Building Applications

VERSION 4.0

PowerBuilder

December 1994

Contents

About This Manual

Subject

This how-to manual guides you through the process of using PowerBuilder to develop and maintain graphical client/server applications. It does this by:

- Walking you through the **lifecycle** of an application and exploring the individual phases that make up an application development project
- Equipping you with collections of **techniques** for implementing many common application features, along with advice for choosing those techniques best suited to your requirements

Audience

You should read this manual if you're involved in any phase of an application development project that uses PowerBuilder.

Sample application

Accompanying this manual is a sample Order Entry application (abnc_ord) that's used to illustrate many of the issues, features, and techniques you'll be reading about. It's a good idea to examine the components of this application in PowerBuilder as you encounter references to them. That way, you'll be able to see the comments in those components and experiment with real, working examples of what you're trying to learn.

> **Installing the sample application**
> You probably installed the sample Order Entry application when you installed PowerBuilder. But if not, you can do so now.
>
> For more information, see the *Installation and Deployment Guide.*

PART ONE

Planning a Project

A walk through the various phases that make up an application development project when you're using PowerBuilder. Includes: introduction, designing an application, setting up an application, developing an application, and deploying an application.

CHAPTER 1

Introduction

About this chapter

This chapter tells you what you need to do to start planning an application development project.

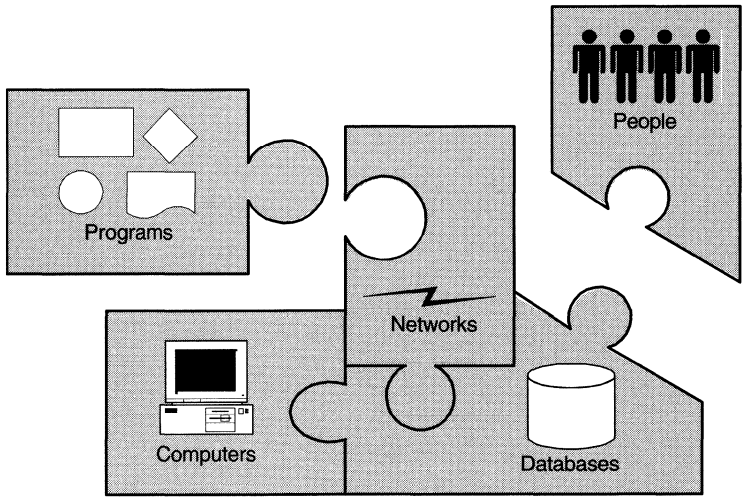
Contents

Where to begin

You've got the job of building, revising, or maintaining a graphical client/server application. And you've got PowerBuilder to help you. Now you need to determine the best way to carry out the project.

The pieces of the puzzle

One of the key characteristics of client/server applications is that they involve getting a variety of disparate and dispersed pieces to fit together. These pieces include computers, networks, databases, programs, and people.



Learning about these pieces

As a result, the best way to begin your project planning is to gather all of the pertinent information about the pieces that must play a role in your application.

Gathering information about the project

To figure out the parameters of your project, you need to ask a range of questions. The answers you find to these questions will help you determine how to use PowerBuilder over the course of your project to produce the desired application.

General questions

- **What is the *purpose* of the application?**

  It's important to determine the business goal that the application is intended to meet. For instance, its purpose might be to handle an institution's payroll, or to control and report on product inventory, or to provide an informational kiosk that customers can use in a store.

- **Who are the target *users* of the application?**

  You should establish a basic profile of the people who'll be using the application, including their requirements and skills. For example, they might be data entry clerks with minimal computer skills, or customer service representatives for whom quick response time is critical, or executives with particular report layout needs.

- **What is the required *delivery date* for the application?**

  Good scheduling practices can greatly benefit client/server application development projects. That's because these projects typically involve a lot of coordination of pieces and tasks.

- **What is the *current status* of the application?**

  It might be a brand-new application. Or maybe the application already exists and you need to make some maintenance fixes to it or create a new version of it.

  In the case of a new application, you should consider whether this is a one-time quick-and-dirty effort, or the start of a more substantial long-term effort (where it's worth investing more time in groundwork).

- **What is the *staffing* for the development project?**

  Staffing your development project is a matter of matching up the tasks to be done with the skills of the people you have available to do them. You'll learn more about this in just a moment.

Technical questions

- **What is the hardware/software *environment* in which the application must work?**

  The combinations of hardware and software that a client/server application might work with are virtually limitless. To determine what's appropriate for your application, you need to consider such topics as these: the platform(s) on which the client portion of the application is to run, the server database(s) that the client portion is to access, the network(s) that provide this access, other programs (applications, utilities, services) with which your application is to interact.

- **What are the *conventions and standards* you must adhere to during the development project?**

  Another way to ease coordination work during a client/server application development project is to spell out any conventions and standards up front. For instance, you might have conventions and standards to follow for user-interface design, coding style, documentation, error processing, and more.

- **What *reusable materials* do you have available for use in the application?**

  PowerBuilder makes it easy for you to reuse components you've developed (or acquired) for a previous application. So whenever you start a new project, it's wise to determine which features you can implement by using existing components instead of creating them from scratch.

- **What additional *development tools* do you have (besides PowerBuilder) to assist you with the project?**

  If you have any development support tools available (such as version control tools, CASE tools, or testing tools), you should figure out when and how you intend to use them during the course of the project.

Sample project-planning form

You might consider creating a form that you can fill in at the start of a project to gather and organize your planning information. For example:

Project Planner

Purpose of application: _____

Target users: _____

Delivery dates: _____

Current status of application: _____

Staffing: _____

Hardware/software environment: _____

Conventions/standards to follow: _____

Existing components to use: _____

Support tools for development: _____

Unless you're working independently, you'll usually need to talk with one or more other people to fill in all of this information. In that case, it's probably time to begin assembling your project team.

Assembling the project team

Client/server development projects require a number of different *skill sets* to handle the various pieces involved in an application.

If you work in an organization where job roles are specialized, your project team will typically include multiple people, each of whom contributes one or more of those skill sets. The success of your project will depend a lot on how well the efforts of these team members are coordinated.

If you work independently, you must juggle all of the skill sets yourself. The success of your project will depend a lot on your proficiency with each of them.

Client/server skill sets

The skill sets you'll usually need over the course of a client/server application development project include:

Skill set	What it involves
Project management	Decision-making, scheduling, coordination
End-user representation	Understanding of what users want the application to do and how they want to interact with it
System design	Architecting of the application to meet its requirements and to fit well in the computing environment where it is intended to operate
Database administration	Development and management of test and production databases
Network administration	Configuration and monitoring of server computers and the networks that connect client computers to them
Standards control	Establishment and enforcement of conventions and standards for such things as user-interface design, coding style, documentation, and error processing
Object management	Administration of the application components that developers create with PowerBuilder (with particular attention to facilitating the reuse of these components in multiple applications)
Application development	Creation and maintenance of application components with PowerBuilder (which includes painting them and coding logic for them), proficiency with SQL, familiarity with the client computer's operating system and any other programs to be accessed from the application

Skill set	What it involves
Documentation	Writing of comments, documents, or online Help about the application for reference by developers or end users
Multimedia artistry	Creation of pictures, sounds, or other multimedia elements to be used in the user interface of the application
Quality assurance	Testing and debugging of the application

At this point in your planning, you should know something about the application to be built and the project team who'll be helping you build it. Now you can start sketching out the major phases that will make up your project.

Sketching out the phases of the project

You can think of a particular development project as a walk through the **lifecycle** of your application. When you create a new application, you are in essence stepping through the first iteration of its lifecycle. If you later need to build an updated version of that application, you step through its lifecycle again.

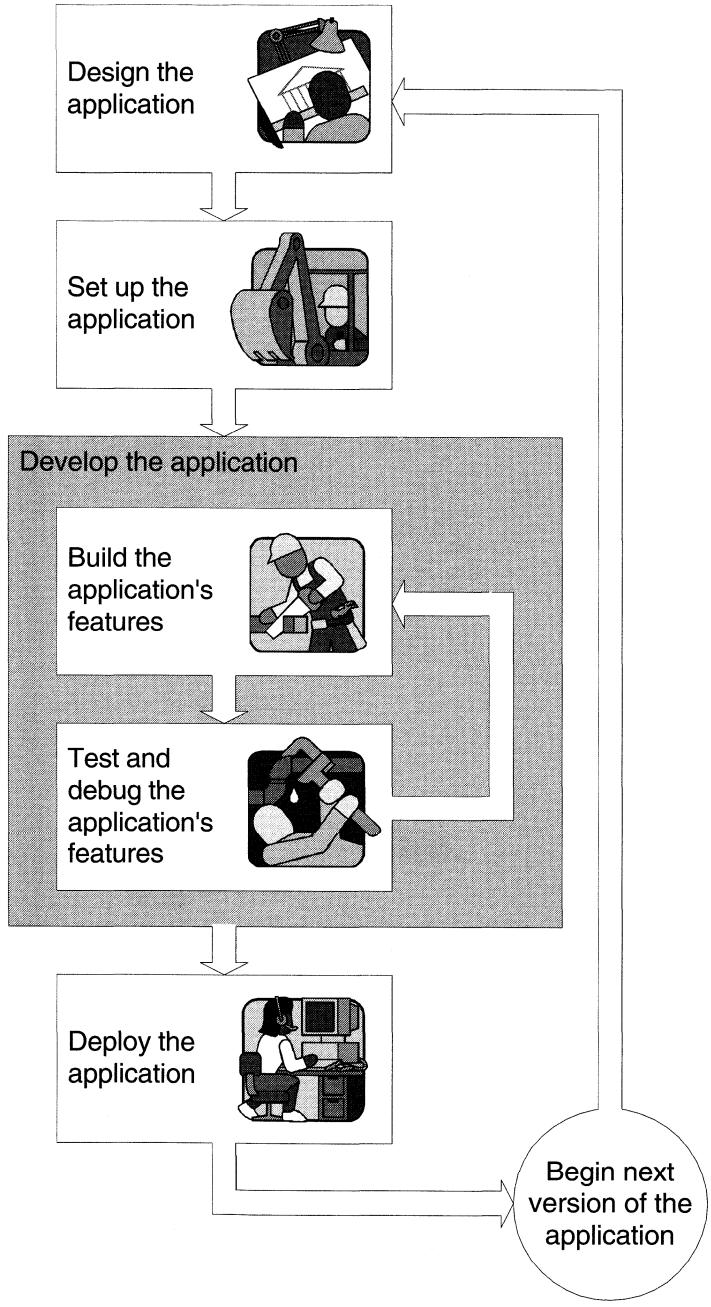
Exploring the application lifecycle

In either case, the major phases of work that make up the lifecycle are the same:

Phase	What it involves
Design	Determining your requirements for the application (including its processing and presentation)
	Drawing up a blueprint for how you'll meet those requirements by implementing particular application features with PowerBuilder
Setup	Doing the groundwork necessary to ensure a successful, orderly project and a well-constructed application
Development	Building the features you've designed for the application
	Testing and debugging those features
Deployment	Packaging up the completed application and distributing it to end users

How the phases fit together

Here's an illustration of the application lifecycle that shows how you'll proceed through these phases as you work on a development project.

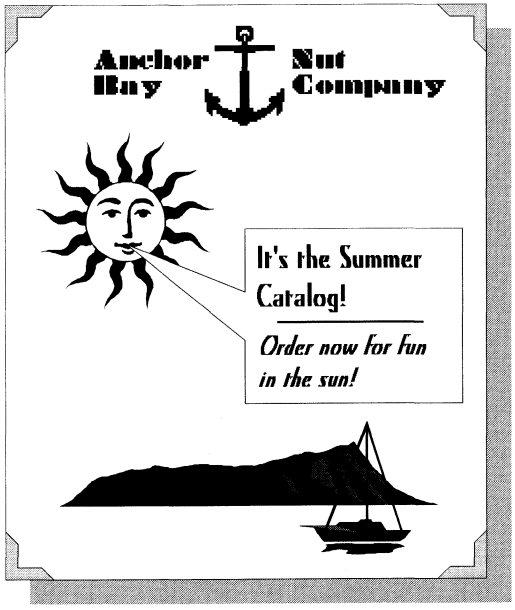

Design the application
Set up the application
Develop the application
Build the application's features
Test and debug the application's features
Deploy the application
Begin next version of the application

Looking at a sample project

To get a better idea of how all of this information gathering, team building, and planning actually works when you're starting a project, consider the following example.

A company in need of an application

The Anchor Bay Nut Company is a moderately sized enterprise in the business of selling a variety of delicious nut products. Most sales are made through its eye-catching catalog, which is mailed quarterly to thousands of homes.



Not long ago, the company realized that it needed to implement a new system for handling the orders that its phone sales people were taking from catalog customers. Knowing from the outset that this new system should have a graphical user interface and fit into the company's client/server environment, they initiated a project that would use PowerBuilder to construct an appropriate Order Entry application.

