the text *Style*:

Style: *text that describes the style*

The text that follows *Style*: is the text that displays below the icon for the form style in the New Form dialog box in InfoMaker.

For example, if you save a w_pbstyle_freeform window with the comment `Style: Maintain corporate data` in a style library, InfoMaker users see this when they create a new form:



You can specify the comment either when first saving the window or in the Library painter.

For more information about designing windows, see the PowerBuilder *User's Guide*.

Building a style from scratch

Once you understand how form styles work, you can build one from scratch.

❖ **To build a form style from scratch:**

1. Create a new window.
2. Place a DataWindow control in the window.
3. In the Properties view for the control, name the control using one of the special names.

   For the list of special names, see “Naming the DataWindow controls in a form style” on page 734.
4. Change properties for the control as desired.

   For example, you can add vertical and horizontal scroll bars.

   Do not associate the control with a DataWindow object
   InfoMaker users specify the data for the control when they create a new form.
5. If the form style you are building uses two DataWindow controls, place another DataWindow control in the window and name it to conform with the valid combinations.

   For the list of valid combinations, see “Naming the DataWindow controls in a form style” on page 734.
6. Save the window and specify a comment for it.

   For instructions, see “Identifying the window as the basis of a style” on page 737.

Completing the style

To complete your form style, enhance the window and menu to provide the processing you want. For example, you can:

- Work with the central DataWindow control
- Add controls to the window
- Define actions (functions that appear as actions in your form style)

- Modify the menu and its associated toolbar
- Write scripts for the window, its controls, and menu items
- Add other capabilities, such as drag and drop, to the window

Working with the central DataWindow controls

The DataWindow controls with special names are the heart of a form. It is in these controls that users manipulate the data in the form.

You need to understand:

- How the freeform DataWindow is sized in the form
- How to retrieve data into the control in the form

How the freeform DataWindow is sized

All form styles you build contain a freeform DataWindow (as do all the built-in styles). Regardless of what size you specify for the freeform DataWindow control in the Window painter in PowerBuilder, the freeform DataWindow fills the entire form in the Form painter in InfoMaker. InfoMaker enlarges the freeform DataWindow so that users can place data (such as computed fields) anywhere in the form.

This means that a window background color that you specify in PowerBuilder is ignored in the form.

Retrieving rows into the central DataWindow control

When an InfoMaker user runs a form, InfoMaker automatically populates the SQLCA Transaction object with the correct values, so you do not have to do that in a script. To retrieve rows into the central DataWindow control, all you have to do is set the Transaction object for the control and then retrieve rows.

For example, to retrieve data into the control named dw_freeform, code:

```
dw_freeform.SetTransObject(SQLCA)
dw_freeform.Retrieve()
```

You would code this in the window’s Open event to present the data to the user when the form opens.

For more information about Transaction objects, see Chapter 12, “Using Transaction Objects.”

Adding controls

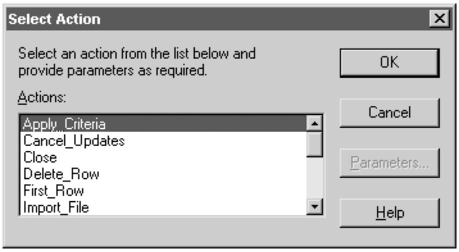
All windows serving as the basis for a form style have at least one DataWindow control. In addition, you can add any other controls that you can add to standard PowerBuilder windows, such as command buttons, user objects, text, edit boxes, pictures, and drawing objects.

Users of the form can move the controls you place in the window, but they cannot delete them.

Users can also add controls to the form in the Form painter. They make CommandButtons and PictureButtons work by associating actions with them. Actions are described next.

Defining actions

Often users want to add buttons (CommandButtons or PictureButtons) to a form created using a custom form style. When you create the form style, you specify what the added buttons can do by defining actions for the form style. When users place a button, they select the desired action from a list:



Select an action from the list below and provide parameters as required.

Actions:

Apply_Criteria
Cancel_Updates
Close
Delete_Row
First_Row
Import_File

Actions are implemented as public window-level functions.

❖ **To define an action:**

1. In the Script view in the Window painter, select Insert>Function from the menu bar.
2. Define the window-level function (for how, see the *PowerBuilder User's Guide*).

   If you want the window function to be available to a form user as an action, be sure to define the function as public. Function arguments you define are used as parameters of the action. Each public window function you define is listed as an action in the Select Action dialog box in the Form painter.

Defining functions not available as actions
If you want to define and use window functions that are not available as actions in forms, define them as private.

Using menus

You specify the menu and toolbar that display when users run a form by defining a menu in the Menu painter and associating it with the window that serves as the basis for your form style.

Each menu item in the menu you define displays when a form is run. In addition, InfoMaker adds Window and Help menus to allow users to manipulate windows and get online Help when running a form in the InfoMaker environment.

Providing online Help
You can define a Help item in the menu bar, then define menu items that display in the Help drop-down menu. The Help items do not display when users run a form within InfoMaker, but they do display when a form is run from an executable. For more information about InfoMaker executable files, see the *InfoMaker User's Guide*.

Item in a toolbar

As with MDI applications, you can specify that a menu item should display as an item in a toolbar when the form is run.

Scripting

You use the same scripting techniques for menus used in forms as you do for menus used in standard windows. Typically you communicate between a window and its menu by defining user events for the window, then triggering them from the menu using the menu object's ParentWindow property to refer to the form window; this technique is used in the built-in form styles.

For more information

For more information about using menus and user events, see the *PowerBuilder User's Guide*.

For more information about associating toolbars with menus, see Chapter 5, "Building an MDI Application."

Writing scripts

You write scripts for the window, its controls, and Menu objects the same way you write them for standard windows and menus. When working with DataWindow controls, remember that you do not have to set the properties of the SQLCA Transaction object—InfoMaker does that automatically when users run a form.

You can define global user-defined functions and structures to support the scripts you code, but note that since InfoMaker does not have an application object, form styles cannot use global variables or global external function declarations.

Adding other capabilities

You can make forms as sophisticated as you want. For example, you can implement drag and drop features, and mail-enable your form.

For complete information about the features you can build into a window, see the *PowerBuilder User's Guide*.

Using the style

Once you complete a form style (or at least have a version that you want to test), you can put it to use.

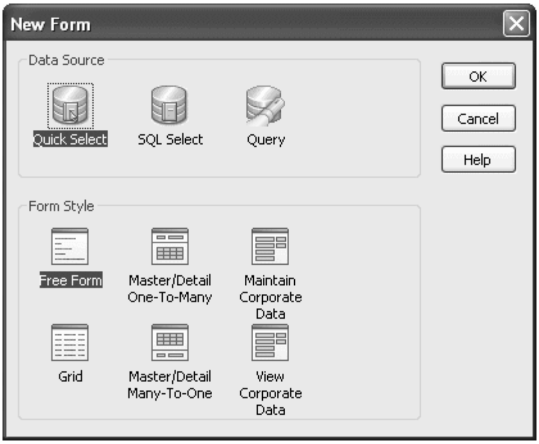
❖ **To make a style available to InfoMaker users:**

1. Make sure the window and menu that define the form style are in a library that is accessible to InfoMaker users (the **style library**).
2. Add any other PowerBuilder objects that you use in the form style (such as windows, user objects, global user-defined functions, and global structures) to the same library.
3. Add the style library to the path for an InfoMaker user.

   For more information, see the *InfoMaker User's Guide*.

Building a form with the custom form style

When an InfoMaker user using the style library creates a new form, all custom form styles display in the Form Style box in the New Form dialog box:



Custom styles display with a generic icon.

InfoMaker users simply select a data source and a custom style to start building a form based on your form style. You should provide documentation to users of your form styles.

Understanding inheritance

When users build a form, they are working with a window that is a descendant of the window that you built for the form *style*. That is, the form style window you built in PowerBuilder is the ancestor, and the form window used in InfoMaker is the descendant. This means that if you change the form style, the changes are picked up the next time users work with a form using that style.

For example, you can add controls to the form style and have the controls display automatically when users later open existing forms using the style.

Caution
Be careful: do not make changes that invalidate forms already built using the style.

Managing the use of form styles

You can store style libraries on the network to make them readily available to all InfoMaker users. You do this with a shared initialization file on a network: you place an InfoMaker initialization file that references the shared style libraries out on the network, then set up InfoMaker users so that they can access the initialization file.

❖ **To make style libraries available throughout your organization:**

1 Place the style libraries on the network in a directory accessible to InfoMaker users.

2 Open InfoMaker, go to the Library painter, and make sure all style libraries are listed in the search path.

3 Close InfoMaker.

4 Copy your InfoMaker initialization file to a directory on the network that is accessible to all InfoMaker users.

This is the shared initialization file. It records all the style libraries in the StyleLib variable in the [Application] section.

5 Set up InfoMaker users so that they can access the shared initialization file.

Each InfoMaker user needs to specify the location of the shared initialization file in InfoMaker.

For more information, see "Specifying the location of the shared InfoMaker initialization file in InfoMaker" next.

Specifying the location of the shared InfoMaker initialization file in InfoMaker

Once the shared initialization file has been defined in a user’s InfoMaker initialization file, the user’s style library search path consists of the style libraries defined in the user’s local InfoMaker initialization file plus all style libraries defined in the shared initialization file. When the user creates a new form, the form styles defined in all the style libraries display in the New Form dialog box.

Each InfoMaker user needs to tell InfoMaker where to find the shared initialization file.

❖ **To specify the location of a shared InfoMaker initialization file:**

1 Select Tools>System Options from the InfoMaker menu bar.

2 On the General property page, enter the path for the shared InfoMaker initialization file.

3 Click OK.

InfoMaker saves the path for InfoMaker initialization in the registry.

Preventing the use of built-in styles

You might not want the built-in form styles to be available to InfoMaker users. That is, you might want all forms to be based on one of your organization's user-defined styles. You can ensure this by suppressing the display of the built-in styles in the New Form dialog box.

❖ **To suppress the display of built-in styles:**

1 Set up a shared initialization file on the network as described in the preceding section.

2 Add this line to the [Window] section of the shared initialization file:

```
ShowStandardStyles = 0
```

With this line specified in the shared initialization file, users can choose only from user-defined form styles when creating a new form. (Note that a ShowStandardStyles line in a user's local InfoMaker initialization file is ignored by InfoMaker.)

PART 9

Deployment Techniques

This part explains how to package your application for deployment and what files you need to deploy.

CHAPTER 40

Packaging an Application for Deployment

About this chapter

This chapter tells you how to prepare a completed executable application for deployment to users.

Contents

About deploying applications

PowerBuilder lets you develop and deploy applications for many application architectures.

Traditional client/server applications

The primary focus of this chapter is on building an executable file and packaging a single- or two-tier application for deployment. The chapter helps you decide whether to use compiled code or pseudocode, whether to use dynamic libraries (PBDs or DLLs) and how to organize them, and whether to deploy resources such as bitmaps and icons separately or use a PowerBuilder Resource file (PBR).

Internet and distributed applications

When you build a client in a multitier application, you need to make many of the same choices as you do for a traditional client/server application. If you are building EAServer or COM components or using the PowerBuilder window plug-in or the PowerBuilder window ActiveX, you need to know about PowerBuilder dynamic libraries (PBDs) and PBRs, described in “Learning what can go in the package” on page 751.

For more information

For detailed information about the files you need to deploy with client/server, multitier, and Web applications, see Chapter 41, “Deploying Applications and Components.”

Creating an executable version of your application

The next few sections tell you more about the packaging process and provide information to help you make choices about the resulting application. They cover:

- Compiler basics
- What can go in the package
- How to choose a packaging model
- How to implement your packaging model
- How to test the executable application you create

Compiler basics

When you plan an application, one of the fundamental topics to think about is the compiler format in which you want that application generated. PowerBuilder offers two alternatives: **Pcode** and **machine code**.

Pcode

Pcode (short for pseudocode) is an interpreted language that is supported on all PowerBuilder platforms. This is the same format that PowerBuilder uses in libraries (PBL files) to store individual objects in an executable state. Advantages of Pcode include its size, reliability, and portability.

Machine code

PowerBuilder generates and compiles code to create a machine code executable or dynamic library. The key advantage of machine code is speed of execution.

PowerBuilder DLLs cannot be called
PowerBuilder machine code DLLs cannot be called from other applications.

Deciding which one to use

Here are some guidelines to help you decide whether Pcode or machine code is right for your project:

- **Speed** If your application does intensive script processing, you might want to consider using machine code. It will perform better than Pcode if your code makes heavy use of looping constructs, floating point or integer arithmetic, or function calls. If your application does not have these characteristics, machine code does not perform noticeably better than Pcode. If you think your application might benefit from the use of machine code, perform some benchmark testing to find out.

Pcode is faster to generate than machine code. Even if you plan to distribute your application using machine code, you might want to use Pcode when you want to quickly create an executable version of an application for testing.

- **Size** The files generated for Pcode are smaller than those generated for machine code. If your application is to be deployed on computers where file size is a major issue, or if you deploy it using a Web download or file transfer, then you might decide to give up the speed of machine code and choose Pcode instead.

Learning what can go in the package

No matter which compiler format you pick, an application that you create in PowerBuilder can consist of one or more of the following pieces:

- An executable file
- Dynamic libraries
- Resources

To decide which of these pieces are required for your particular project, you need to know something about them.

About the executable file

If you are building a single- or two-tier application that you will distribute to users as an executable file, rather than as a server component or a Web application, you always create an executable (EXE) file.

At minimum, the executable file contains code that enables your application to run as a native application on its target platform. That means, for example, that when users want to start your application, they can double-click the executable file's icon on their desktop.

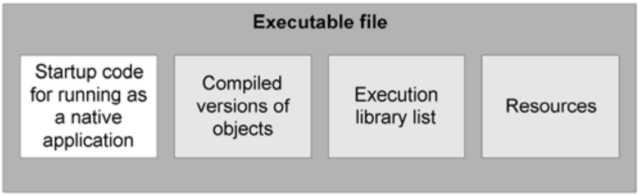
What else can go in the executable file Depending on the packaging model you choose for your application, the executable file also contains one or more of the following:

- *Compiled versions of objects* from your application's libraries

  You can choose to put all of your objects in the executable file so that you have only one file to deliver, or you can choose to split your application into one executable file and one or more dynamic libraries. For more information, see "About dynamic libraries" on page 752.

- *An execution library list* that the PowerBuilder execution system uses to find objects and resources in any dynamic libraries you have packaged for the application
- *Resources* that your application uses (such as bitmaps)

Figure 40-1: Executable file contents


Executable file
Startup code for running as a native application
Compiled versions of objects
Execution library list
Resources

About dynamic libraries

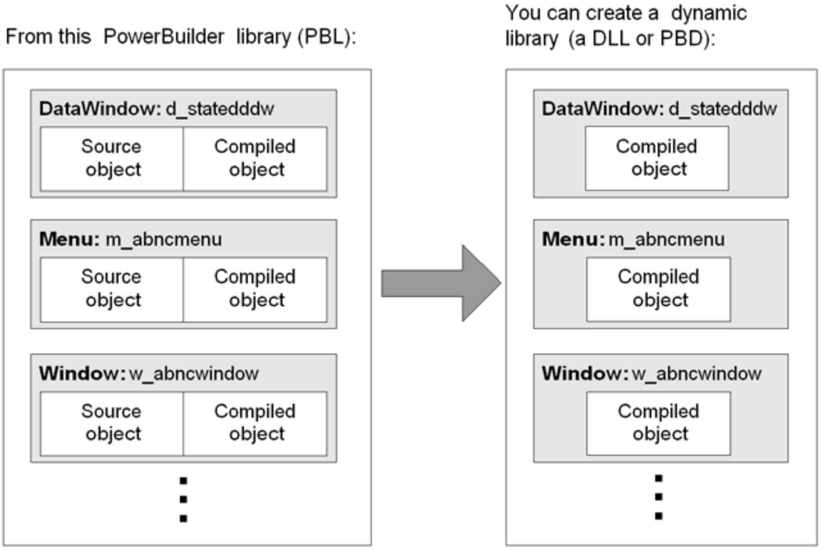
As an alternative to putting your entire application in one large executable file, you can deliver some (or even all) of its objects in one or more dynamic libraries. The way PowerBuilder implements dynamic libraries depends on the compiler format you choose.

Table 40-1: PowerBuilder dynamic libraries

If you are generating	Your dynamic libraries will be
Machine code	DLL files (dynamic link libraries). Machine-code dynamic libraries are given the extension .dll. These dynamic libraries are like any other standard shared libraries in your operating environment. The only caveat is that they are not intended to be called from external programs.
Pcode	PBD files (PowerBuilder dynamic libraries). These dynamic libraries are similar to DLLs in that they are linked to your application at runtime. They are not interchangeable with DLLs, however, because they have a different internal format. You cannot mix the two different kinds of dynamic libraries (DLLs and PBDs) in one application.

As with an executable file, only *compiled* versions of objects (and not their sources) go into dynamic libraries.

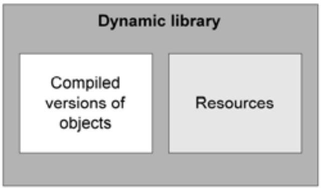
Figure 40-2: Compiled objects in dynamic libraries



What else can go in dynamic libraries Unlike your executable file, dynamic libraries do not include any start-up code. They cannot be executed independently. Instead, they are accessed as an application executes when it cannot find the objects it requires in the executable file.

Dynamic libraries can include resources such as bitmaps. You might want to put any resources needed by a dynamic library's objects in its DLL or PBD file. This makes the dynamic library a self-contained unit that can easily be reused. If performance is your main concern, however, be aware that resources are loaded faster at runtime when they are in the executable file.

Figure 40-3: Resources in dynamic libraries



Why use them Table 40-2 lists several reasons why you might want to use dynamic libraries.

Table 40-2: Reasons to use dynamic libraries

Reason	Details
Modularity	They let you break up your application into smaller, more modular files that are easier to manage.
Maintainability	They enable you to deliver application components separately. To provide users with a bug fix, you can often give them the particular dynamic library that was affected.
Reusability	They make it possible for multiple applications to reuse the same components because dynamic libraries can be shared among applications as well as among users.
Flexibility	They enable you to provide your application with objects that it references only dynamically at runtime (such as a window object referenced only through a string variable). You cannot put such objects in your executable file (unless they are DataWindow objects).
Efficiency	They can help a large application use memory efficiently because: • PowerBuilder does not load an entire dynamic library into memory at once. Instead, it loads individual objects from the dynamic library only when needed. • Your executable file can remain small, making it faster to load and less obtrusive.

Organizing them Once you decide to use a dynamic library, you need to tell PowerBuilder which library (PBL file) to create it from. PowerBuilder then places compiled versions of *all* objects from that PBL file into the DLL or PBD file.

If your application uses only some of those objects, you might not want the dynamic library to include the superfluous ones, which only make the file larger. The solution is to:

1. *Create a new PBL file* and copy only the objects you want into it.
2. *Use this new PBL file* as the source of your dynamic library.

About resources

In addition to PowerBuilder objects such as windows and menus, applications also use various resources. Examples of resources include:

- *Bitmaps* that you might display in Picture or PictureButton controls
- *Custom pointers* that you might assign to windows

When you use resources, you need to deliver them as part of the application along with your PowerBuilder objects.

What kinds there are A PowerBuilder application can employ several different kinds of resources. Table 40-3 lists resources according to the specific objects in which they might be needed.

Table 40-3: PowerBuilder objects and resources

These objects	Can use these kinds of resources
Window objects and user objects	Icons (ICO files) Pictures (BMP, GIF, JPEG, RLE, and WMF files) Pointers (CUR files)
DataWindow objects	Pictures (BMP, GIF, JPEG, RLE, and WMF files)
Menu objects (when in an MDI application)	Pictures (BMP, GIF, JPEG, RLE, and WMF files)

Delivering them When deciding how to package the resources that need to accompany your application, you can choose from the following approaches:

- *Include them in the executable file.*

  Whenever you create an executable file, PowerBuilder automatically examines the objects it places in that file to see if they explicitly reference any resources (icons, pictures, pointers). It then copies all such resources right into the executable file.

  PowerBuilder does not automatically copy in resources that are dynamically referenced (through string variables). To get such resources into the executable file, you must use a **resource (PBR) file**. This is simply a text file in which you list existing ICO, BMP, GIF, JPEG, RLE, WMF, and CUR files.

  Once you have a PBR file, you can tell PowerBuilder to read from it when creating the executable file to determine which additional resources to copy in. (This might even include resources used by the objects in your dynamic libraries, if you decide to put most or all resources in the executable file for performance reasons.)

- *Include them in dynamic libraries.*

  You might often need to include resources directly in one or more dynamic libraries, but PowerBuilder does not automatically copy any resources into a dynamic library that you create even if they are explicitly referenced by objects in that file. You need to produce a PBR file that tells PowerBuilder which resources you want in this particular DLL or PBD file.

Use a different PBR file for each dynamic library in which you want to include resources. (When appropriate, you can even use this approach to generate a dynamic library that contains only resources and no objects. Simply start with an empty PBL file as the source.)

- *Deliver them as separate files.*

  This means that when you deploy the application, you give users various image files in addition to the application's executable file and any dynamic libraries. As long as you do not mind delivering a lot of files, this can be useful if you expect to revise some of them in the future.

  Keep in mind that this is not the fastest approach at runtime, because it requires more searching. Whenever your application needs a resource, it searches the executable file and then the dynamic libraries. If the resource is not found, the application searches for a separate file.

  Make sure that your application can find where these separate files are stored, otherwise it cannot display the corresponding resources.

You can use one of these approaches or any combination of them when packaging a particular application.

Using a PBR file to include a dynamically referenced DataWindow object
You might occasionally want to include a dynamically referenced DataWindow object (one that your application knows about only through a string variable) in the executable file you are creating. To do that, you must list its name in a PBR file along with the names of the resources you want PowerBuilder to copy into that executable file.

You *do not* need to do this when creating a dynamic library, because PowerBuilder automatically includes every DataWindow object from the source library (PBL file) in your new DLL or PBD file.

Creating a PowerBuilder resource file

A PBR file is an ASCII text file in which you list resource names (such as BMP, CUR, ICO, RLE, and WMF files) and DataWindow objects. To create a PBR file, use a text editor. List the name of each resource, one resource on each line, then save the list as a file with the extension PBR. Here is a sample PBR file:

```
ct_graph.ico
document.ico
codes.ico
```

```
button.bmp
next1.bmp
prior1.bmp
```

❖ To create and use a PowerBuilder resource file:

1 Using a text editor, create a text file that lists all resource files referenced dynamically in your application (see below for information about creating the file).

When creating a resource file for a dynamic library, list *all* resources used by the dynamic library, not just those assigned dynamically in a script.

2 Specify the resource files in the Project painter. The executable file can have a resource file attached to it, as can each of the dynamic libraries.

When PowerBuilder builds the project, it includes all resources specified in the PBR file in the executable file or dynamic library. You no longer have to distribute your dynamically assigned resources separately; they are in the application.

Naming resources

If the resource file is in the current directory, you can simply list the file, such as:

```
FROWN.BMP
```

If the resource file is in a different directory, include the path to the file, such as:

```
C:\BITMAPS\FROWN.BMP
```

Paths in PBR files and scripts must match exactly
The file name specified in the PBR file must exactly match the way the resource is referenced in scripts.

If the reference in a script uses a path, you must specify the same path in the PBR file. If the resource file is not qualified with a path in the script, it must not be qualified in the PBR file.