How they gathered their project information

The first thing they did to get the project going was to collect and organize the basic facts about the requested application and the pieces it would involve:

Project Planner

Anchor Bay Nut Company

Purpose of application:	Order entry application for phone-in catalog orders.
Target users:	Phone sales staff (these people are already using several other GUI applications).
Delivery dates:	Must be in production by Sept. 1 (start of busy season). Beta test end of July?
Current status of application:	New application, replacing manual paper-based system (look at the old forms to figure out data input and output needs).
Staffing:	Need people from DBA staff, operations, app development, maybe freelance artist and doc writer, user rep from phone sales staff.

PAGE 1

Project Planner

Anchor Bay Sat Company

Hardware/software environment:

User machines are PCs with MS Windows. Also using MS Mail and MS Word.

They have some Macs too (may want to run the app on these in the future).

Server stuff – Need to access the ANCHRBAY database on the server as main source of data (maybe also ABNCSALE database for supporting info). Network traffic will be about 12 - 15 concurrent users (but expected to grow to 25 next year).

Conventions/standards to follow:

See company standards manual. (Ask George about finalizing the UI conventions.)

PAGE 2

Project Planner

Anchor Bay Nut Company

Existing components to use:	Should be able to use at least a couple of our existing PowerBuilder libraries on this project:
	* Use ABNC_COM.PBL as the app framework (ancestor objects to inherit from).
	* Use ABNC_GDE.PBL to provide the same guide services in this app that we put into our other Sales apps.
	(Elaine is now handling our object management – ask her for details about these libraries.)
Support tools for development:	Use the new version control system. Look into getting some testing tools that work with PowerBuilder.

PAGE 3

How they assembled their project team

In the process of collecting this information, they also began assembling a project team of people from various groups within the company. Together, these team members would provide the various skills needed during the different phases of the project:

Skill set	Who provided it
Project management	Jerry (manager of application development staff)
End-user representation	Edna and Carlos (users from the phone sales staff)
System design	Elaine and Newman (application developers)
Database administration	Kramer (one of the company's two DBAs)
Network administration	Craig (from the operations staff)
Standards control	George (application developer)
Object management	Elaine
Application development	Elaine, Newman, and George
Documentation	Walter (a freelance technical writer)
Multimedia artistry	Claudette (a freelance artist)
Quality assurance	All of the above

How they carried out the project

As you continue through the remaining chapters in Part One of this manual, you'll learn details of what these team members did during each phase of the development project to build their new Order Entry application. You'll also read about later maintenance projects in which they built subsequent versions of the application (the most recent of which is included in your PowerBuilder package).

In Chapter 4, "Developing an Application," you'll examine components of the completed Order Entry application to see working examples of particular PowerBuilder techniques.

Where to go from here

Now that you've got a general idea of what a client/server application development project entails and how to launch one, it's time for a more detailed exploration of the individual project phases.

To learn about the first phase, turn to Chapter 2, "Designing an Application."

CHAPTER 2

Designing an Application

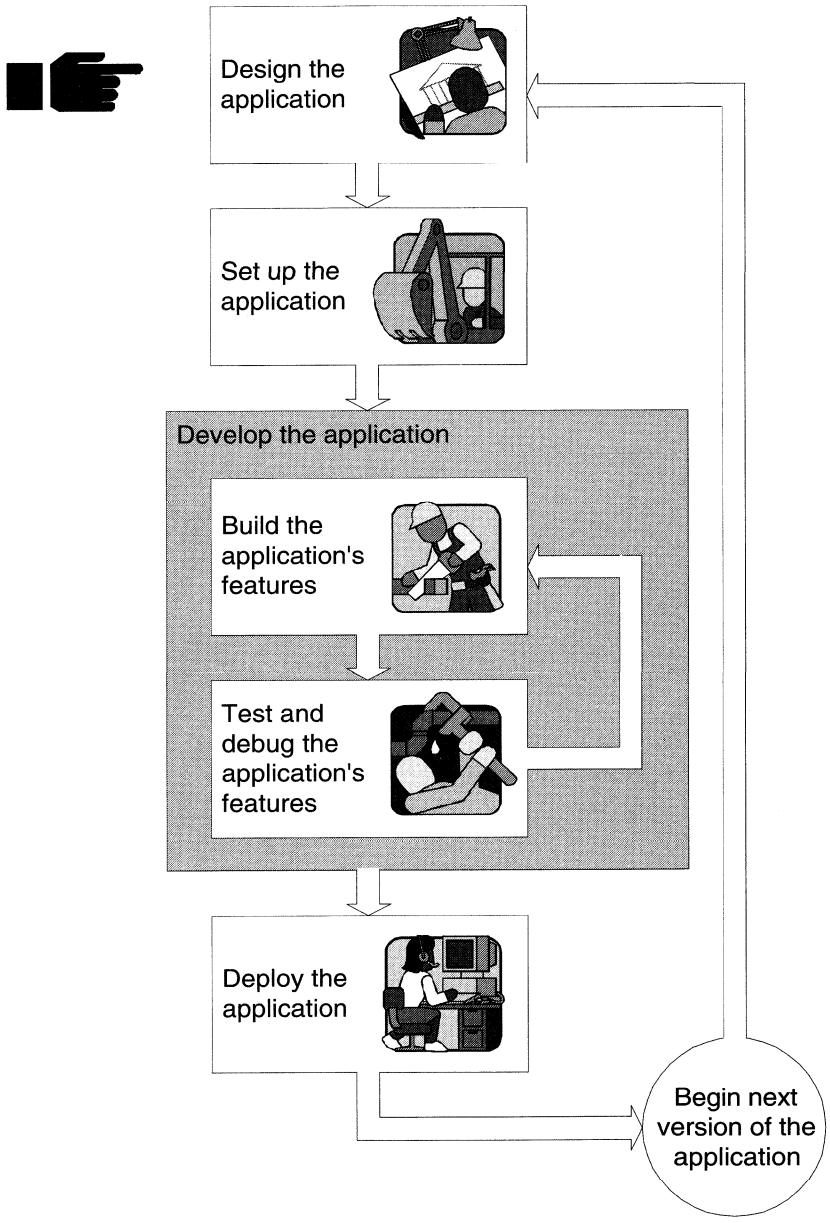
About this chapter

This chapter tells you about the process of designing an application that you want to build.

Contents

Orientation

Here's where you are now in the lifecycle of your application development project:


Design the application
Set up the application
Develop the application
Build the application's features
Test and debug the application's features
Deploy the application
Begin next version of the application

Determining the application's requirements

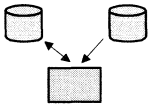
One of the keys to successful application development (in PowerBuilder or any other development environment) is designing what you're going to build before you begin building it. And the way to start this design phase is to determine the detailed requirements that your application must meet to satisfy the needs of its users.

Determining requirements is really nothing more than figuring out, in real-world terms, what the application is to do when someone uses it. Your goal should be to specify:

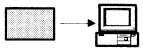
- A **conceptual walk-through** of the application, from the point when users start it to the point when they exit from it
- One or more **perspectives** of the application's composition, such as by flow, by components, by dependencies, or by usage

Kinds of requirements

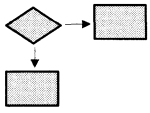
To figure out the requirements of a typical client/server application, it's helpful to break them down into several different categories:



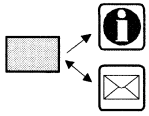
Data access Which database tables the application needs to use for its data input and output. Which basic database operations (create, retrieve, update, delete) the application needs to perform on these tables.



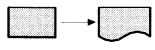
User interface How the application is to present this data to users. How users are to interact with the application to view the data, manipulate it, and navigate through the application's various displays.



Processing What the application is to do with the data to prepare it for display to the user or update to the database tables. What the flow of control is to be as the application performs its various operations.



Program interaction What data sharing (if any) the application needs to do with other programs. What additional services (if any) the application must get through the execution of other programs.



Output What reports the application needs to print and how it is to format them. What other materials (such as files) the application is to generate as a byproduct of its processing.

Ways to draft requirements

You can collect and organize your application requirements by using an assortment of different approaches. What matters most is that the product be something you can use (and maintain) easily and effectively through the entire project to guide your development work.

Some common approaches for drafting application requirements include:

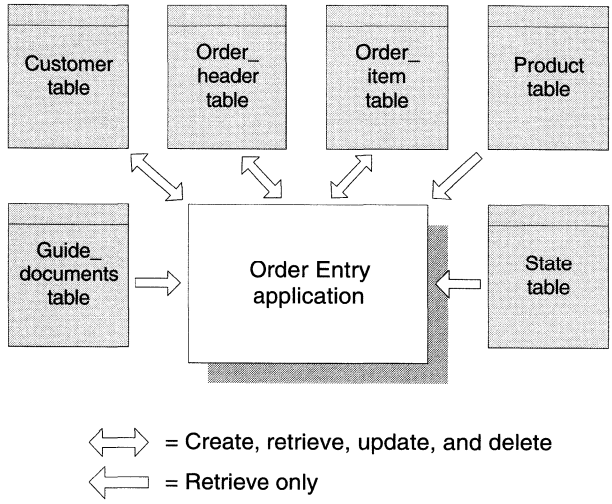
- Writing specification documents
- Drawing flowcharts
- Using CASE (Computer Aided Software Engineering) tools

> **Choosing a CASE tool**
> If you use a CASE tool that's made to work with PowerBuilder, it can help you in the next step of the design process (mapping the application's requirements to PowerBuilder features) as well as in the later phases of your project.
>
> For a list of these CASE tools, look in your PowerBuilder package or talk to a Powersoft sales representative.

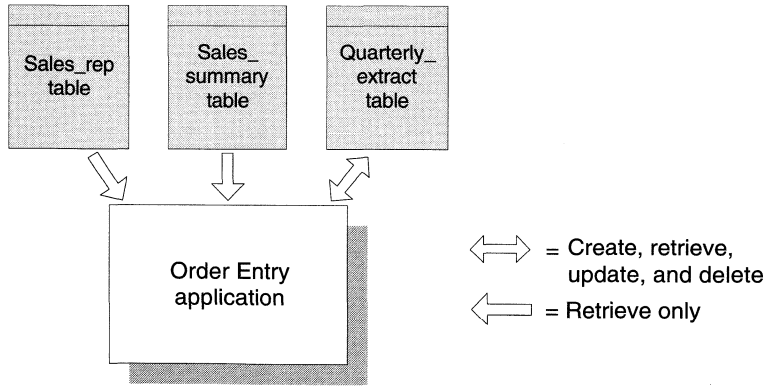
Sample application

Here are some of the requirements that the system designers at the Anchor Bay Nut Company came up with for their Order Entry application.

Data access The Order Entry application needs to access the company's main database (ANCHRBAY.DB) from the server so that it can use these tables:



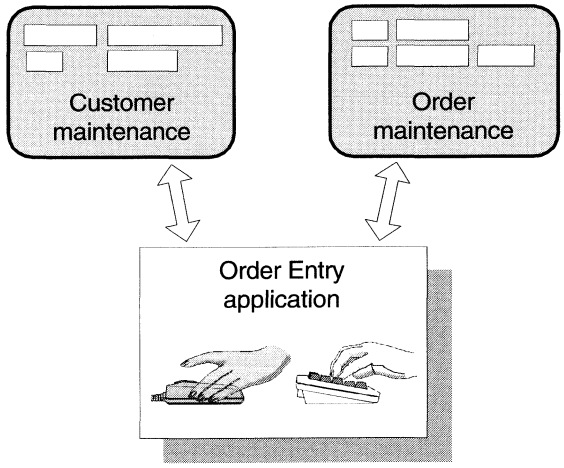
The application also needs to access the company's sales database (ABNCSALE.DB) from the server so that it can use these tables:



Both ANCHRBAY.DB and ABNCSALE.DB are Watcom SQL databases.