For example, if the script reads:

```
p_logo.PictureName = "FROWN.BMP"
```

then the PBR file must read:

```
FROWN.BMP
```

If the PBR file says something like:

```
C:\MYAPP\FROWN.BMP
```

and the script does not specify the path, PowerBuilder cannot find the resource at runtime. That is because PowerBuilder does a simple string comparison at runtime. In the preceding example, when PowerBuilder executes the script, it looks for the object identified by the string `FROWN.BMP` in the executable file. It cannot find it, because the resource is identified in the executable file as `C:\MYAPP\FROWN.BMP`.

In this case, the picture does not display at runtime; the control is empty in the window.

Including DataWindows objects in a PBR file

To include a DataWindow object in the list, enter the name of the library (with extension PBL) followed by the DataWindow object name enclosed in parentheses. For example:

```
sales.pbl(d_emplist)
```

If the DataWindow library is not in the directory that is current when the executable is built, fully qualify the reference in the PBR file. For example:

```
c:\myapp\sales.pbl(d_emplist)
```

Choosing a packaging model

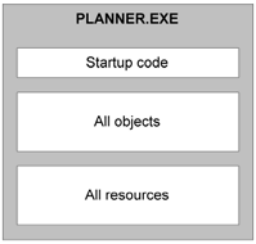
As indicated in the previous section, you have many options for packaging an executable version of an application. Here are several of the most common packaging models you might consider.

A standalone executable file

In this model, you include everything (all objects and resources) in the executable file, so that there is just one file to deliver.

Illustration Figure 40-4 shows a sample of what this model can look like.

Figure 40-4: Standalone executable model



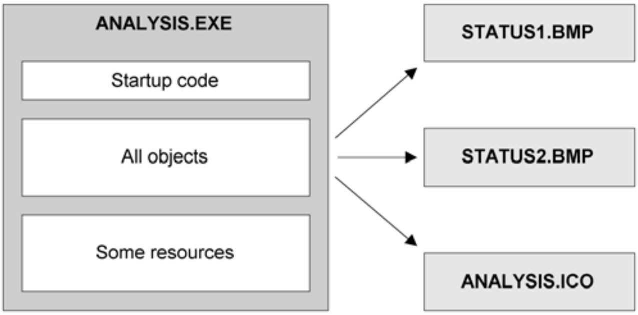
Use This model is good for small, simple applications—especially those that are likely not to need a lot of maintenance. For such projects, this model ensures the best performance and the easiest delivery.

An executable file and external resources

In this model, you include all objects and most resources in the executable file, but you deliver separate files for particular resources.

Illustration Figure 40-5 shows a sample of what this model can look like.

Figure 40-5: Executable with external resources model



Use This model is also for small, simple applications, but it differs from the preceding model in that it facilitates maintenance of resources that are subject to change. In other words, it lets you give users revised copies of specific resources without forcing you to deliver a revised copy of the executable file.

You can also use this model to deal with resources that must be shared by other applications or that are large and infrequently needed.

An executable file and dynamic libraries

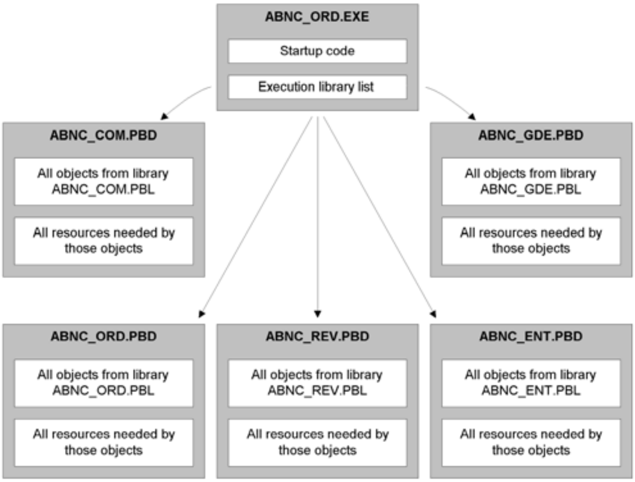
In this model, you split up your application into an executable file and one or more dynamic library files (DLLs or PBDs). When doing so, you can organize your objects and resources in various ways. Table 40-4 shows some of these techniques.

Table 40-4: Object and resource organization with dynamic libraries

To organize	You can
Objects	Place them all in dynamic libraries so that there are none in the executable file, which facilitates maintenance, *or* Place a few of the most frequently accessed ones in the executable file to optimize access to them and place all the rest in dynamic libraries.
Resources	Place most or all of them in dynamic libraries along with the objects that use them, which facilitates reuse, *or* Place most or all of them in the executable file to optimize access to them.

Illustration Figure 40-6 shows a sample of what this model can look like.

Figure 40-6: Executable with dynamic libraries model


ABNC_ORD.EXE
Startup code
Execution library list
ABNC_COM.PBD
All objects from library ABNC_COM.PBL
All resources needed by those objects
ABNC_GDE.PBD
All objects from library ABNC_GDE.PBL
All resources needed by those objects
ABNC_ORD.PBD
All objects from library ABNC_ORD.PBL
All resources needed by those objects
ABNC_REV.PBD
All objects from library ABNC_REV.PBL
All resources needed by those objects
ABNC_ENT.PBD
All objects from library ABNC_ENT.PBL
All resources needed by those objects

Use This model is good for most substantial projects because it gives you flexibility in organizing and maintaining your applications.

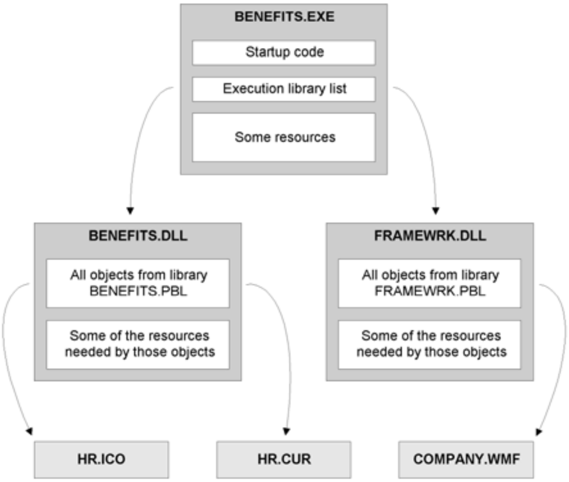
For instance, it enables you to make revisions to a particular part of an application in one dynamic library. However, you must always rebuild the entire application and deliver all the dynamic libraries to customers whenever you make a revision to any library.

An executable file, dynamic libraries, and external resources

This model is just like the preceding one except that you deliver separate files for particular resources (instead of including all of them in your executable file and dynamic libraries).

Illustration Figure 40-7 shows a sample of what this model can look like.

Figure 40-7: Executable with dynamic libraries and external resources model


BENEFITS.EXE
Startup code
Execution library list
Some resources
BENEFITS.DLL
All objects from library BENEFITS.PBL
Some of the resources needed by those objects
FRAMEWRK.DLL
All objects from library FRAMEWRK.PBL
Some of the resources needed by those objects
HR.ICO
HR.CUR
COMPANY.WMF

Use This model is good for substantial applications, particularly those that call for flexibility in handling certain resources. Such flexibility may be needed if a resource:

- Might have to be revised
- Must be shared by other applications
- Is large and infrequently used

Implementing your packaging model

When you have decided which is the appropriate packaging model for your application, you can use the packaging facilities in PowerBuilder to implement it. For the most part, this involves working in the **Project painter**. You can use the Project painter to build components, proxy libraries, and HTML files as well as executable applications.

Using the Project painter for executable applications

The Project painter for executable applications orchestrates all aspects of the packaging job by enabling you to:

- Specify the *executable file* to create
- Specify any *dynamic libraries* (DLL or PBD files) to create

- Specify the *resources you want included* in the executable file or in each particular dynamic library (by using appropriate PBR files that indicate where to get those resources)
- Choose *machine code or Pcode* as the compiler format to generate

  With machine code, you can also specify a variety of code generation options (such as optimization, trace information, and error context information).
- Choose *build options*, including whether you want the Project painter to do a full or incremental rebuild of your application's objects when generating the executable application
- Save all of these specifications as a **project object** that you can use whenever necessary to rebuild the whole package

For more information on using the Project painter, see the *PowerBuilder User's Guide*.

Building individual dynamic libraries

When you make revisions to an existing application, your changes might not affect all its dynamic libraries. You can rebuild individual dynamic libraries from the pop-up menu in the System Tree or the Library painter.

If changes are isolated and do not affect inherited objects in other PBLs, you might be able to distribute individual PBDs to your users to provide an upgrade or bug fix. However, Sybase recommends that you always perform a full rebuild and distribute the executable file and all the application's dynamic libraries whenever you revise an application.

Testing the executable application

Once you create the executable version of your application, test how it runs before proceeding with delivery. You may have already executed the application many times within the PowerBuilder development environment, but it is still very important to run the executable version as an *independent* application—just the way end users will.

To do this, you:

1 Leave PowerBuilder and go to your operating system environment.

2 Make sure that the PowerBuilder runtime libraries are accessible to the application.

You can do this by verifying that the location of the PowerBuilder virtual machine and other runtime files is in your PATH environment variable, or you can create a registry entry for the application that specifies the path.

3 Run the application's executable file as you run any native application.

Tracing the application's execution

To help you track down problems, PowerBuilder provides **tracing and profiling** facilities that you can use in the development environment and when running the executable version of an application. Even if your application's executable is problem free, you might consider using this facility to generate an audit trail of its operation. For more information on tracing execution, see the *PowerBuilder User's Guide*.

Delivering your application to end users

When you deliver the executable version of your application to users, you need to install all of the various files and programs in the right places, such as on their computers or on the network.

Automating the deployment process

If you want to automate the deployment process, you might want to use a software distribution application such as InstallShield. Such applications typically install all the executables, resource files, data sources, and configuration files your users need to run your application. They also update the users' initialization files and registry.

Installation checklist

You can use the following checklist to make sure you install everything that is needed. For easy reading, the checklist is divided into:

- Installing environmental pieces
- Installing application pieces

Installing environmental pieces

Checklist item	Details
Install the PowerBuilder runtime DLLs.	You should install all of these DLL files (which contain the PowerBuilder execution system) locally on each user computer. They are needed to run PowerBuilder applications independently (outside the development environment). This applies to applications generated in machine code as well as those generated in Pcode. For details on installing the runtime DLLs, see "PowerBuilder runtime files" on page 775. *Handling maintenance releases* If you are using a maintenance release of PowerBuilder in your development environment, make sure you provide users with the runtime DLLs from that maintenance release.
Install the database interface(s).	You should install on each user computer any database interfaces required by the application, such as the ODBC interface and other native database interfaces. For details on installing any database interfaces you need, see Chapter 41, "Deploying Applications and Components." For more information about database interfaces, see *Connecting to Your Database*.
Configure any ODBC drivers you install.	If you install the ODBC interface (and one or more ODBC drivers) on user computers, you must also configure the ODBC drivers. This involves defining the specific data sources to be accessed through each driver. For details on configuring ODBC drivers, see *Connecting to Your Database*.
Set up network access if needed.	If the application needs to access any server databases or any other network services, make sure each user computer is properly connected.
Configure the operating (windowing) system.	A particular application might require some special adjustments to the operating or windowing system for performance or other reasons. If that is the case with your application, be sure to make those adjustments to each user computer.

Installing application pieces

Checklist item	Details
Copy the executable application.	Make copies of the files that make up your executable application and install them on each user computer. These files can include: • The executable (EXE) file • Any dynamic libraries (DLL or PBD files) • Any files for resources you are delivering separately (such as ICO, BMP, GIF, JPEG, RLE, WMF, or CUR files) *Handling maintenance releases* If you plan to revise these files on a regular basis, you might want to automate the process of copying the latest versions of them from a server on your network to each user computer. You might consider building this logic right into your application. You might also make it copy updates of the PowerBuilder runtime DLLs to a user’s computer.
Copy any additional files.	Make copies of any additional files that the application uses and install them on each user computer. These files often include: • Initialization (INI) files • Help (HLP) files • Possibly various others such as text or sound files In some cases, you might want to install particular files on a server instead of locally, depending on their use.
Copy any local databases to be accessed.	If the application needs to access a local database, copy the files that comprise that database and install them on each user computer. Make sure that you also install the appropriate database interface and configure it properly if you have not already done so.
Install any other programs to be accessed.	If the application needs to access any external programs, install each one in an appropriate location—either on every user computer or on a server. Also, perform any configuration required to make those programs work properly. For example, you might need to register ActiveX controls. For more information, see “Deploying ActiveX controls” on page 771.