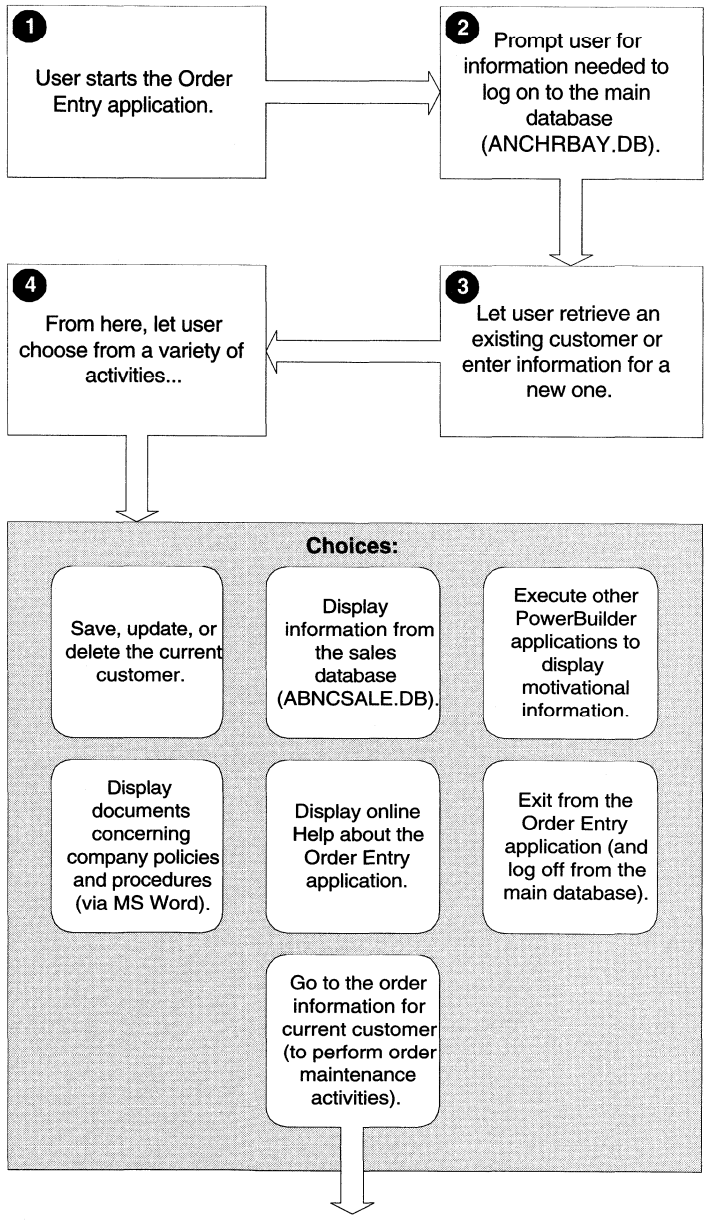
User interface The Order Entry application needs to present two major displays with which the user is to interact:

- One for viewing (retrieving) and maintaining (creating, updating, deleting) customer data in the Customer database table
- One for viewing and maintaining order data in the Order_header and Order_detail database tables



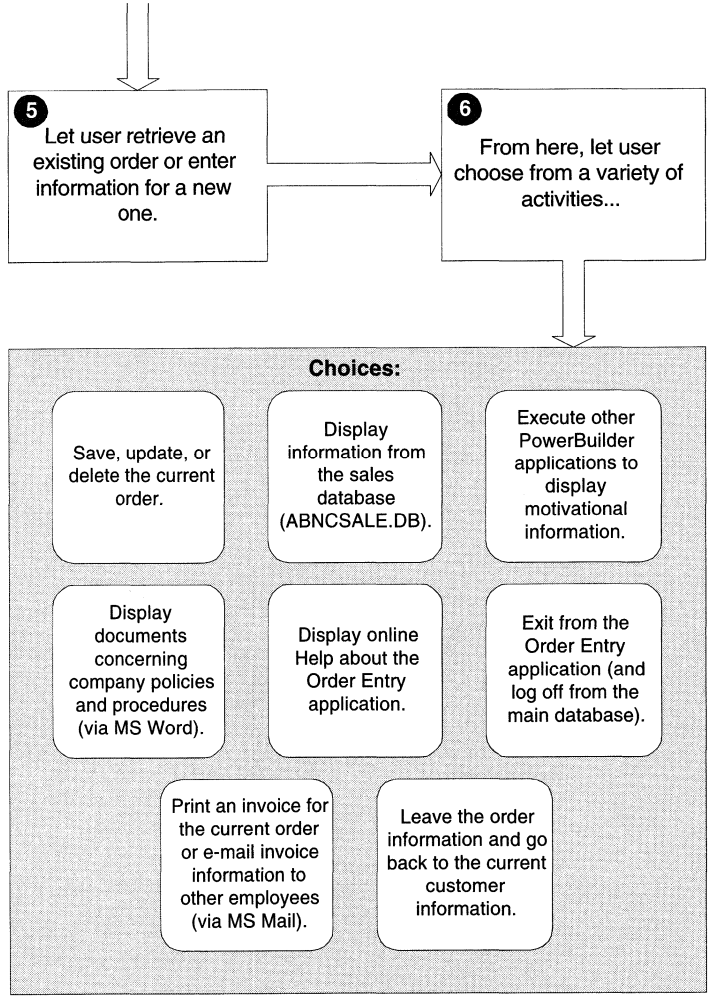
In the course of supporting these customer and order maintenance activities, the application must also present a number of more minor displays (such as for logging on to the databases, soliciting choices from the user, displaying messages, and more).

Processing Here's the basic flow of processing that the user should encounter when navigating through the Order Entry application (ignoring many of its supporting activities for the sake of simplicity):



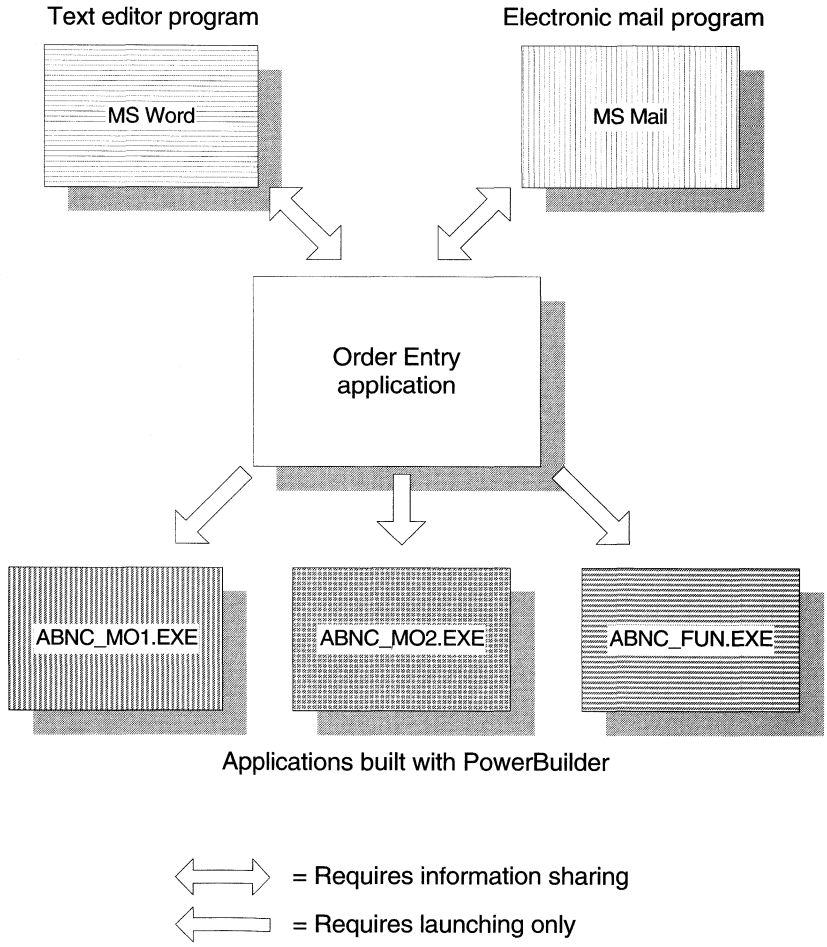
Continued on next page

Continued from previous page



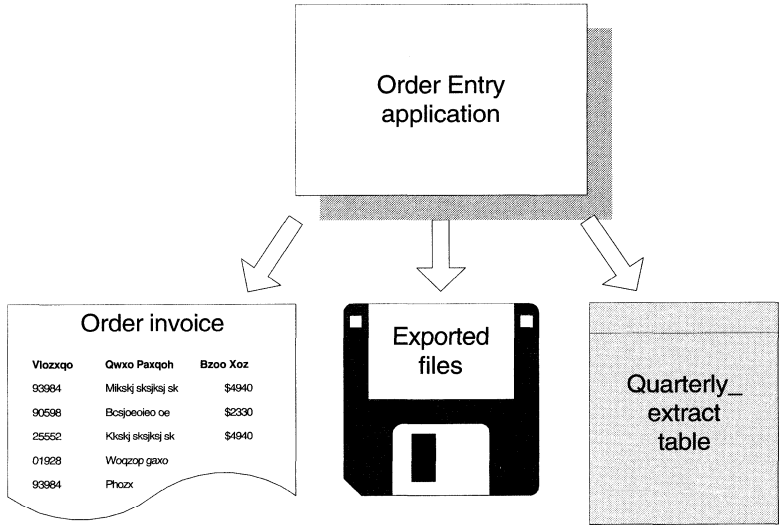
Program interaction In support of its customer and order maintenance activities, the Order Entry application also needs to access some external programs, including these:

- A text editor program (specifically, Microsoft Word) to let the user view documents concerning company policies and procedures
- An electronic mail program (specifically, Microsoft Mail) to let the user send order invoices to other employees
- Several existing applications (that were created with PowerBuilder) to display motivational information to the user



Output The Order Entry application must be able to produce various kinds of output when requested by the user. This includes:

- Printing an invoice for a particular order
- Exporting customer or order data to a file (using any one of several different file formats)
- Extracting quarterly data from the Sales_rep and Sales_summary tables of the sales database and storing that data in a new table (named Quarterly_extract)



Mapping the requirements to PowerBuilder features

Once you know the detailed requirements that your application must meet, you need to map them to features that PowerBuilder provides. In other words, you've got to figure out how the application will accomplish its data-access, user-interface, processing, program-interaction, and output work within the world of PowerBuilder.

The following sections will guide you through this portion of the design phase. You'll learn about:

- Accessing data: servers and other data sources
- Implementing the user interface: windows and controls
- Controlling the application's processing: events and scripts
- Extending the application's processing: external programs
- Routing application output: reports, files, and pipelines

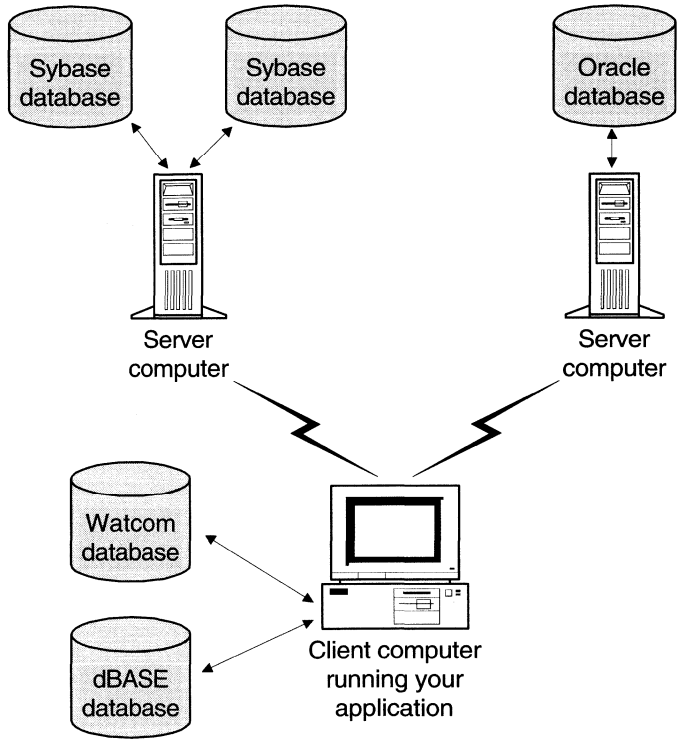
Accessing data: servers and other data sources

The requirements of your application probably call for access to at least one table. In fact, it's usually the case that applications need to access several.

What makes this access tricky in a client/server environment is that these tables are likely to be in different places and different formats. Specifically:

- The tables may be stored **in one or more databases**
- Those databases may be located **in a variety of locations**: on the client computer, on one or more server computers, or on a mix
- Those databases may be implemented **under a variety of DBMSs** (database management systems)

For example:

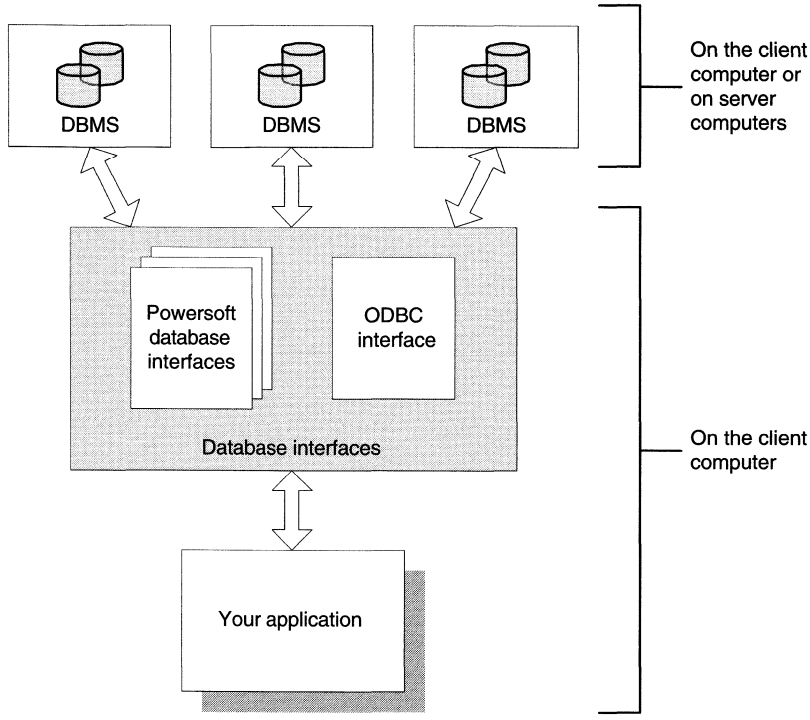


What features does PowerBuilder provide that you can use in your application to talk to these databases and get to the required tables?

Talking to databases with PowerBuilder

PowerBuilder provides an approach to data access that enables you to successfully handle this potential database diversity in the applications you build. It does this by separating the DBMS-specific aspects of data access from an application to make it as independent as possible. This means you can focus on the logical use of a table in your application instead of how that table is implemented in one particular database or another.

Introducing the database interfaces PowerBuilder handles DBMS specifics in a separate software layer that you install on the client computer along with your application. This layer consists of various database interfaces, each of which knows how to talk to a particular kind of DBMS and how to take advantage of the unique features of that DBMS. When your application requests any kind of access to a database, it relies on the appropriate database interface (depending on the DBMS for that database) to carry out the operation.



Kinds of database interfaces Notice that the preceding figure shows two different kinds of database interfaces that PowerBuilder provides for your application to use:

- **The ODBC interface** You'll design your application to use this interface if you want to access one or more ODBC-compliant databases.

  ODBC is the Open Database Connectivity API (application programming interface) developed by Microsoft to give applications standardized access to diverse data sources (which are usually databases, but can also be other kinds of files such as spreadsheets or text files). The ODBC interface that's included with PowerBuilder was developed by Powersoft to let your applications talk to ODBC, which in turn talks to the actual data sources.

- **The Powersoft database interfaces** Sometimes you won't want to access a particular database via ODBC or won't be able to because that database is not ODBC-compliant. For these cases, Powersoft offers a variety of native interfaces, each of which knows how to talk to a specific DBMS (such as SQL Server or Oracle).

So if you want to access a SQL Server database, for example, you'll design your application to use the Powersoft SQL Server interface.

You can design your application to use any combination of these database interfaces.

For more information on database interfaces, see *Connecting to Your Database.*

What this approach does for you The major benefit of this layered approach to data access is that it helps insulate your application from the complicated and potentially dynamic logistics of the typical client/server environment. As a result, the data access you design into your application can be:

- **Flexible** You can make your application independent of a database's location or DBMS. That way, if the database needs to be moved (from client to server, from server to client, or from server to server) or migrated to a different DBMS, then the application itself doesn't have to be disrupted. (This even helps you while developing the application by making it easy to switch between test and production versions of a database.)

  Adapting your application to such changes can often be just a matter of pointing it to the database's new location and database interface.

- **Consistent** You can work with all of the tables in your application in the same way—using the same table-processing features—regardless of the DBMSs that are involved. (You'll learn more about these table-processing features of PowerBuilder in just a moment.) That means you don't have to design DBMS-specific processing routines.

  Even in cases where you want to take advantage of capabilities unique to certain DBMSs (such as stored procedures, outer joins, referential integrity checking), you'll still use consistent PowerBuilder techniques to do so. Of course, if you're planning to migrate to a different DBMS later, you should make sure it also supports those capabilities (or else you'll need to modify the application).

Setting up the databases to access As you'd expect, before your application can access particular databases and their tables, they must exist. In lots of cases the necessary database design, definition, and population work may have already been done (by you or someone else such as a DBA) through DBMS facilities or other design tools.

But if this database setup work is not yet done or if some modifications are needed, you can use certain painters in PowerBuilder to help do the job.

For more information, see "Designing and defining your databases" in Chapter 3, "Setting Up an Application."

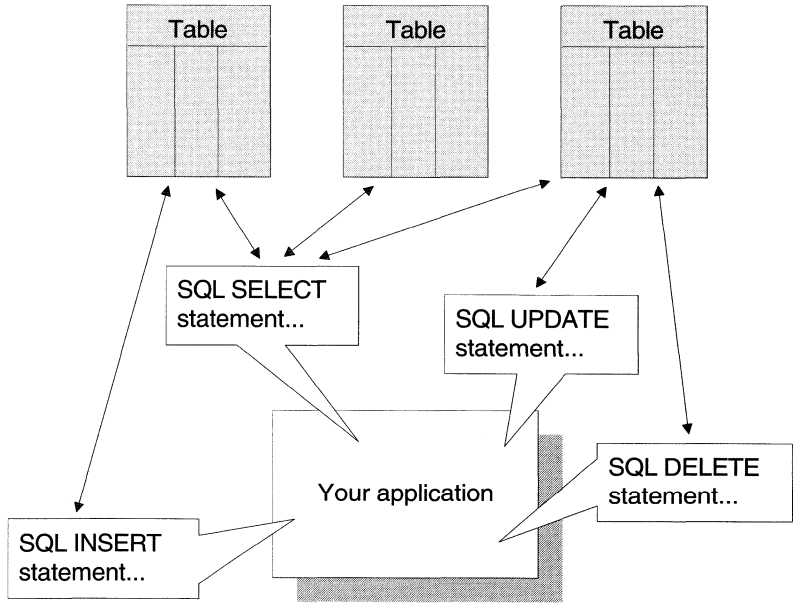
Doing the table processing you need

Now that you know something about the PowerBuilder features you'll use to talk to databases in your application, it's time to look at the features PowerBuilder provides for processing the tables in those databases.

In the blueprint of your application, you're likely to have numerous places where you need to create, retrieve, update, or delete rows in tables. In each case, you can choose from a couple of different approaches that PowerBuilder offers for implementing the required operations:

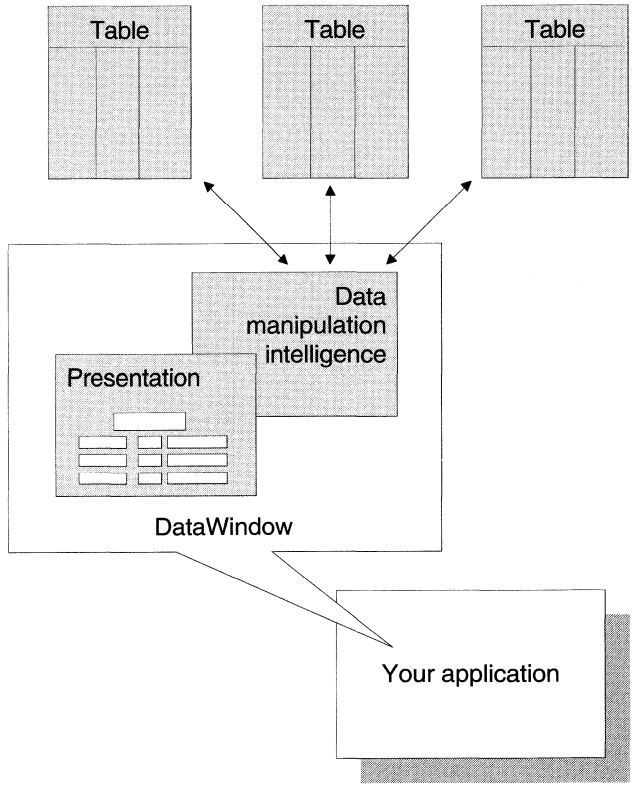
- You can embed **SQL statements** in your application to manipulate the rows.

  PowerBuilder supports all of the usual features of this industry-standard language, along with DBMS-specific syntax and some powerful extensions of its own. In general, you should think about using embedded SQL in places where your design calls for row manipulation without the need for display:



- You can use **DataWindows** in your application to manipulate the rows *and* display them to the user.

  DataWindows are a very special feature of PowerBuilder that you'll want to use for most of the table processing your application design requires. That's because they contain both the intelligence to do robust row manipulation (including creation, retrieval, updating, deletion) and the presentation (user-interface) abilities to let people see and work with those rows:



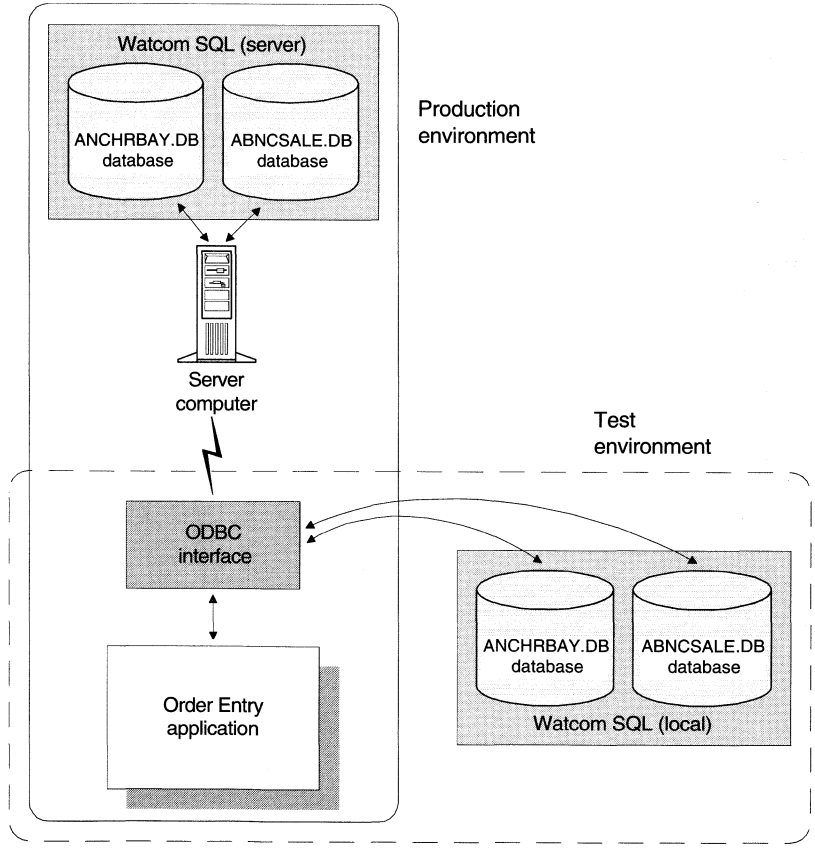
You'll learn a lot more about embedded SQL and DataWindows later in this manual.

Sample application

Here's how the system designers at the Anchor Bay Nut Company mapped their data access requirements to PowerBuilder features.

Talking to their databases As you read earlier, the Order Entry application needs to access two different Watcom SQL databases (ANCHRBAY.DB and ABNCSALE.DB) from their server computer. Because Watcom SQL is an ODBC-compliant DBMS, the application can talk to it through the PowerBuilder ODBC interface.

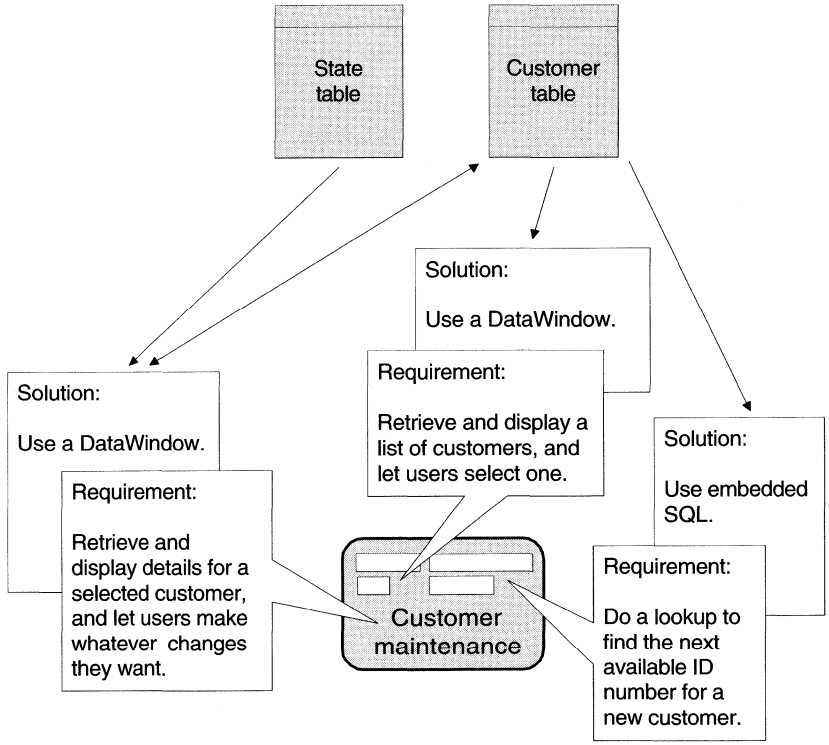
In addition to these requirements, there's one other logistical matter. While the application is being built, developers need to access *test* versions of ANCHRBAY.DB and ABNCSALE.DB stored locally on their individual computers, instead of the *production* versions stored on the server. This just means they'll need to point the application to the local DB files at the start of the project and then point it to the server DB files toward the end (leaving time for testing how the application runs against the server).