Checklist item	Details
Ensure that the application can find the files it needs.	Make sure you install the various files that your application uses on paths where it can find them: • If the application refers to a file *by a specific path*, then install the file on that path. • If the application refers to a file *by name only*, then install the file on some path that the application is able to search—typically the current one.
Update the system registry with values for the application.	If you rely on the Windows registry to manage certain information needed by the application, such as the application path, be sure to update the registry with such values.
Set up the application's icon.	To enable users to start the application, use the windowing system on each user computer to display the executable file's icon where you want. Alternatively, users can also start the application in any other manner provided for native applications under their windowing system.

Starting the deployed application

Users can run your application just as they run other Windows applications. For example, they can double-click the executable file in Explorer or create an application shortcut on the desktop and double-click the shortcut.

If users create a shortcut, the Target text box on the Shortcut properties page should specify the path to the executable, and the Start In text box should specify the location of the runtime DLLs.

CHAPTER 41

Deploying Applications and Components

About this chapter

This chapter provides the information required to deploy applications and components to users' computers and servers. It describes a tool you can use to package PowerBuilder runtime files, and lists the files you need to deploy with various kinds of targets.

These lists of files sometimes need to be updated, as, for example, when new database interfaces become available. For information about such changes, see the *Release Bulletin* for the version of PowerBuilder you are using.

For information about deploying JSP targets, see *Working with JSP Targets*.

Contents

Deploying applications, components, and supporting files

Regardless of the type of application you are deploying, you must include any supporting files such as dynamic libraries, resources like BMP and ICO files, online Help files, and initialization files. Each application type requires a different set of supporting files. The PowerBuilder runtime files, such as *pbvm110.dll* and *pbdwe110.dll*, and PowerBuilder database interfaces such as *pbsnc110.dll* and *pbo10110.dll*, can be freely distributed with your application with no licensing fee.

Planning for deployment

Chapter 40, “Packaging an Application for Deployment,” helps you make decisions about deploying a PowerBuilder executable application, such as whether to use dynamic libraries, Pcode or machine code, and resource files. It also provides a checklist to make sure you install all the required pieces.

If you are deploying a Web application or a transaction server component, you will find the information about PowerBuilder dynamic libraries (PBDs) and PowerBuilder resource files (PBRs) in that chapter helpful. You should also read the documentation for specific types of applications, components, or plug-ins elsewhere in this book or in the book listed in Table 41-1.

Table 41-1: Additional deployment documentation

Application type	Where to find information
EAServer components	“Deploying a component to EAServer” on page 507
Application server components	The *User’s Guide* in the PowerBuilder Application Server Plug-in documentation set.
COM components	“Deploying a PowerBuilder COM server” on page 576
DataWindow plug-in applications	Chapter 32, “Using the DataWindow Plug-in”
PowerBuilder window plug-in	Chapter 33, “Using the PowerBuilder Window Plug-in”
PowerBuilder window ActiveX	Chapter 34, “Using the PowerBuilder Window ActiveX”
.NET applications and components	*Deploying Applications and Components to .NET*
Web DataWindow and DataWindow Web control for ActiveX	*DataWindow Programmer’s Guide*

Finding information in this chapter

This chapter is intended to help you write installation programs using a third-party software package that creates installation configurations. It tells you which files each computer needs, where you can find the files, where they should be installed, and what registry settings need to be made. PowerBuilder Enterprise also provides a tool, described in “PowerBuilder Runtime Packager” on page 771, to help you package the files your application needs.

Use Table 41-2 to locate information about the specific files you need to deploy with your application.

Table 41-2: PowerBuilder files required for deployment

Scenario	See these sections
All PowerBuilder client applications	“PowerBuilder runtime files” on page 775
PowerBuilder client application accessing data on a database server	“Database connections” on page 777
PowerBuilder clients for EJBs, SOAP Web services, and XML services	“PowerBuilder extensions” on page 788
PowerBuilder clients that save data in PDF or XSL-FO format	“PDF and XSL-FO export” on page 789
Web application using the DataWindow Web control for ActiveX	“DataWindow Web control for ActiveX” on page 792
Web application using PowerBuilder plug-ins or the window ActiveX	“Plug-ins and PowerBuilder window ActiveX controls” on page 793
EAServer component created in PowerBuilder	“PowerBuilder components on EAServer” on page 794
COM component created in PowerBuilder	“PowerBuilder COM servers” on page 796
Web application using Web DataWindow with EAServer	“Files required on the transaction server” on page 799 “Files required on the dynamic page server” on page 799
Web application using Web DataWindow with ASP and COM+ or IIS	“Files required on the COM+ or IIS server” on page 800

Installed and deployment paths

The *Installed path* listed after some of the tables in this chapter is the location where files are installed when you install PowerBuilder and select the default installation location. When you build an installation program for your application, you can copy files from this location to your staging area.

The *Deployment path* tells you where these files can be installed on the computer on which you install your application or component.

App Path registry key

Some tables are followed by a list of the *Registry entries* your installation program needs to make so that your application or component can find the files it needs. When an application runs on Windows, it looks for supporting files in these locations and in this order:

1. The directory where the executable file is installed.
2. The Windows system and Windows directories (for example, in *C:\WINDOWS\system32*, *C:\WINDOWS\system*, and *C:\WINDOWS*).
3. In an application path that can be specified in the registry.
4. In the system path.

You do not need to specify an application path, but it is recommended.

Specifying an application path

To specify the path the application uses to locate supporting files, your installation program should create an App Path key for your application in this registry location:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\
     CurrentVersion\App Paths
```

Set the data value of the (Default) string value to the directory where the application is installed and create a new string value called Path that specifies the location of shared files. The following example shows a typical registry entry for an application called *myapp.exe* that uses SQL Anywhere. The registry key is enclosed in square brackets and is followed by string values for the key in the format `"`*Name*`"="`*Value*`"`:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\
   CurrentVersion\App Paths\myapp.exe]
"Default"="C:\Program Files\myapps\myapp.exe"
"Path"="C:\Program Files\myapps;C:\Program Files\
   sybase\shared\PowerBuilder;c:\program files\
   SQL Anywhere 10\win32\;"
```

About REG files
Registry update files that have a *.REG* extension can be used to import information into the registry. The format used in registry key examples in this chapter is similar to the format used in registry update files, but these examples are *not intended* to be used as update files. The path names in data value strings in registry update files typically use a pair of backslashes instead of a single backslash, and the “Default” string value is represented by the at sign (@).

Use the examples to help determine which registry keys your installation program should add or update.

Deploying ActiveX controls

If your application uses ActiveX controls, OLE controls, or OCX controls, you must:

- Deploy the control files with your application
- Make sure each control is registered
- Make sure required files are in the target computer’s system directory

If your application uses a control that is not self registering, your setup program needs to register it manually on each user’s computer. To find out whether a control is self registering, see the documentation provided with the control. Depending on the development and deployment platforms and the controls you are deploying, you might need to copy additional DLLs or license files to the Windows system directories on the target computer.

PowerBuilder Runtime Packager

The PowerBuilder Runtime Packager is a tool provided with PowerBuilder Enterprise that packages the PowerBuilder files an application needs at runtime into a Microsoft Windows Installer (MSI) package file. Windows Installer is an installation and configuration service that is installed with more recent Microsoft Windows operating systems.

You can use the MSI file generated by the Runtime Packager as part of an installation package that includes the other files that your application needs.

You must have Microsoft Windows Installer on your system in order to run the Runtime Packager successfully. The Installer is always available on Windows XP and Windows 2003.

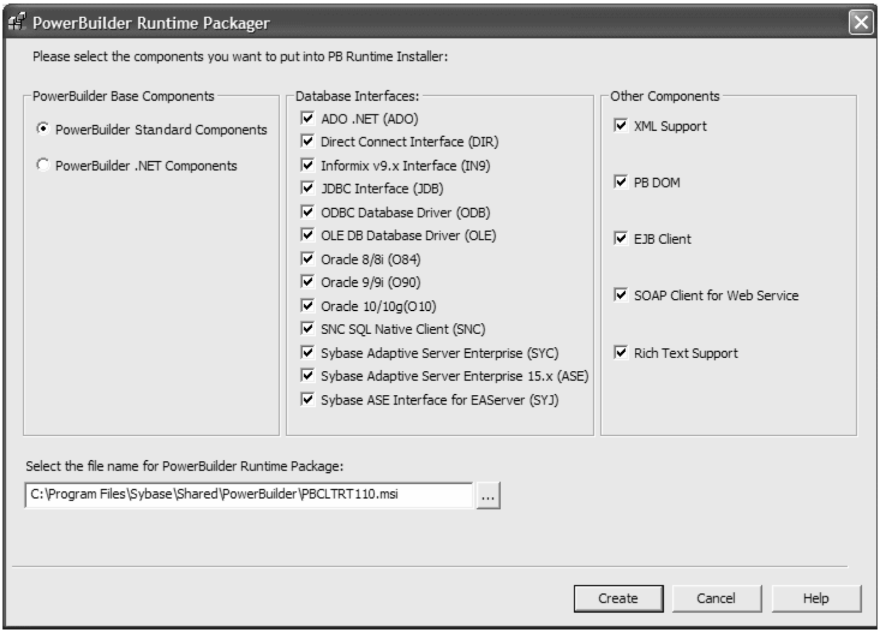
To get more information and to obtain the latest version of Windows Installer, see the Microsoft documentation at http://msdn.microsoft.com.

The Runtime Packager can be used with client applications installed on Windows systems and applications deployed to the .NET Framework. It does not package the files required if your application uses the DataWindow Web control for ActiveX or a plug-in, and it does not install most third-party components. See “Third-party components” on page 775 for more information.

Make sure that you read the sections referenced in Table 41-2 on page 769 that apply to your application for more information about where files that are not installed by the Runtime Packager should be installed.

❖ To use the PowerBuilder Runtime Packager:

1 Select Programs>Sybase>PowerBuilder 11.0>PowerBuilder Runtime Packager from the Windows Start menu or launch the pbpack110 executable file in your *Shared\PowerBuilder* directory.



PowerBuilder Runtime Packager

Please select the components you want to put into PB Runtime Installer:

PowerBuilder Base Components

● PowerBuilder Standard Components
○ PowerBuilder .NET Components

Database Interfaces:

☑ ADO .NET (ADO)
☑ Direct Connect Interface (DIR)
☑ Informix v9.x Interface (IN9)
☑ JDBC Interface (JDB)
☑ ODBC Database Driver (ODB)
☑ OLE DB Database Driver (OLE)
☑ Oracle 8/8i (O84)
☑ Oracle 9/9i (O90)
☑ Oracle 10/10g(O10)
☑ SNC SQL Native Client (SNC)
☑ Sybase Adaptive Server Enterprise (SYC)
☑ Sybase Adaptive Server Enterprise 15.x (ASE)
☑ Sybase ASE Interface for EAServer (SYJ)

Other Components

☑ XML Support
☑ PB DOM
☑ EJB Client
☑ SOAP Client for Web Service
☑ Rich Text Support

Select the file name for PowerBuilder Runtime Package:

C:\Program Files\Sybase\Shared\PowerBuilder\PBCLTRT110.msi

2 Select a location for the generated MSI file.

3 If you are deploying a .NET target, select PowerBuilder .NET Components. Otherwise, select PowerBuilder Standard Components.