Your version of the Order Entry application
When you use the Anchor Bay Nut Company's Order Entry application on your computer, you're accessing their local versions of the two Watcom SQL databases, ANCHRBAY.DB and ABNCSALE.DB.

Doing their table processing The requirements for the Order Entry application specify that it must process several different tables from the two databases in a variety of different ways. In most cases, the processing involves letting the user display and/or interact with the table data, so the best solution is to use DataWindows. In the remaining situations (where no display of the data is involved), the developers can embed SQL statements in the application to do the job.

For example, here's the plan for handling some of the table processing operations in the customer maintenance portion of the application:



Implementing the user interface: windows and controls

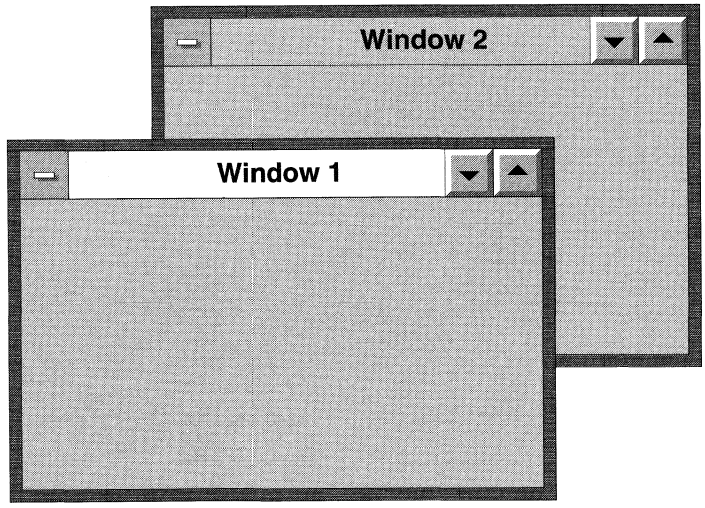
The user-interface requirements of your application probably involve enabling people to interact in a number of ways with data that the application accesses. As you design an interface to meet those requirements, you need to consider a variety of issues, including:

- How each kind of data is to be arranged on the screen
- How users are to interact with each kind of data
- How these interactions are to be organized into coherent activities (typically, business-oriented activities)
- How users are to navigate from one of those activities to another

What features does PowerBuilder provide to do this presentation, interaction, organization, and navigation work?

User-interface basics in PowerBuilder

To construct the user interface of an application in PowerBuilder, you'll use **windows** as the major building blocks. Windows in PowerBuilder look and work just like the windows you're already familiar with in your graphical operating environment (which means that they consist of rectangular frames that you can usually move and resize on the screen, and even lay on top of each other).

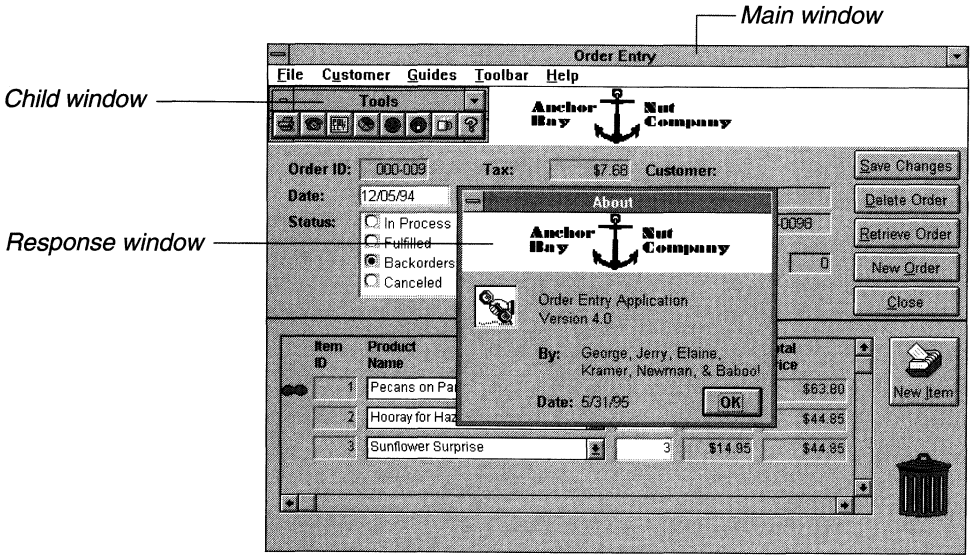


Windows provide features you can use to let people view information, manipulate information, and initiate actions.

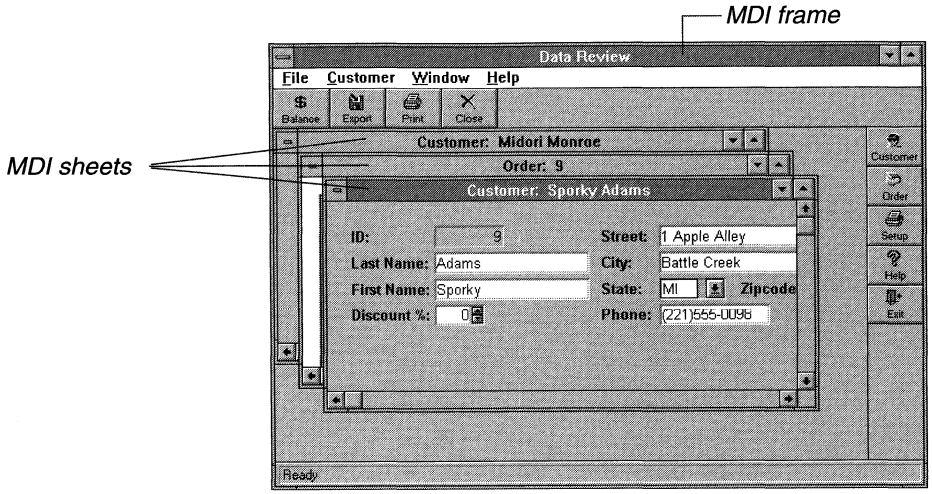
The role of a window You can design the user interface of an application to involve just one window. But most of the time, you'll involve several different windows, with each one playing a particular role to help the user get a larger job done. The role of any individual window is usually to present a particular kind of information and let users interact with that information in certain ways.

Types of windows PowerBuilder provides several different types of windows that you can use in your application. Each type has some unique characteristics that make it good for fulfilling one or more specific presentation or interaction needs:

- **Main windows** are where you'll usually have users perform the major activities of the application. You can think of them as home bases for those activities.
- **Response windows** and **message boxes** are good for situations where you want to force users to consider some information and/or choose some action before they can do anything else in the application.
- **Popup windows** and **child windows** are handy for displaying additional pieces of information, or providing additional services, that support the users' activities as they work in a particular main window.



- **MDI frames** are useful in many applications where users need a convenient and organized way to work with multiple main windows. When placed inside one of these frames, main windows act as **sheets** that users can easily shuffle around. (You'll learn more about what MDI is in just a moment.)



For more information on all of these window types, see the *User's Guide*.

Introducing controls What goes inside the windows you design? PowerBuilder provides a wide range of controls you can place in a window to hold the information users need to see and to implement the interactions users need to perform. There are controls for:

- **Displaying and/or manipulating values** These controls include:

  StaticText
  SingleLineEdit
  MultiLineEdit
  EditMask

- **Making choices** These controls include:

  ListBox
  DropDownListBox
  CheckBox
  RadioButton

- **Showing information graphically** These controls include:

  Graph
  HScrollBar
  VScrollBar

- **Dressing up a window** These controls include:

  GroupBox
  Line
  Oval
  Rectangle
  RoundRectangle
  Picture

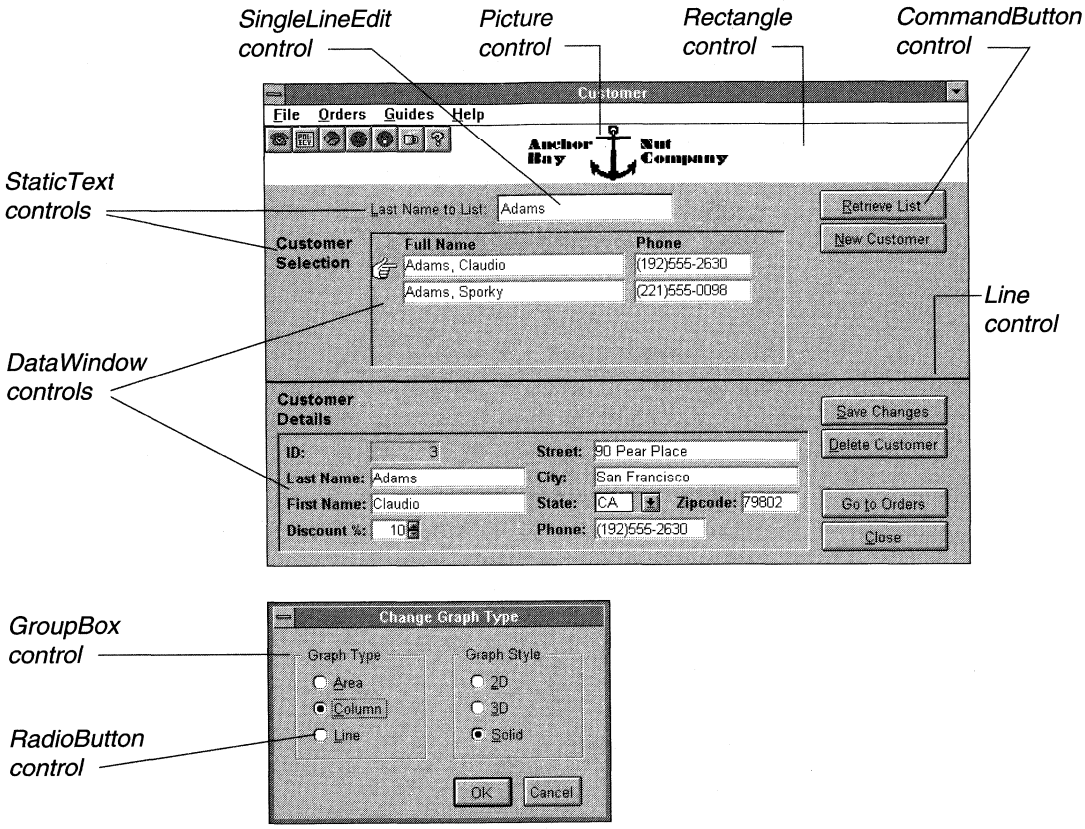
- **Presenting your DataWindows** PowerBuilder provides a special kind of control called a:

  DataWindow

  that you use when you want a window to display one of the DataWindows you've designed for your application. By the way, when you do design a DataWindow, you'll specify that it is to present the data it accesses in a style that's just like one or more of the controls you've just read about. As a result, the look and use of what's inside your DataWindow control will be consistent with the look and use of the other controls around it in the window.

- **Initiating actions** These controls include:

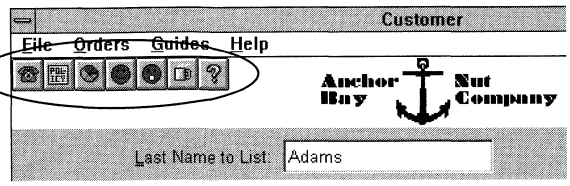
  CommandButton
  PictureButton



In addition to the preceding kinds of controls (which PowerBuilder provides as standard), you can also use:

- **Controls you define yourself** You can define your own controls based on one or more of the standard PowerBuilder controls and store them as application components called **user objects**. Then you can include these custom-made controls in any windows you want.

  For example, here's a custom-made toolbar control that is built from some standard PowerBuilder PictureButton controls:



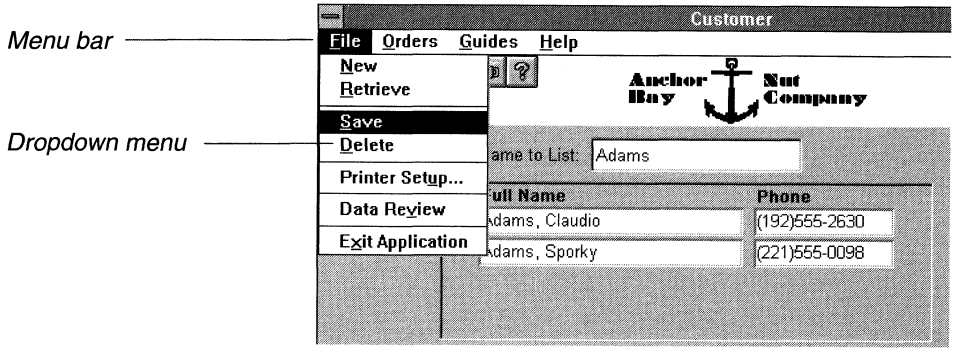
- **Controls you get from external sources** You can use controls that were created outside of PowerBuilder (such as VBX controls or controls in DLLs) by defining them as user objects too. Then you can include these external controls in any windows you want.

You can pretty much use any combination of any controls in a particular window, depending on the activities that the window is intended to support. And you can arrange them in whatever way best suits your needs.

For more information on the various kinds of controls, see the *User's Guide*.

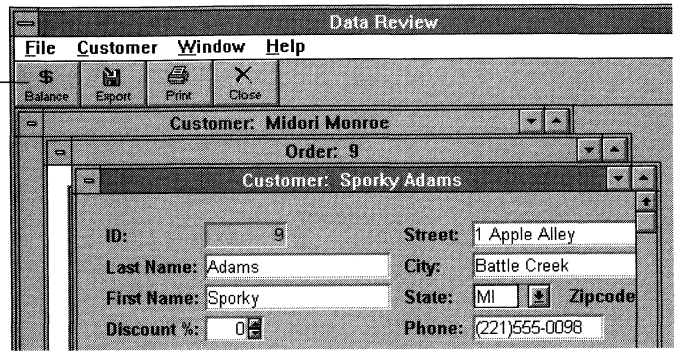
Introducing menus Another way to let the user initiate actions in a window is to use menus. A menu lists commands (**menu items**) that are currently available so that the user can select one. PowerBuilder provides a couple of different methods for presenting menus in a window:

- **On the window's menu bar** In many of the windows you design, you'll want to display a menu of one or more items across the top in the menu bar. This enables users to move through those items and pull down submenus (dropdown menus) that you've defined for each one:



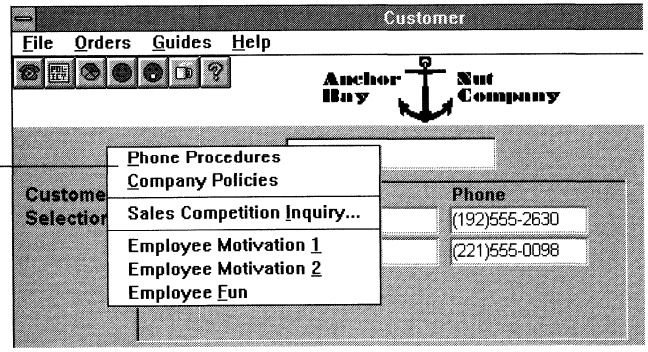
If the window is an MDI frame, you can optionally define a **toolbar** to accompany the menu. The toolbar displays buttons corresponding to one or more menu items, giving the user an alternative way to select those items:

MDI toolbar



- **Inside the window as a popup** Sometimes you may want to let users initiate certain actions by popping up a menu within the window. A popup menu lists its items vertically, enabling users to move up or down to select the item they want:

Popup menu



Popup menus can be handy for giving users quick access to a subset of the most common actions they perform in the window or to just those actions that apply to the current control.

Not all types of windows support menus (for example, response windows don't), but in those that do (such as main windows) you can use a menu bar, popup menus, both, or neither. It all depends on your design goals for the window.

For more information on the composition of menus, see the *User's Guide*.

Choosing a user-interface style

To design a successful user interface for an application—one that enables people to smoothly perform all of their required activities—you must do more than just draw up the individual windows that support those activities. You've also got to figure out the connections among your windows, including:

- How users are to navigate through the application, from one window to another
- Which windows should be available to use at a given moment
- Whether a particular window will depend in some way on one or more other windows

The best way to start addressing these issues is to decide on the user-interface style that your application is to follow.

Your choices There are three user-interface styles you'll usually consider using. Each one will give your application a particular look and feel:

Style	Description
SDI	Single Document Interface. In this style, users typically work with one main window at a time to perform an activity (although that window may display various popup or child windows to do supporting chores as a user works). When users want to perform a different kind of activity, they go to a different main window to do it.
MDI	Multiple Document Interface. In this style, users work within a frame that lets them perform activities on multiple sheets of information. MDI tends to be most useful in applications where users require the ability to do several different things at a time.
Combination	In this style, the application uses an SDI approach to implement some of its activities and an MDI approach to implement others.

Determining which style best suits your application depends on a lot of factors, among them: the nature of the application's data, the complexity of the application's processing, the preferences of the application's users, and the conventions (if any) you're obligated to adhere to. You'll learn more about making this determination later.

See "Establishing standards and conventions" in Chapter 3, "Setting Up an Application."

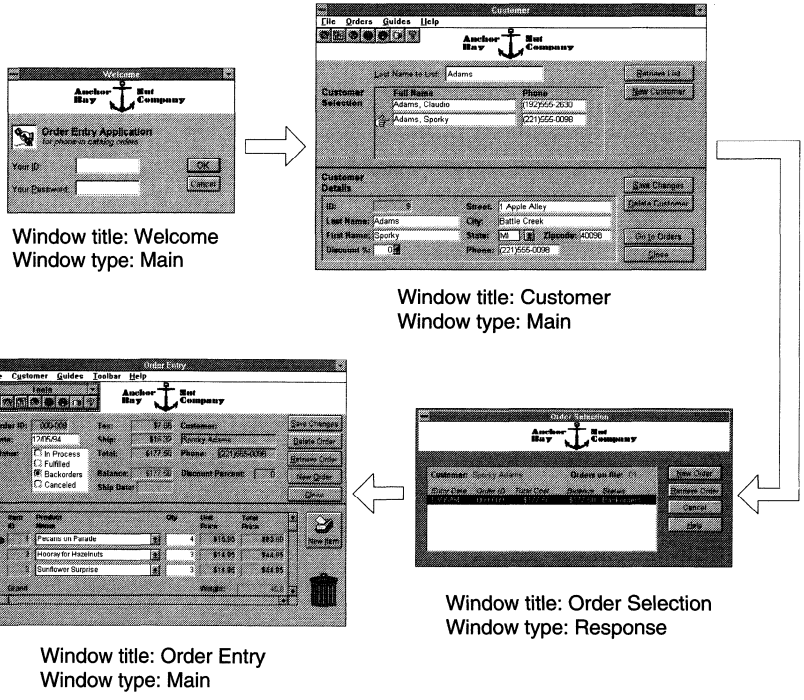
Sample application

Here's how the system designers at the Anchor Bay Nut Company mapped their user-interface requirements to PowerBuilder features.

Choosing their user-interface style Because of the diverse processing that the Order Entry application must perform, it needs to use a combination style of user interface. Specifically, it should:

- **Use an SDI approach** for the main activities that the phone sales staff will perform with it—entering customer and order information. SDI is chosen to keep things simple, since a salesperson usually needs to deal with just one customer at a time and then enter just one new order for that customer.
- **Use an MDI approach** for certain peripheral activities that the staff may occasionally perform while working in the application. One of these activities is the Data Review, which needs to let the user display information about multiple customers and orders at the same time.

Putting their windows together To perform its main activities, the application requires the following major windows (along with a number of more minor supporting windows that aren't shown):



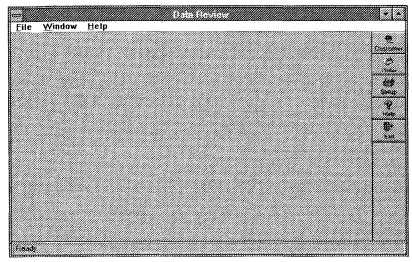
Window title: Welcome
Window type: Main

Window title: Customer
Window type: Main

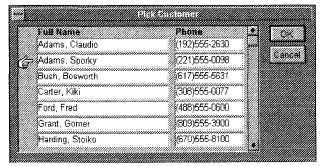
Window title: Order Selection
Window type: Response

Window title: Order Entry
Window type: Main

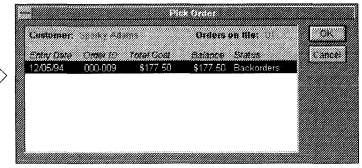
For comparison, take a look at the major windows required for the Data Review portion of the application:



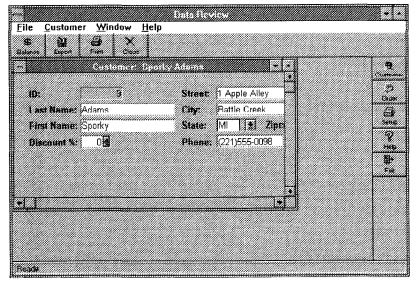
Window title: Data Review
Window type: MDI frame



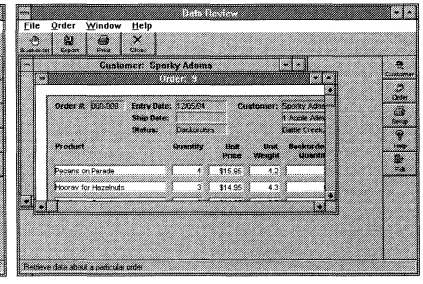
Window title: Pick Customer
Window type: Response



Window title: Pick Order
Window type: Response



Window title: Customer *name*
Window type: Main (used in the frame as an MDI sheet)



Window title: Order *number*
Window type: Main (used in the frame as an MDI sheet)

Controlling the application's processing: events and scripts

Now that you've sketched out the windows you need, it's time to take a closer look at how they must work to meet your application's processing requirements. Specifically, you have to think about how you'll:

- Control the **flow of processing** in your application, both within a window and from one window to another
- Supply the specific **processing logic** that is to be performed for each activity of the application during the course of this flow

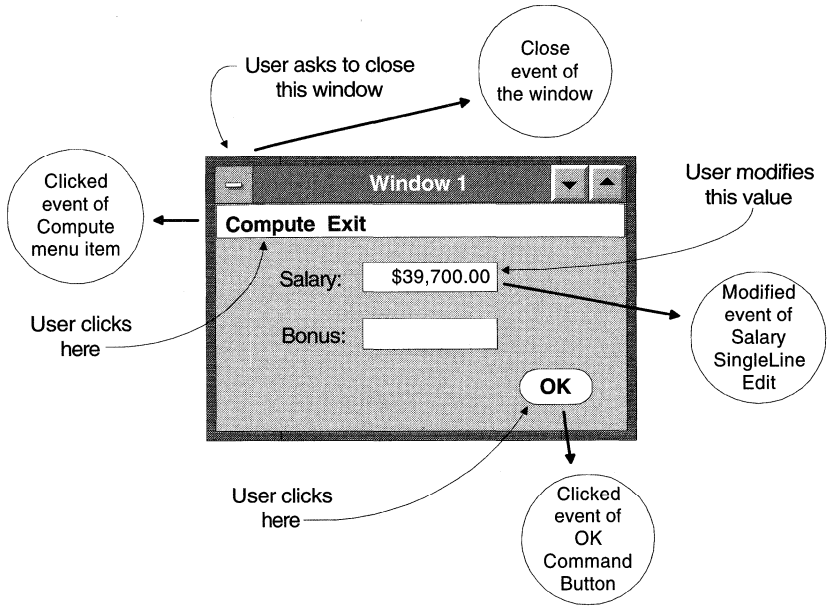
What features does PowerBuilder provide to take care of these processing-related aspects of an application?

Controlling the flow of processing

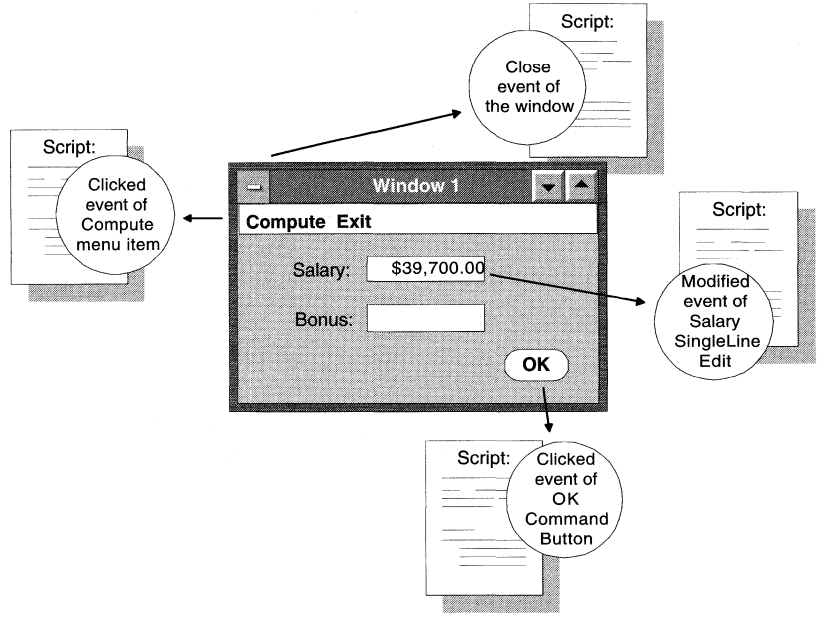
PowerBuilder provides an approach to flow-of-control in an application that puts users in charge. PowerBuilder applications are **event-driven**, which means that they wait to see what actions a user takes in a window to determine what processing to perform.