4 Select the database interfaces your application requires.

The DLLs for the database interfaces you select are added to the package. For ODBC and OLE DB, the *pbodb110.ini* file is also added. For JDBC, the *pbjdbc12110.jar* and *pbjvm110.dll* files are also added. The Java Runtime Environment (JRE) is *not* added. See "Third-party components" on page 775.

Other ODBC or OLE DB files your application may require are not added. For information about deploying these files, see "ODBC database drivers and supporting files" on page 779 and "OLE DB database providers" on page 783.

5 If your application uses DataWindow XML export or import or XML Web DataWindows, check the XML support check box.

The Runtime Packager adds *PBXerces110.dll*, *xerces-c_2_6.dll*, and *xerces-depdom_2_6.dll*.

6 If your application uses the XML services provided by the PowerBuilder Document Object Model or if it is an EJB or SOAP Web services client, select the appropriate check boxes.

The Runtime Packager adds the DLLs, PBXs, and JAR files required to the package. The Runtime Packager adds required files for both the EasySoap and .NET Web service engines when you select the SOAP Client for Web Service check box. For more information about required files for these services, see "PowerBuilder extensions" on page 788.

Web service DataWindows
You can also check the SOAP Client for Web Service box if your application uses Web service DataWindows. Two of the files added when you check this box, *Sybase.PowerBuilder.WebService.Runtime.dll* and *Sybase.PowerBuilder.WebService.Runtime.RemoteLoader.dll*, are also required for Web service DataWindows.

7 If your application uses a rich text control or DataWindow, select the Rich Text Support check box.

The Runtime Packager installs the files listed for Rich Text support in Table 41-5 on page 776

8 Click Create.

The Runtime Packager creates an MSI file that includes the files required by the components you selected, as well as the runtime DLLs for standard PowerBuilder applications or the runtime DLLs and .NET assemblies for PowerBuilder .NET applications listed in Table 41-3.

Table 41-3: Base components

Base components selected	Files
PowerBuilder Standard Components	*libjcc.dll* *libjutils.dll* *pbacc110.dll* *pbcomrt110.dll* *pbdwe110.dll* *pbdwr110.dll* *pbdwr110.pbd* *pbjag110.dll* *pbjvm110.dll* *pbshr110.dll* *pbtra110.dll* *pbtrs110.dll* *pbvm110.dll*
PowerBuilder .NET components	*pbshr110.dll* *pbrth110.dll* *pbdwm110.dll* PowerBuilder .NET assemblies: *Sybase.PowerBuilder.ADO.dll* *Sybase.PowerBuilder.Common.dll* *Sybase.PowerBuilder.Core.dll* *Sybase.PowerBuilder.Datawindow.Web.dll* *Sybase.PowerBuilder.DataWindow.Win.dll* *Sybase.PowerBuilder.Datawindow.Interop.dll* *Sybase.PowerBuilder.Db.dll* *Sybase.PowerBuilder.DBExt.dll* *Sybase.PowerBuilder.EditMask.Win.dll* *Sybase.PowerBuilder.EditMask.Interop.dll* *Sybase.PowerBuilder.Graph.Web.dll* *Sybase.PowerBuilder.Graph.Win.dll* *Sybase.PowerBuilder.Graph.Core.dll* *Sybase.PowerBuilder.Graph.Interop.dll* *Sybase.PowerBuilder.RTC.Win.dll* *Sybase.PowerBuilder.RTC.Interop.dll* *Sybase.PowerBuilder.Interop.dll* *Sybase.PowerBuilder.Web.dll* *Sybase.PowerBuilder.Win.dll*

The MSI file is a compressed file that can be executed directly on any Windows platform. It registers any self-registering DLLs, adds the installation destination path to the Windows Registry, sets the system PATH environment variable, and adds information to the Registry for the Install/Uninstall page in the Windows Control Panel. It can also be used in some third-party installation software packages.

For .NET applications, the MSI file installs the .NET assemblies into the Global Application Cache (GAC). The .NET Framework 2.0 must be installed on the target computer. If it is not, the installation stops and an error message displays. If the IE Web Controls required for .NET Web Forms targets are not installed, a warning message displays but installation continues. After you run the MSI file on a production server for .NET Web Forms applications or Web services, or on a client computer for .NET Windows Forms or smart client applications, you must restart the target system.

Third-party components

PowerBuilder applications have some dependencies on third-party components that are not installed with the Runtime Packager.

Note
Some files must be available on the user’s computer before you install the PowerBuilder runtime files. For example, the Microsoft .NET Active Template Library (ATL) module, *atl71.dll*, must be installed on the user’s computer before the *pbjvm110.dll* can be registered.

For the most current information about third-party components, see the *Release Bulletin*. For information about components that can be freely downloaded, see the Sybase Web site at http://www.sybase.com/softwarelicenses/third_party_legal. You may be able to redistribute some components provided that you meet the conditions described in the license.

PowerBuilder runtime files

Database connectivity
Files required for database connectivity are listed separately in “Database connections” on page 777.

Core runtime files

Table 41-4 lists the core PowerBuilder runtime files.

Table 41-4: Core PowerBuilder runtime files

Name	Required for
pbvm110.dll	All.
pbshr110.dll	All. *pbvm110.dll* has dependencies on this file.
libjcc.dll	All. *pbvm110.dll* has dependencies on this file.
libjutils.dll	All. *libjcc.dll* has dependencies on this file.
pbdwe110.dll	DataWindows and DataStores (for .NET applications, use *pbdwm110.dll* instead).

Microsoft files

When you deploy the core PowerBuilder runtime files, you must also deploy the *msvcr71.dll* and *msvcp71.dll* Microsoft Visual C++ runtime libraries and the Microsoft .NET Active Template Library (ATL) module, *atl71.dll*, if they are not present on the user's computer. The PowerBuilder runtime files have a runtime dependency on these files.

Microsoft Windows GDI+ is the subsystem of the Windows XP or Windows Server 2003 operating system that implements enhanced graphic capabilities for screens and printers. The PowerBuilder runtime files have a runtime dependency on *gdiplus.dll*. If you deploy your PowerBuilder application to the Windows 2000 platform, you must make sure that *gdiplus.dll* is available on the target computer.

Additional runtime files

Table 41-5 lists additional runtime files that your application might not require. For example, *pbvm110.dll* is required for all deployed applications, but *pbrtc110.dll* and its associated runtime files are required only if your application uses Rich Text controls or RichText DataWindow objects.

For more information about deploying applications that use the *pbjvm110.dll* for Java support, see "Java support" on page 786.

Table 41-5: Additional PowerBuilder runtime files

Name	Required for
pbacc110.dll	Accessibility support (Section 508)
pbdwr110.dll, *pbdwr110.pbd*	Web DataWindow support
PBXerces110.dll, *xerces-c_2_6.dll*, *xerces-depdom_2_6.dll*	XML Web DataWindow support and XML support for DataWindows and DataStores
Sybase.PowerBuilder.WebService.Runtime.dll, *Sybase.PowerBuilder.WebService.RuntimeRemoteLoader.dll*	Web service DataWindows
pbjvm110.dll	Java support

Name	Required for
pbrth110.dll	.NET Web Forms and ADO.NET
pbrtc110.dll, *tp13.dll*, *tp13_bmp.flt*, *tp13_css.dll*, *tp13_doc.dll*, *tp13_gif.flt*, *tp13_htm.dll*, *tp13_ic.dll*, *tp13_ic.ini*, *tp13_jpg.flt*, *tp13_obj.dll*, *tp13_pdf.dll*, *tp13_png.flt*, *tp13_rtf.dll*, *tp13_tif.flt*, *tp13_tls.dll*, *tp13_wmf.flt*, *tp13_wnd.dll*, *tp4ole13.ocx*	Rich Text support
pblab110.ini	Label DataWindow presentation-style predefined formats
pbtra110.dll, *pbtrs110.dll*	Database connection tracing

Installed path *\Program Files\Sybase\Shared\PowerBuilder* or, for most of the required rich text files, *\Program Files\Sybase\Shared\PowerBuilder\RTC*.

Deployment path Same directory as the application, in a directory on the system path, or in the App Path registry key.

Registry entries See “App Path registry key” on page 770.

Localized runtime files

Localized runtime files are provided for French, German, Italian, Spanish, Dutch, Danish, Norwegian, and Swedish. These files are usually available shortly after the general release of a new version of PowerBuilder. The localized runtime files let you deploy PowerBuilder applications with standard runtime dialog boxes in the local language. They handle language-specific data when the application runs.

For more information, see “Localizing the product” on page 705.

Database connections

If you are deploying an executable or component that accesses a database, your users need access to the DBMS and to the database your application uses.

Where to install database connectivity files
You *do not need to deploy database connectivity files* with a client application that relies on a middle-tier component on another computer to perform database transactions. Database connectivity files must be deployed on the computer that interacts with the database server.

You need to:

- If necessary, install the DBMS runtime (client) files in the application directory or in a directory on the system path

  If your application uses a standalone SQL Anywhere database, you can install the SQL Anywhere Runtime Edition files on the user's computer. For more information, see "SQL Anywhere files" on page 781. Otherwise follow the instructions and licensing rules specified by the vendor.

- Make sure each user has access to the database the application uses

  If your application uses a local database, install the database and any associated files, such as a log file, on the user's computer.

  If your application uses a server database, make sure the user's computer is set up to access the database. This may be the task of a database administrator.

- Install any database interfaces your application uses on the user's computer
- If your application uses the ODBC interface, configure the ODBC database drivers and data sources, as described in "Configuring ODBC data sources and drivers" on page 782

For more information about database drivers and interfaces, see:

- "Native database drivers" next
- "ODBC database drivers and supporting files" on page 779
- "OLE DB database providers" on page 783
- "ADO.NET database interface" on page 784
- "JDBC database interface" on page 785

Native database drivers

Table 41-6 lists the native database drivers supplied with PowerBuilder. If an application or component uses the database specified, the file is required on the computer. The first two characters of the native database file name are PB, the next three characters identify the database, and the last two identify the version of PowerBuilder.

Table 41-6: PowerBuilder native database drivers

Name	Required for
pbin9110.dll	INFORMIX I-Net 9
pbo84110.dll	Oracle 8.0.x and Oracle8*i* 8.1.x
pbo90110.dll	Oracle9*i*
pbo10110.dll	Oracle 10*g*
pbsnc110.dll	SQL Native Client for Microsoft SQL Server
pbdir110.dll	Sybase DirectConnect
pbase110.dll	Sybase Adaptive Server Enterprise CT-LIB for Adaptive Server 15 only
pbsyc110.dll	Sybase Adaptive Server Enterprise CT-LIB
pbsyj110.dll	Sybase Adaptive Server Enterprise CT-LIB for EAServer deployment only

Installed path *\Program Files\Sybase\Shared\PowerBuilder*

Deployment path Same directory as the application, in a directory on the system path, or in the App Path registry key.

Registry entries See “App Path registry key” on page 770.

Notes When you deploy a PowerBuilder custom class user object to EAServer, you need to use the SYJ database interface rather than SYC to connect to an Adaptive Server Enterprise database. You cannot use SYJ in the PowerBuilder development environment, but you can use the SYJ Database Profile Setup dialog box to set the appropriate connection parameters. You can then copy the syntax from the Preview tab into the script for your Transaction object.

ODBC database drivers and supporting files

This section lists files that are required for all ODBC database connections from PowerBuilder or InfoMaker applications, as well as files required for a specific database interface or DBMS.

PowerBuilder ODBC interface files

The following PowerBuilder ODBC interface files are required if your application uses ODBC:

Table 41-7: PowerBuilder ODBC interface files

Name	Description
pbodb110.dll	PowerBuilder ODBC interface
pbodb110.ini	PowerBuilder ODBC initialization file

Installed path *\Program Files\Sybase\Shared\PowerBuilder*

Deployment path Same directory as the application, in a directory on the system path, or in the App Path registry key.

Registry entries See "App Path registry key" on page 770.

Notes The INI and DLL files must be in the same directory. If you have modified the *pbodb110* initialization file, make sure you deploy the modified version.