How it works Whenever the user does something involving one of the application's user-interface components (such as a window, control, or menu item), that action triggers a particular **event**. For example, each of the following actions triggers a different event:

Doing this	Triggers
Clicking on a particular CommandButton control in a window	The Clicked event of that CommandButton control
Clicking on a particular menu item in a window's menu	The Clicked event of that menu item
Modifying the value in a particular SingleLineEdit control of a window	The Modified event of that SingleLineEdit control
Closing a particular window	The Close event of that window



What an event does When an event is triggered, your application executes a corresponding **script**, which contains any processing logic you've written for that event:



You'll learn more about scripts in just a moment.

Determining which events to work with Your job as a designer is to figure out all of the events of interest that might occur in your application and to provide appropriate processing logic for each event (in its script). To do that, you've got to know more about the various kinds of events there are.

Each kind of user-interface component has its own set of several different events that can happen to it. For instance:

This component	Has
A CommandButton control	About a dozen different events, including: Clicked, GetFocus, and LoseFocus
A menu item	Just a couple of events: Clicked and Selected
A SingleLineEdit control	About a dozen different events, including: Modified, GetFocus, and LoseFocus
A window	More than 25 different events, including: Open, Close, Resize, Timer, and Clicked

There are even some events that apply to your application as a whole, including:

- One that's triggered **when the application starts** (the Open event).

  You'll normally use the script for this event to specify the initial window you want the application to display.

- One that's triggered **when the application ends** (the Close event).

Although this quickly adds up to a lot of events, you don't have to worry about handling every one. In many cases, you'll need to write scripts for just one or two of the events of a particular component (and sometimes you won't need any for that component).

For more information about the events for each kind of user-interface component, see *Objects and Controls*.

If you need to take control Letting users drive the flow of processing is appropriate most of the time, but on occasion you'll want the application to temporarily take control. In these situations, you can write code in the script of one event that *manually* causes another event to occur. When doing this, you can either:

- **Trigger the event** so that its script executes right away, or

- **Post the event to a queue** so that its script execution is deferred (until after the scripts of any earlier events have executed)

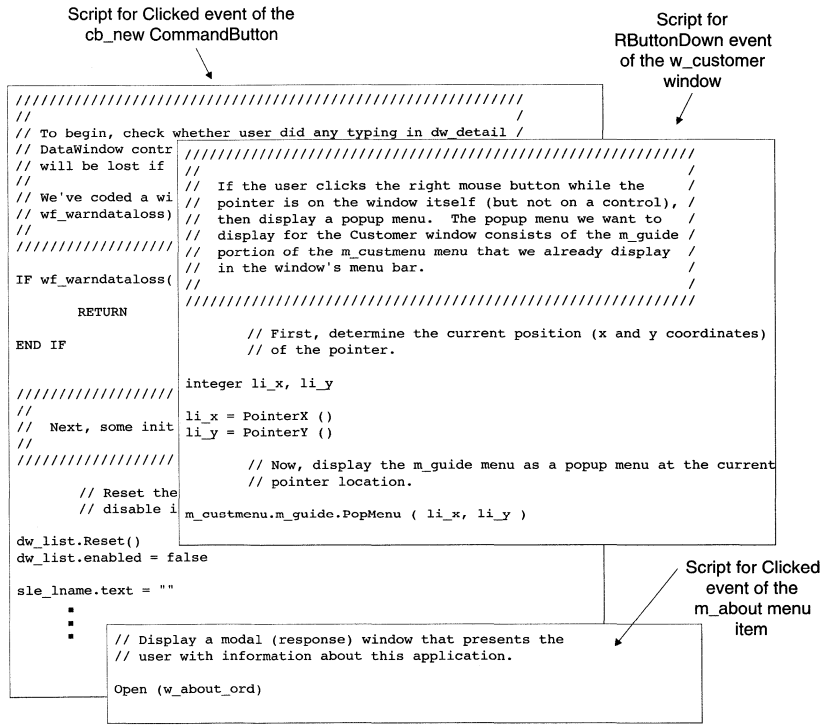
You can also define your own events for any particular component and then manually trigger or post them to execute their scripts. These are called **user events**, and they can be useful for such things as:

- Extending the processing of other event scripts by serving as subroutines
- Responding to certain lower-level messages (from your operating environment) that PowerBuilder doesn't provide as standard events

You'll learn more about triggering, posting, and user events later in this manual.

Specifying the processing logic to perform

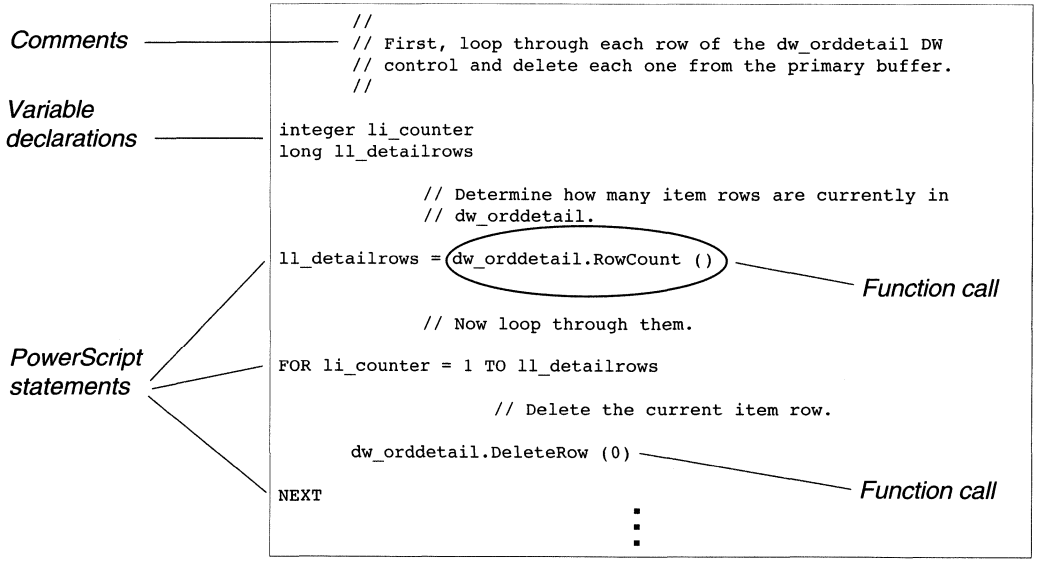
Once you know which events you need to deal with in your application, you've got to plan an appropriate script for each one. A script is a body of procedural code that you write in the **PowerScript language** to express the processing logic to perform. Most scripts are relatively short (tens of lines long, not hundreds), since they just need to express the processing for particular events and not for the whole application.

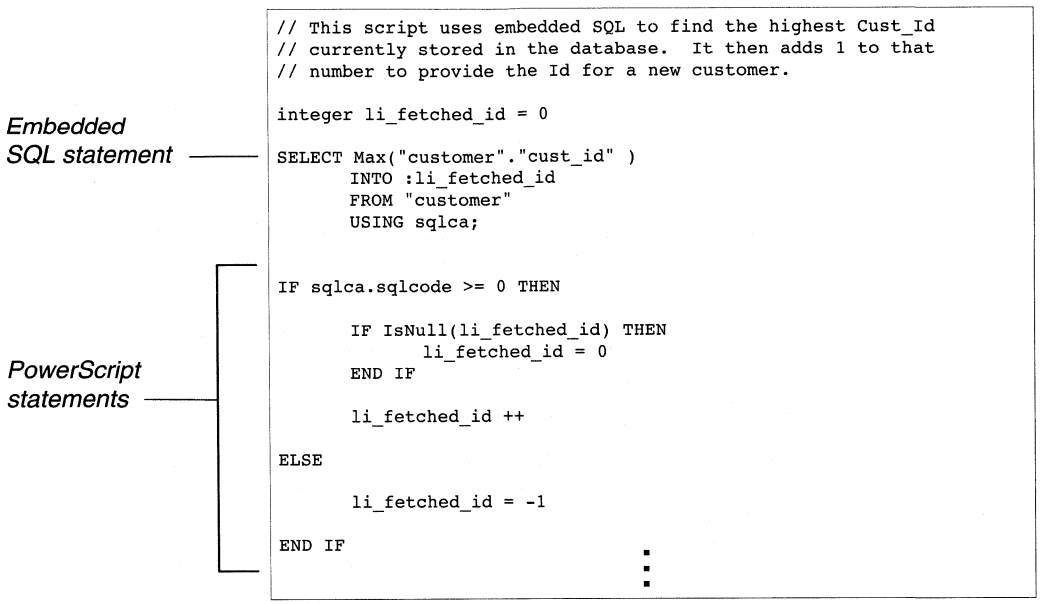


Ingredients of a script PowerScript is a high-level language that provides several different syntactic ingredients you can use to write the code you need. These ingredients include:

- **Variable declarations** PowerBuilder supports many data types, as well as arrays and structures of variables. It also offers several levels of scoping that you can choose from for each variable you declare.
- **PowerScript statements** These statements mostly provide flow-of-control mechanisms (such as branching and looping) that you can use to steer the processing in a particular script.
- **Function calls** PowerBuilder supplies a multitude of built-in functions you can call. You'll use these built-in functions to handle much of the processing in your scripts.

  PowerBuilder also lets you create your own functions and then call them in your scripts. More about these in just a moment.
- **Embedded SQL statements** If you need to perform some table processing and you've decided to use SQL (instead of DataWindows) to do it, you can embed the appropriate SQL statements in a script. PowerBuilder lets you embed standard as well as dynamic SQL statements, and it supports DBMS-specific clauses and reserved words.
- **Comments** PowerBuilder makes it easy to insert comments in your code. You can use comments to document how a script works or to temporarily block out some code that you don't want to execute.





For more information about scripts and the syntax you can code in them, see the *User's Guide* and *PowerScript Language*.

Creating functions to do some of your processing You don't have to put all of the code you write in event scripts. In some cases you'll find that it's better to separate out certain chunks of code to be independent of any particular event.

For these cases, PowerBuilder gives you the ability to create your own functions, known as **user-defined functions**. When you create a user-defined function, you'll specify:

- The arguments it requires when you execute it (if any)
- A script of the code it is to execute
- The value it is to return (if any)

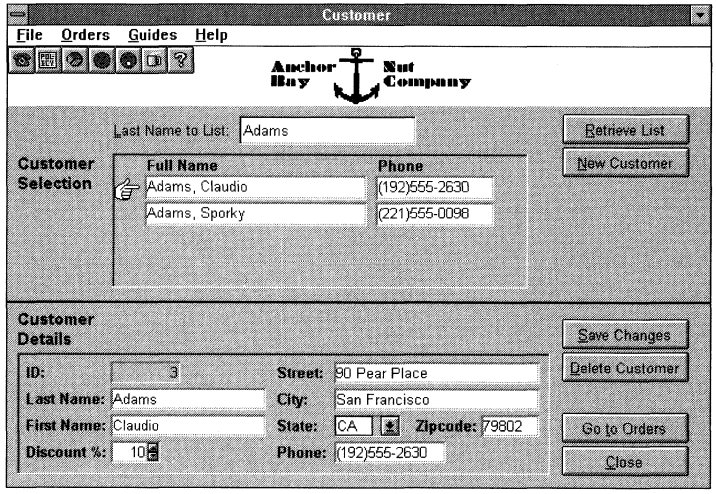
You can then call that user-defined function from event scripts or from other user-defined functions. This enables you to: centralize processing logic that's required in multiple situations, organize your application's code so that it's easier to maintain and extend, or simply break an especially long body of code into more manageable subroutines.

For more information about creating user functions, see the *User's Guide*.

Sample application

Here's how the system designers at the Anchor Bay Nut Company mapped their processing requirements to PowerBuilder features.