Microsoft ODBC files

Table 41-8 lists the Microsoft ODBC files that are required if your application uses ODBC.

Table 41-8: Microsoft ODBC files

Name	Description
DS16GT.dll *DS32GT.dll* *ODBC32.dll* *ODBC32GT.dll* *ODBCAD32.exe* *ODBCCP32.cpl* *ODBCCP32.dll* *ODBCCR32.dll* *ODBCINST.cnt* *ODBCINST.hlp* *ODBCINT.dll* *ODBCTRAC.dll*	Microsoft ODBC driver manager, DLLs, and Help files

Installed path Windows system directory.

Deployment path Windows system directory.

Registry entries None.

Notes The Microsoft ODBC Driver Manager (*ODBC32.dll*) and supporting files are usually already installed in the user's Windows system directory.

SQL Anywhere files

If your PowerBuilder application uses a SQL Anywhere database, you need to deploy the SQL Anywhere DBMS as well as SQL Anywhere's ODBC database drivers.

Restrictions
PowerBuilder includes SQL Anywhere for use during the development process. However, this product cannot be deployed royalty-free to your users.

If your application requires the data definition language (DDL), a transaction log, stored procedures, or triggers, see your Sybase sales representative.

For more information about deploying SQL Anywhere databases and applications, see the SQL Anywhere documentation.

If your application uses a standalone database, you can deploy the SQL Anywhere Desktop Runtime System to users' computers without incurring additional license fees. The runtime system allows the user to retrieve and modify data in the database, but does not allow modifications to the database schema. It does not support transaction logs, stored procedures, or triggers.

A full installation for the SQL Anywhere driver, runtime engine, and supporting files is available in the *Support\SA10Runtime* directory on the CD. Table 41-9 lists some of the files that are installed. For more information see the *RuntimeEdition.html* file in the installed directory.

Table 41-9: SQL Anywhere files

Name	Description
dbodbc10.dll	SQL Anywhere ODBC driver
dbbackup.exe	SQL Anywhere backup utility
dbcon10.dll	Connection dialog box, required if you do not provide your own dialog box and your end users are to create their own data sources, if they need to enter user IDs and passwords when connecting to the database, or if they need to display the Connection dialog box for any other purpose
dbisqlc.exe	Interactive SQL utility
dblgen10.dll	Language-specific string library (*EN* indicates the English version)
dblib10.dll	Interface library
dbtool10.dll	SQL Anywhere database tools
dbunlspt.exe	SQL Anywhere unload utility
dbvalid.exe	SQL Anywhere validation utility
rteng10.exe	Restricted runtime engine

Name	Description
dbctrs10.dll	Performance utility
dbserv10.dll	Server utility

Installed path *\Program Files\Sybase\SQL Anywhere 10\win32*

Deployment path Same directory as the application, in a directory on the system path, or in the App Path registry key.

Registry entries See “App Path registry key” on page 770 and "Configuring ODBC data sources and drivers" next.

Notes Supporting files should be installed in the same directory as *dbodbc10.dll*. If you are not using the English string library, make sure you deploy the appropriate version of the language-specific string library.

Configuring ODBC data sources and drivers

ODBC.INI To allow the user to connect to a particular data source, your installation program must provide a definition for that data source in the ODBC.INI key in the registry on the computer that accesses the data source, in *HKEY_CURRENT_USER* for a user DSN or in *HKEY_LOCAL_MACHINE* for a system DSN. The data source definition specifies the name and location of the database driver as well as the command required to start the database engine. The data source in the ODBC Data Sources key must also be listed in ODBC.INI.

The following shows typical registry entries for a data source called MyApp DB that uses SQL Anywhere. Registry keys are enclosed in square brackets and are followed by string values for that key in the format `"`*Name*`"="`*Value*`"`:

```
[HKEY_CURRENT_USER\SOFTWARE\ODBC\ODBC.INI\MyApp DB]
"Driver"="C:\Program Files\Sybase\SQL Anywhere 10\
   win32\dbodbc10.dll"
"Start"="c:\program files\sybase\SQL Anywhere 10\win32\
   rteng10.exe -c9m"
"UID"="dba"
"PWD"="sql"
"Description"="Database for my application"
"DatabaseFile"="C:\Program Files\myapps\myapp.db"
"AutoStop"="Yes"


[HKEY_CURRENT_USER\SOFTWARE\ODBC\ODBC.INI\
   ODBC Data Sources]
"MyApp DB"="SQL Anywhere 10.0"
```

ODBCINST.INI Your installation program needs to make two types of entry in the *ODBCINST.INI* key in *HKEY_LOCAL_MACHINE\SOFTWARE\ODBC* for each driver that your deployed application uses:

- Add a string value with the name of the driver and the data value "Installed" to the *ODBC DRIVERS* key in *ODBCINST.INI*
- Add a new key for each driver to the *ODBCINST.INI* key with string values for Driver and Setup

  Some drivers require additional string values in *ODBCINST.INI*.

If the ODBC database driver files are not located in a directory on the system path, you also need to add their location to the App Paths key for the executable file.

If you are using ODBC drivers obtained from a vendor, you can use the driver's setup program to install the driver and create registry entries.

The following shows a typical registry entry for SQL Anywhere. A registry key is enclosed in square brackets and is followed by string values for the key in the format "*Name*"="*Value*":

```
[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\
   SQL Anywhere 10.0]
"Driver"="c:\program files\sybase\SQL Anywhere 10\
   win32\dbodbc10.dll"
"Setup"="c:\program files\sybase\SQL Anywhere 10\
   win32\dbodbc10.dll"
```

For more information about the contents of the registry entries for ODBC drivers and data sources, see *Connecting to Your Database*.

OLE DB database providers

If your application uses OLE DB to access data, you must install Microsoft's Data Access Components software on each user's computer if it is not installed already.

The PowerBuilder OLE DB interface requires the functionality of the Microsoft Data Access Components (MDAC) version 2.8 or higher software. Version 2.8 is distributed with Windows XP Service Pack 2 and Windows Server 2003.

To check the version of MDAC on a computer, users can download and run the MDAC Component Checker utility from the MDAC Downloads page at http://msdn2.microsoft.com/en-us/data/aa937730.aspx.

On the Windows Vista operating system, the Windows Data Access Components (DAC) version 6.0 includes some changes to work with Vista but is otherwise functionally equivalent to MDAC 2.8.

OLE DB data providers installed with MDAC
Several Microsoft OLE DB data providers are automatically installed with MDAC, including the providers for SQL Server (SQLOLEDB) and ODBC (MSDASQL).

PowerBuilder OLE DB interface files

The PowerBuilder OLE DB interface file is required if your application uses OLE DB. The ODBC initialization file is required if you have used it to customize OLE DB settings:

Table 41-10: PowerBuilder OLE DB interface files

Name	Description
pbole110.dll	PowerBuilder OLE DB interface
pbodb110.ini	PowerBuilder ODBC initialization file

Installed path *\Program Files\Sybase\Shared\PowerBuilder*

Deployment path Same directory as the application, in a directory on the system path, or in the App Path registry key.

Registry entries See "App Path registry key" on page 770.

Notes The INI and DLL files must be in the same directory. If you have modified the *pbodb110* initialization file, make sure you deploy the modified version.

ADO.NET database interface

The PowerBuilder ADO.NET interface supports the OLE DB, Microsoft SQL Server .NET, Oracle ODP.NET, and Sybase ASE data providers. If you use ADO.NET, you must deploy *pbado110.dll*, *pbrth110.dll*, *Sybase.PowerBuilder.Db.dll*, *Sybase.PowerBuilder.DbExt.dll*, and, for OLE DB, the OLE DB data provider.

The files *pbado110.dll* and *pbrth110.dll* are standard DLL files, and you can deploy them in the same way as other PowerBuilder DLLs. However, *Sybase.PowerBuilder.Db.dll* and *Sybase.PowerBuilder.DbExt.dll* are .NET assemblies. You can use one of three techniques to deploy the files:

- Deploy *Sybase.PowerBuilder.Db.dll* and *Sybase.PowerBuilder.DbExt.dll* in the same directory as the executable file that calls the ADO.NET driver.

- Use a .NET application configuration file to assign the path of *Sybase.PowerBuilder.Db.dll* and *Sybase.PowerBuilder.DbExt.dll*. The file contains configuration settings that the common language runtime (CLR) reads as well as settings that the application reads. For an executable file, the configuration file has the same name as the executable file with the extension *.config*. The *pb110.exe.config* file in your PowerBuilder 11.0 directory is an example.

  For more information about configuration files, see the Microsoft Visual Studio SDK documentation.

- Add the *Sybase.PowerBuilder.Db.dll* and *Sybase.PowerBuilder.DbExt.dll* assemblies to the Global Assembly Cache (GAC). For more information about the GAC, see the section on the Global Assembly Cache in the Microsoft Visual Studio SDK documentation. If you use the Runtime Packager, the assemblies are installed in the GAC.

JDBC database interface

The PowerBuilder JDB interface supports the Sun Java Runtime Environment (JRE) versions 1.2 and later.

If your application or component uses JDBC connections, you must deploy the JDB driver as well as the appropriate Java package for the Java VM you are using. The Java virtual machine and a vendor-supplied JDBC-compliant driver, such as Sybase jConnect® for JDBC, must also be installed and configured on the computer that accesses the data source.

For more information about the Java VM, see "Java support" next.

Table 41-11: PowerBuilder JDB files

Name	Description
pbjdb110.dll	PowerBuilder JDBC Driver (JDB) for JRE 1.2 or later
pbjdbc12110.jar	Java package for PowerBuilder JDB driver and JRE 1.2 or later

Installed path *\Program Files\Sybase\Shared\PowerBuilder*

Deployment path Same directory as the application, in a directory on the system path, or in the App Path registry key.

Registry entries Make sure the CLASSPATH environment variable includes the PowerBuilder *pbjdbc12110.jar* file. For example:

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control
```

```
       \Session Manager\Environment]
   "CLASSPATH"="C:\Program Files\sybase\shared\
       PowerBuilder\pbjdbc12110.jar;...
```

Notes See "PowerBuilder components on EAServer" on page 794 and "Web DataWindow on EAServer" on page 799.

Java support

You must deploy the PowerBuilder *pbjvm110.dll* file with any applications or components that use the Java Runtime Environment (JRE), and there must be a JRE installed on the target computer. The JRE is required for JSP targets, EJB clients, JDBC connections, and saving as PDF using XSL-FO. You can copy the JRE installed with PowerBuilder to the same directory as the PowerBuilder runtime files on the target computer, or use an existing JRE whose location is defined in the user's system PATH environment variable.

Locating the Java VM

When a PowerBuilder application requires a Java VM, the PowerBuilder runtime searches for the *jvm.dll* file in a subdirectory of the directory where *pbjvm110.dll* is installed on the user's computer. The *jvm.dll* file is installed in the *JRE\bin\client* directory of JDK 1.4 and later installations, and in the *JRE\bin\classic* directory in JDK 1.2 and 1.3 installations.

PowerBuilder adds the location of *jvm.dll* to the beginning of the path currently being used by the PowerBuilder application. This path is a copy of the path defined in the user's PATH system environment variable. PowerBuilder does *not* modify the environment variable maintained in the Windows registry.

To locate the *jvm.dll*, PowerBuilder first determines where *pbjvm110.dll* is installed. Suppose *pbjvm110.dll* is installed in *C:\Sybase\Shared\PowerBuilder*. Then PowerBuilder uses this search procedure to add the location of the *jvm.dll* to the path currently in use:

1. Search for the directory structure *JRE\bin\client* (for JDK 1.4 or later) in *C:\Sybase\Shared\PowerBuilder\* and, if found, add it to the beginning of the path.
2. If not found, search for a JDK directory structure that contains *JRE\bin\client* in *C:\Sybase\Shared\PowerBuilder\* and, if found, add it to the beginning of the path.

3 If not found, search for the directory structure *JRE\bin\classic* (for JDK 1.2 or 1.3) in *C:\Sybase\Shared\PowerBuilder\* and, if found, add it to the beginning of the path.

If none of these directory structures is found, PowerBuilder uses the first *jvm.dll* whose location is defined in the user's PATH environment variable. If no *jvm.dll* is found, the Java VM does not start.

The runtime Java VM classpath

When PowerBuilder starts a Java VM, the Java VM uses internal path and class path information to ensure that required Java classes are always available. At runtime, the Java VM uses a class path constructed by concatenating these paths:

- The system JAVA_HOME environment variable.
- A class path added programmatically when the Java VM is started. For example, EJB client applications can pass a class path to the CreateJavaVM method.
- The PowerBuilder runtime static registry class path. This is a path built into the *pbjvm110.dll* file that corresponds to the path in the Windows Registry that is used when you are developing an application in PowerBuilder. It contains classes required at runtime for features that use a Java VM.
- The system CLASSPATH environment variable.
- The current directory.

Overriding the runtime static registry classpath

If necessary, you can override the JVM settings and properties defined for runtime use in the static registry. PowerBuilder uses the following algorithm to locate configuration information:

1 When the first request is made for a JVM, PowerBuilder looks for registry entries for the configuration information and properties to be passed to the function that creates the JVM.

2 If PowerBuilder finds a registry entry for the configuration information, it uses it instead of the static registry. If it does not find a registry entry, it uses the static registry.

3 If PowerBuilder finds a registry entry for custom properties to be passed to the JVM, it uses those instead of the static registry. If it does not find a registry entry, it uses the static registry entries.

To override the default settings, create a new key named *PBRTConfig* in the *HKEY_LOCAL_MACHINE\Software\Sybase\PowerBuilder\11.0\Java* key, then add either or both of the following subkeys: *PBJVMconfig* and *PBJVMprops*.

To duplicate the static registry entries, add the same string values to these subkeys that you see in the PBIDEConfig key, that is:

Subkey	String value name	String value data
PBJVMconfig	Count	1
	0	-verbose:jni,class
PBJVMprops	java.compiler	NONE

You can override either the configuration or properties entries or both. If you make incorrect entries, PowerBuilder attempts to recover by defaulting to the static registry. However, you should be cautious about making any changes since you can cause incorrect behavior in the JVM if you do not configure it correctly.

PowerBuilder extensions

Several PowerBuilder extension files are provided with PowerBuilder 11.0. If your application uses one of these extensions, you must deploy the files listed in Table 41-12.

Table 41-12: Files required for PowerBuilder built-in extensions

Extension	Files
PowerBuilder Document Object Model	*pbdom110.pbx*, *PBXerces110.dll*, *xerces-c_2_6.dll*, *xerces-depdom_2_6.dll*
EJB client	*pbejbclient110.pbx*, *pbejbclient110.jar*
SOAP client for Web services	*ExPat110.dll* , *libeay32.dll*, *ssleay32.dll*, *xerces-c_2_6.dll*, *xerces-depdom_2_6.dll*, *EasySoap110.dll*, *pbnetwsruntime110.dll*, *pbsoapclient110.pbx*, *pbwsclient110.pbx*, *Sybase.PowerBuilder.WebService.Runtime.dll*, *Sybase.PowerBuilder.WebService.RuntimeRemoteLoader.dll*

In addition to the files listed in the table for EJB client, a Java Runtime Environment (JRE) compatible with the JDK on the EJB server must be available on the client and listed in the CLASSPATH.

For more information, see “Java support” on page 786.

PDF and XSL-FO export

PowerBuilder can save the DataWindow’s data and presentation as a Portable Document Format (PDF) file using two techniques. By default, PowerBuilder saves as PDF using a distiller. PowerBuilder can also save to PDF or XSL Formatting Objects (XSL-FO) format using the Apache XML Formatting Objects processor.

Using the Ghostscript distiller

In order for users to use the SaveAs method to save data as PDF with the distiller, they must first download and install Ghostscript on their computers as described in the procedure that follows.

The use of GPL Ghostscript is subject to the terms and conditions of the GNU General Public License (GPL). Users should be asked to read the GPL before installing GPL Ghostscript on their computers. A copy of the GPL is available on the GNU Project Web server at http://www.gnu.org/licenses/gpl.html.

The use of AFPL Ghostscript is subject to the terms and conditions of the Aladdin Free Public License (AFPL). Commercial distribution of AFPL Ghostscript generally requires a written commercial license. For more information, see the Ghostscript Web site at http://www.ghostscript.com/awki.

❖ **To install Ghostscript:**

1. Into a temporary directory on your computer, download the self-extracting executable file for the version of Ghostscript you want from one of the sites listed on the Ghostscript Web site at http://www.ghostscript.com/awki.

   See the *Release Bulletin* for the version of Ghostscript that was used for testing.

2. Run the executable file to install Ghostscript on your system.

   The default installation directory is *C:\Program Files\gs*. You can select a different directory and/or choose to install shortcuts to the Ghostscript console and readme file.

After installing Ghostscript, you should read the *readme.htm* file in the *doc* subdirectory in the Ghostscript installation directory to find out more about using Ghostscript and distributing it with your application.

Save Rows As fails
To save as PDF in the DataWindow painter, select File>Save Rows As and select PDF as the Save As type. If you do not install Ghostscript and use the default export properties, PowerBuilder displays a pop-up window notifying you that Save Rows As failed. If you install Ghostscript and then change the name of the directory where Ghostscript is installed, Save Rows As PDF fails silently.

Location of files

When you save a DataWindow object as PDF using the distill method, PowerBuilder searches in the following locations for an installation of GPL or AFPL Ghostscript:

- The Windows registry
- The relative path of the *pbdwe110.dll* file (typically *Sybase\Shared\PowerBuilder*)
- The system PATH environment variable

If GPL or AFPL Ghostscript is installed using the Ghostscript executable file, the path is added to the Windows registry.

If the Ghostscript files are in the relative path of the *pbdwe110.dll* file, they must be installed in this directory structure:

```
dirname\pbdwe110.dll
dirname\gs\gsN.NN
dirname\gs\fonts
```

where *dirname* is the directory that contains the runtime DLLs and *N.NN* represents the release version number for Ghostscript.

You might not need to distribute all the fonts provided in the distribution. For information about fonts, see Fonts and font facilities supplied with Ghostscript at http://www.ghostscript.com/doc/current/Fonts.htm.

PostScript printer drivers

If your users have installed a PostScript printer on their computers, the PostScript driver files required to create PDF files are already installed. If they have never installed a PostScript printer, they can use the Printers and Faxes option in the Windows control panel to install a generic PostScript printer. If the Microsoft *Pscript5.dll* has never been installed, they may be prompted to insert the Windows install CD.

You must also deploy the related files that are installed in *Sybase\Shared\PowerBuilder\drivers*. These files can be copied to or installed on users' computers. They must be located in this directory structure:

```
dirname\pbdwe110.dll
dirname\drivers
```

PostScript printer profile

Each user's computer must have a PostScript printer profile called Sybase DataWindow PS. This profile is added to your development computer automatically when you save a DataWindow's rows to a PDF file in the DataWindow painter. You can use this method to add a Sybase DataWindow PS printer on any computer where PowerBuilder is installed.

Users can also add the profile manually using the Windows Add Printer wizard in one of the following ways:

- By clicking the Have Disk button on the Install Printer Software page of the wizard, browsing to the *Adist5.inf* file (installed with PowerBuilder in the *Shared\PowerBuilder\drivers* directory) or to another PostScript driver file, and on the Name Your Printer page, changing the printer name to "Sybase DataWindow PS".
- By selecting a printer with PS in its name (such as "Apple Color LW 12/660 PS") from the list of printers on the Install Printer Software page of the wizard, and changing the printer name to "Sybase DataWindow PS" on the Name Your Printer page.

See the chapter on the Print Manager in the *Deploying Applications and Components to .NET* book if your applications print to PDF or XSL files from an IIS server.

Using the Apache FO processor

If your application uses the Apache processor to save as PDF or XSL-FO, you must deploy the *fop-0.20.4* directory and the Java Runtime Environment (JRE) with your application.

They must be deployed in the same directory as the PowerBuilder runtime files. For example, if you deploy your application and *pbvm110.dll* and the other PowerBuilder runtime files in a directory called *MyApplication*, the Apache processor must be deployed in *MyApplication/fop-0.20.4,* and the JRE in *MyApplication/jre*. However, you do not need to place a copy of the JRE in this location if the full JDK is installed on the target computer and is in the classpath.

The following JAR files must be in the user's classpath:

fop-0.20.4\build\fop.jar
fop-0.20.4\lib\batik.jar
fop-0.20.4\lib\xalan-2.3.1.jar
fop-0.20.4\lib\xercesImpl-2.1.0.jar
fop-0.20.4\lib\xml-apis.jar
fop-0.20.4\lib\avalon-framework-cvs-20020315.jar

For more information about the JRE, see "Java support" on page 786.

On Windows DBCS platforms, you also need to deploy a file that supports DBCS characters to the Windows font directory on the target computer, for example, *C:\WINDOWS\fonts*. For more information about configuring fonts, see the Apache Web site at http://xml.apache.org/fop/fonts.html.

DataWindow Web control for ActiveX

The following files must be deployed to your Web server if you are using the DataWindow Web control for ActiveX.

Table 41-13: PowerBuilder file for DataWindow Web control for ActiveX

Name	Required for
psdwc110.cab	CAB file containing an Open Software Distribution information file and DLLs for the Web ActiveX and the Transaction control
pbjdbc12110.jar	JAR file containing required Java classes

If the target system does not have a Sun JRE installed, the user must download a JRE from the Sun Java Web site. To make the Web ActiveX and the Transaction control available to the client browser, code the CODEBASE attribute in the Object element in the deployed HTML page.

About the CODEBASE attribute

The CODEBASE attribute identifies the location of a CAB or OCX file so that a browser can download it, unpack it if it is a CAB file, and register it on the user's computer. A typical value for CODEBASE uses a relative URL to identify the location of the CAB or OCX file followed by a # sign and a four-part version number separated by commas. The version number is the same as the version number of PowerBuilder. For example:

```
CODEBASE="cabs/psdwc110.cab#10,0,0,5031"
```

Additional files may be required. For more information see “Deploying the DataWindow Web control” in the PowerBuilder online books or the *DataWindow Programmer’s Guide*.

Plug-ins and PowerBuilder window ActiveX controls

Plug-ins

The files listed in Table 41-14 must be deployed to users’ computers if you are using DataWindow or window plug-ins.

Table 41-14: PowerBuilder files for window and DataWindow plug-ins

Name	Required for
nppba110.dll	Standard window plug-in
nppbs110.dll	Secure window plug-in
npdwe110.dll	DataWindow plug-in

Installed path *\Program Files\Sybase\PowerBuilder 11.0\Internet Tools\Plugins*

Deployment path Browser plugins directory.

Notes All plug-ins require a browser that supports plug-ins. Microsoft Internet Explore 5.5 Service Pack 2 and later versions do not support plug-ins. Window plug-ins also require the PowerBuilder runtime DLLs on users’ computers. For more information, see Chapter 32, “Using the DataWindow Plug-in,” and Chapter 33, “Using the PowerBuilder Window Plug-in.”

Window ActiveX controls

The files listed in Table 41-15 must be made available on users’ computers if you are using the PowerBuilder window ActiveX.

Table 41-15: PowerBuilder files for window ActiveX

Name	Required for
pbrx110.ocx	Standard PowerBuilder window ActiveX
pbrxs110.ocx	Secure PowerBuilder window ActiveX

Installed path Windows system directory.

Deployment path Windows system directory.

Registry entries The PowerBuilder window ActiveX controls can be made available on the client computer by copying them to the computer and registering them using the REGSVR32 utility. Alternatively, include the CODEBASE attribute in the Object element in the HTML page on the Web server. See “About the CODEBASE attribute” on page 792.

Notes PowerBuilder window ActiveX controls require a browser that supports ActiveX controls. They also require that the PowerBuilder runtime DLLs and the Microsoft files *MFC42.dll*, *MSVCRT.dll*, *URL.dll*, and *URLMON.dll* be available on users' computers. The Microsoft files are usually already installed in the target computer's Windows system directory. If they are not installed, they can be downloaded as a CAB file from the Microsoft Internet Explorer Components Gallery at http://activex.microsoft.com/controls/vc/mfc42.cab.

For more information, see Chapter 34, "Using the PowerBuilder Window ActiveX."

PowerBuilder components on EAServer

To run PowerBuilder components in EAServer, the runtime libraries for the version and build number of PowerBuilder in which the components were developed must be available on the server. When you install a maintenance release of PowerBuilder and deploy new or updated components to EAServer, make sure that the PowerBuilder VM on the servers is also updated.

The files listed in Table 41-16 are required on the EAServer host on which components generated in PowerBuilder will run. If PowerBuilder components do not use a feature or database interface listed in the table, the file does not need to be installed on the server. The PowerBuilder VM also requires the EAServer *libjcc* file and its dependent files. In the UNIX column of the table, *ext* represents the platform-specific extension for library names; for example, on Solaris this file is *libpbvm110x.so*.

Table 41-16: PowerBuilder files required on EAServer host

Windows	UNIX	Description
pbvm110.dll	*libpbvm110x.ext*	PowerBuilder virtual machine (required for all PowerBuilder components)
pbshr110.dll	*pbshr110.ext*	Required by PowerBuilder virtual machine
pbdwe110.dll	*libpbdwe110x.ext*	DataStore support
pbjag110.dll	*libpbjag110x.ext*, *pbjag110.ext*	PowerBuilder support in EAServer
pbdwr110.pbd	*pbdwr110.pbd*	Web DataWindow support (requires PBDWE)
htmldw.js	*htmldw.js*	Web DataWindow support

Windows	UNIX	Description
pbRTC110.dll and additional runtime files listed in Table 41-5	—	RichText support
PBXerces110.dll, *xerces-c_2_6.dll*, *xerces-depdom_2_6.dll*	*libxerces110x.ext*, *libxerces-c_2_1_0.ext*	XML support
pbdom110.pbx	*libpbdom110x.ext*	PBDOM support
EasySoap110.dll, *ExPat110.dll*, *libeay32.dll*, *ssleay32.dll*, *pbsoapclient110.pbx*, *pbwsclient110.pbx*, *pbnetwsruntime110.dll*, *xerces-c_2_6.dll*, *xerces-depdom_2_6.dll*, *Sybase.PowerBuilder.WebService.Runtime.dll*, *Sybase.PowerBuilder.WebService.RuntimeRemoteLoader.dll*	—	SOAP clients for EasySoap and .NET Web services (for .NET Web services, the two *Sybase.PowerBuilder DLL* files must be deployed to the *EAServer\Bin* directory)
pbo84110.dll	*libpbo84110x.ext*	Oracle 8.0.x and Oracle8*i* 8.1.x database driver
pbo90110.dll	*libpbo90110x.ext* (Solaris and Linux only)	Oracle9*i* database driver
pbo10110.dll	*libpbo10110x.ext* (Solaris and Linux only)	Oracle 10*g* database driver
pbodb110.ini	*pbodb110.ini*	PowerBuilder ODBC initialization file
pbodb110.dll	*libpbodb110x.ext*	PowerBuilder ODBC interface
pbsnc110.dll	*libpbsnc110x.ext*	SQL Native Client native database interface
pbsyj110.dll	*libpbsyj110x.ext*	Adaptive Server Enterprise native database interface
pbjdb110.dll	*libjdb110x.ext*	JDBC database driver for Sun Java VM JRE 1.1 or later

Windows	UNIX	Description
—	*libpbwfr110.ext*	PowerBuilder UNIX extension library
pbjdbc12110.jar	*pbjdbc12110.jar*	Java classes for PowerBuilder JDBC drivers (required for JRE 1.2 or later)

On Windows

You can use the PowerBuilder VM installer, available in the PBVM folder on the CD, to install the files listed in Table 41-16. The PBVM setup program also installs the PowerBuilder 11.0 version of the Web DataWindow server component (HTMLGenerator110) and the PBDebugBroker110 component required for remote debugging.

You can use other database drivers with EAServer components, but you must use one of the drivers listed in the table above if you want to take advantage of EAServer support for transaction management and instance pooling.

On UNIX

If the files listed in Table 41-16 were not installed when you installed EAServer, they might be available for your platform from the Sybase Downloads site on the EBFs/Maintenance page at http://downloads.sybase.com/.

PowerBuilder components deployed to EAServer on a UNIX platform can have neither graphical dependencies nor dependencies on the Windows application programming interface. The shared libraries must be installed in the EAServer *lib* directory on the UNIX server. The Java classes required for the PowerBuilder JDBC driver must be installed in the EAServer *html/classes/com/sybase/powerbuilder/jdbc* directory.

Connection caches on UNIX PowerBuilder components require the database drivers listed inTable 41-16 in order to use connection caches. When a PowerBuilder component uses a connection cache, the appropriate PowerBuilder driver is loaded.

PowerBuilder COM servers

In addition to the COM server file generated in PowerBuilder, the PowerBuilder runtime files listed in Table 41-17 should be installed on the computer on which the server will run if the component uses the feature listed in the table.

Table 41-17: PowerBuilder files required on COM server host

Name	Description
pbvm110.dll	PowerBuilder virtual machine (required for all PowerBuilder components)
pbcomrt110.dll	PowerBuilder COM runtime
pbshr110.dll	Required by *pbvm110.dll*
libjcc.dll	Required by *pbvm110.dll*
libjutils.dll	Required by *libjcc.dll*
pbdwe110.dll	Required if the component uses DataStores
pbrtc110.dll and additional runtime files listed in Table 41-5	Required if the component uses Rich Text
pbodb110.ini	PowerBuilder ODBC initialization file required for database connectivity
pbodb110.dll	PowerBuilder ODBC interface required for database connectivity

Installed path *\Program Files\Sybase\Shared\PowerBuilder*

Deployment path In the same directory as the COM server or a directory on the system path.

Registry entries The PowerBuilder COM server is self registering, which means that you can use the REGSVR32 utility to register it on the computer on which it will be used.

Notes For information about deploying PowerBuilder COM servers, see “Deploying a PowerBuilder COM server” on page 576.

Database drivers

If your component connects to a database, you must also deploy the appropriate database drivers. You can use any database driver with a COM component, but you *must use ODBC* if you want to take advantage of COM+ support for transaction management and connection pooling. If you require support for connection pooling only, you can use any thread-safe ODBC driver. If you also require support for transactions, you must use a driver that supports the Microsoft Distributed Transaction Coordinator (DTC), such as the Microsoft ODBC driver for Oracle or the Microsoft ODBC driver for SQL Server.

For more information about database drivers, see “Database connections” on page 777.

PowerBuilder automation servers

To use a PowerBuilder automation server, you need to deploy the following files:

- The PBD or DLL containing the implementation of the automation server
- The generated type library file if you chose to create one or, if you built a named server, the *pbaen110.tlb* file that provides type library information for PowerBuilder.Application
- A copy of the registry update (*.reg*) file generated in the Project painter, edited to reference the directory where you will deploy the PBD or DLL and the type library file
- The PowerBuilder runtime and database connectivity files required by the server as described in “PowerBuilder COM servers” on page 796

After deploying the files listed above, run the registry update file to register the server on the target computer. For more information about using automation servers, see “Deploying an application that uses the automation server” on page 401.

Creating registry information for OLE automation objects

If your application includes user objects that are used for OLE inbound automation, you must update the user’s registry with information about the server. The Automation Server Project wizard sets up a project that helps you to generate a globally unique identifier (GUID), a registration (REG) file, and a type library (TLB) file. The Automation Server project wizard is available on the Project page of the New dialog box.

For more information, see “Deploying an application that uses the automation server” on page 401.

Web DataWindow on EAServer

You can run the Web DataWindow server component on EAServer using JSP as the page server. The component transaction server and the page server can run on the same or on different computers.

Files required on the transaction server

Two types of files are required on the transaction server:

- The PBLs or PBDs containing the definitions of your DataWindow objects

  Make sure they are installed in a directory in the server's path. If EAServer is running as a service, they must be on the system path or the names of the PBLs or PBDs must be fully qualified.

- PowerBuilder runtime files (including *pbvm110.dll*, *pbshr110.dll*, *pbjag110.dll*, and *pbdwe110.dll* on Windows) and *pbdwr110.pbd*, which contains the implementation of the DataWindow HTMLGenerator110 component

  You can use the PowerBuilder VM installer, available in the PBVM folder on the CD, to install these files. For more information about required files, see "PowerBuilder components on EAServer" on page 794.

You also need to create a connection cache for the database the component will access. For more information, see the *DataWindow Programmer's Guide*.

Custom components
You can create a custom version of the DataWindow component and configure its properties for more efficient reuse. For more information, see the *DataWindow Programmer's Guide*.

Files required on the dynamic page server

To use JSP as the page server and connect to the EAServer component using Java, you must have the following files on the JSP server computer:

HTML pages, templates, and scripts These are the files that you created for your application. If you used a JSP target to generate these files, you can use the built-in deployment controller to install them. For more information, see *Working with JSP Targets*.

EAServer client software for Java The files listed in Table 41-18 are required on the JSP server.

Table 41-18: EAServer client files required on JSP server

Name	Description
easclient.jar	Java class required for clients
easj2ee.jar	Java class required for J2EE support

Sun Java Development Kit (JDK) Any versions of the JDK that were installed with EAServer are located in the *Sybase\Shared\Sun* directory. The JDK that is installed with PowerBuilder is located in the *Sybase\Shared\PowerBuilder* directory.

If you are using JDK 1.2 or 1.3, make sure that the *JRE\bin\classic* subdirectory of the JDK is listed in the system PATH environment variable.

If you are using JDK 1.4 or later, make sure that the *JRE\bin\client* subdirectory of the JDK is listed in the system PATH environment variable.

Web DataWindow on COM+ or IIS

You can run the Web DataWindow server component on COM+ using ASP as the page server. You can also run the Web DataWindow on a Microsoft IIS application server.

Files required on the COM+ or IIS server

Two types of files are required on the COM+ or IIS server:

- The PBLs or PBDs containing the definitions of your DataWindow objects.

  Make sure that they are installed in a directory on the *system* path.

  Use the system path and system DSNs
  Because COM+ and IIS both run as system services, required files must be available on the system path and data sources must be defined as system DSNs. The user path and user DSNs are not referenced by the server.

- PowerBuilder runtime files and *pbdwr110.dll*, which contains the implementation of the DataWindow HTMLGenerator component.

Table 41-19: PowerBuilder files required on COM+ or IIS server

Name	Description
pbvm110.dll	PowerBuilder virtual machine
pbshr110.dll	Required by *pbvm110.dll*
libjcc.dll	Required by *pbvm110.dll*
libjutils.dll	Required by *libjcc.dll*
pbdwe110.dll	DataWindow support
pbodb110.dll	PowerBuilder ODBC interface
pbodb110.ini	PowerBuilder ODBC interface setup file
pbdwr110.dll	DataWindow HTMLGenerator component

Installed path If PowerBuilder is not installed on the COM+ or IIS server computer, these files must be installed in a directory on the *system* path on the COM+ or IIS server. They can be obtained from the *Sybase\Shared\PowerBuilder* directory on the computer on which PowerBuilder is installed.

Registry entries If *pbdwr110.dll* was copied from another computer, it must be registered on the COM+ or IIS server.

Notes ODBC data sources must be defined as *system* DSNs. For more information about configuring ODBC, see “Configuring ODBC data sources and drivers” on page 782.

If COM+ is hosting the server component and running on a different computer from IIS, you need to create a client install package and install it on the IIS server.

Index