Events to handle The Order Entry application needs to handle a number of different events for its various windows, controls, and menus. For instance, consider the application's Customer window:



To provide all of the processing for this window, scripts are required for each of the following events:

Type of component	Name	Events that need scripts
Window	w_customer	Open, RButtonDown
CommandButton control	cb_close	Clicked
	cb_delete	Clicked
	cb_new	Clicked
	cb_orders	Clicked
	cb_retrieve	Clicked
	cb_save	Clicked

Type of component	Name	Events that need scripts
DataWindow control	dw_detail	Clicked, RowFocusChanged, Uevent_keypressed *(a user event)*
	dw_list	EditChanged, ItemChanged, ItemError
SingleLineEdit control	sle_lname	Modified
Menu item listed under File	m_new	Clicked
	m_retrieve	Clicked
	m_save	Clicked
	m_delete	Clicked
	m_printersetup	Clicked
	m_reviewdata	Clicked
	m_exit	Clicked
Menu item listed under Orders	m_goto	Clicked
Menu item listed under Guides	m_phoneprocedures	Clicked
	m_companypolicies	Clicked
	m_salescompetition inquiry	Clicked
	m_employee motivationa	Clicked
	m_employee motivationb	Clicked
	m_employeefun	Clicked
Menu item listed under Help	m_contents	Clicked
	m_about	Clicked

User-defined functions to create The Order Entry application also requires a few user-defined functions to provide additional processing services that can be called from the event scripts. For example, the application's Customer window needs these three user-defined functions:

User-defined function	Purpose
wf_delcustorders	To be called whenever the user asks to delete a customer to make sure the customer doesn't have any outstanding orders in the database
wf_newcustnum	To be called whenever the user asks to add a customer to compute the next available ID number for a new customer in the database
wf_warndataloss	To be called at the beginning of various operations to check whether any unsaved data might be lost and to warn the user

Take a closer look at the wf_delcustorders user-defined function. Here's the script that must be coded for it:

Script for user-defined function wf_delcustorders

```
// This function uses embedded SQL to see if the customer to
// be deleted has any orders in the database.  It takes one
// argument -- the customer's ID number.
//
//     Return values:
//            * If no orders, then 0
//            * If orders, then that number
//            * If error, then -1


integer li_fetched_count = 0

SELECT Count("order_header"."order_id")
       INTO :li_fetched_count
       FROM "order_header"
       WHERE "order_cust_id" = :ai_custid
       USING sqlca;

IF sqlca.sqlcode = -1 THEN li_fetched_count = -1

RETURN li_fetched_count
```

And here's how that user-defined function is to be called from the Clicked event script of the cb_delete CommandButton control (which is labeled *Delete Customer* in the window):

Calling the user-defined function wf_delcustorders from within another script

```
                                    ⋮

// Call the user-defined window-level function wf_delcustorders ( )
// to check whether the customer to be deleted has any orders
// in the database.  If so, disallow the customer delete operation.


integer li_returned_orders
li_returned_orders = wf_delcustorders (li_delid)

CHOOSE CASE li_returned_orders

      CASE IS > 0
            MessageBox("Delete Not Allowed", &
                  "You can't delete a customer who has orders on file. " +&
                  ls_delname + " has " + String(li_returned_orders) +&
                  " orders on file.", stopsign!)
            RETURN

      CASE IS < 0
            MessageBox("Delete Failure", &
                 "Could not determine whether " + ls_delname +&
                  " has any orders on file. " +&
                  "Please call the Anchor Bay support team.", &
                  exclamation!)
            RETURN

END CHOOSE
                                    ⋮
```

Extending the application's processing: external programs

As you look over your application's processing demands, you may very well discover that some of them require your application to interact with external programs or applications. That's pretty common in the world of graphical client/server programming, where you can often get information and services from other sources instead of having to provide them yourself.

The need to interact with an external program typically involves either:

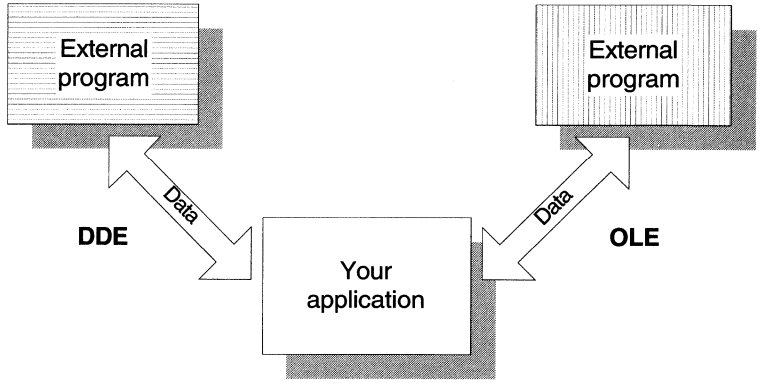
- **Sharing data** between your application and that program, or
- **Getting some service** from that program by executing it from your application

What features does PowerBuilder provide to enable your application to work with external programs in these ways?

Sharing data with other programs

PowerBuilder supports a couple of industry-standard approaches for sharing data between programs:

- Dynamic Data Exchange (DDE)
- Object Linking and Embedding (OLE)



If the plans for your application call for sharing data with an external program, you can choose whichever of these approaches best suits the situation.

Using DDE This approach enables your application to converse with the external program by exchanging commands and data. Your application can act either as a DDE client (which asks the other program to perform some commands or provide some data) or as a DDE server (which responds to command or data requests from the other program).

Building DDE capabilities into your application is a matter of writing some appropriate scripts. To do that, you'll use a few special DDE events and functions that PowerBuilder provides. But before you choose DDE as your solution, make sure that the external program supports it too.

For more information, see Chapter 14, "Using DDE in an Application."

Using OLE This approach is the successor to DDE as a standard for data sharing. As a result, it provides a much more robust set of capabilities for you to use in your application.

For instance, you can embed data (objects) from other programs inside the windows of your application. That way, you can enable a user to point to one of those objects and automatically invoke the capabilities of its corresponding program to edit it. Or you can enable your application to talk to another program under the covers (through a feature called OLE automation) to invoke commands or move data back and forth.

Depending on which features of OLE you need, building these capabilities into your application can involve adding a special **OLE control** to a window, inserting an **OLE column** in a DataWindow, and/or writing some appropriate scripts (using OLE events and functions that PowerBuilder provides). But before you choose OLE as your solution, make sure that the external program supports it too (note that some OLE features in PowerBuilder only require OLE 1.0 support while others require OLE 2.0 support).

For more information, see Chapter 15, "Using OLE in an Application."

Getting services by executing programs

Besides data sharing, there are many other reasons why you might want to employ the services of another program from within your application. To give you as much flexibility as possible, PowerBuilder provides several ways in which your application can execute different kinds of programs. These include:

- Running another application
- Accessing an electronic mail system
- Executing a DLL function
- Defining and using a C++ class
- Executing a database stored procedure
- Defining and using an AppleScript script

Running another application In some situations you might want your application to start up another executable application. This might be another PowerBuilder application you've built or a commercial (off-the-shelf) application (such as a word processor or spreadsheet tool).

For these situations, PowerBuilder provides a Run function you can code in any script to run that executable application.

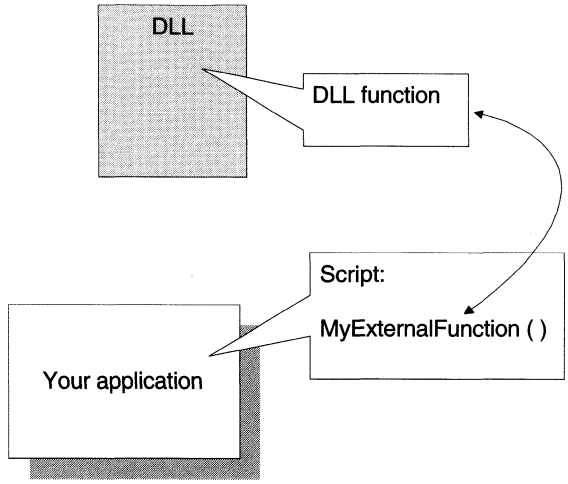
To learn more about the Run function, see the *Function Reference*.

Accessing an electronic mail system One of the requirements of your application may be that you make it mail-enabled. Basically, this involves giving your application the ability to send e-mail messages to people and/or receive e-mail messages from them through an electronic mail system.

PowerBuilder supports access to any e-mail system that uses the mail standard called MAPI (messaging application program interface). To build this access into your application, you'll write some appropriate scripts using a few special mail functions that PowerBuilder provides.

For more information on using MAPI, see Chapter 16, "Building a Mail-Enabled Application." If instead you want to use the mail standard called VIM (vendor independent messaging), see "Adding special-purpose libraries" in Chapter 3, "Setting Up an Application."

Executing a DLL function Sometimes you may want to implement a particular service that your application requires by executing a DLL function—that is, a function located outside of your application in a dynamic link library and written in a language other than PowerScript. PowerBuilder lets you do this by defining an **external function** that provides access to the DLL function. Then you can execute it simply by calling that external function in a script the same way you call any other PowerBuilder function.

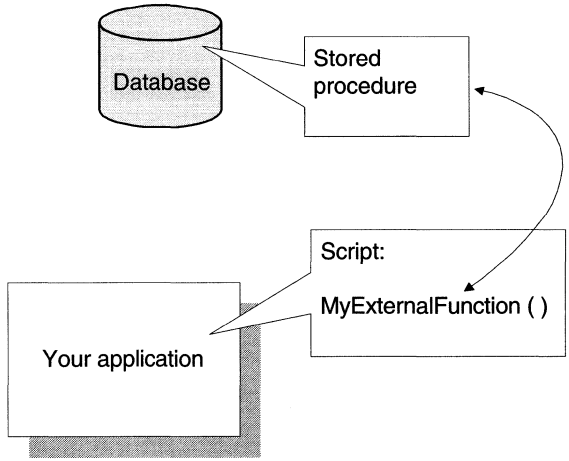


For more information, see Chapter 17, "Adding Other Processing Extensions."

Defining and using a C++ class To give you extra power for implementing application features in which execution speed is particularly critical or lower-level programming is required, Powersoft provides a facility called the **C++ Class Builder**. If you have this facility and you know how to code in C++, you can work with the User Object painter in PowerBuilder to define C++ classes, compile them, and store them in DLLs. Then you can execute the methods of those C++ classes from your PowerBuilder applications as user object functions.

For more information, see the *C++ Class Builder* manual.

Executing a database stored procedure If you're using a database that provides stored procedures, you may want to execute one or more of them to implement some of the services that your application requires. For stored procedures that don't return a result set, you can do this by defining external functions in PowerBuilder that refer to them (similar to what you do to access DLL functions, but with a few more housekeeping steps you'll learn about later). Then you can execute those stored procedures by calling the corresponding external functions in your application's scripts.



The process of executing a stored procedure this way is known as a **remote procedure call** (RPC).

For more information, see Chapter 9, "Using Transaction Objects."

Defining and using an AppleScript script If your application is going to be deployed on the Macintosh, you may want it to execute one or more AppleScript scripts to perform some of the processing it requires. For these situations, PowerBuilder provides:

- The ability to edit and test AppleScript scripts in the PowerBuilder file editor as you develop your application
- The DoScript function, which you can code in any of the application's scripts to execute the desired AppleScript scripts at execution time

To learn more about working with AppleScript scripts in the file editor, see the *User's Guide*. To learn more about the DoScript function, see the *Function Reference*.

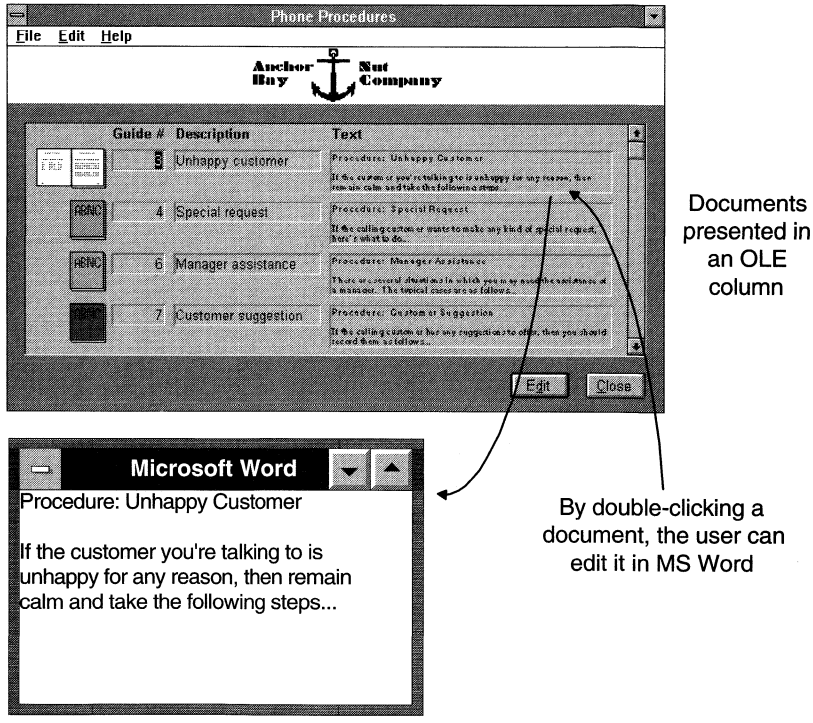
Sample application

Here's how the system designers at the Anchor Bay Nut Company mapped their program interaction requirements to PowerBuilder features.

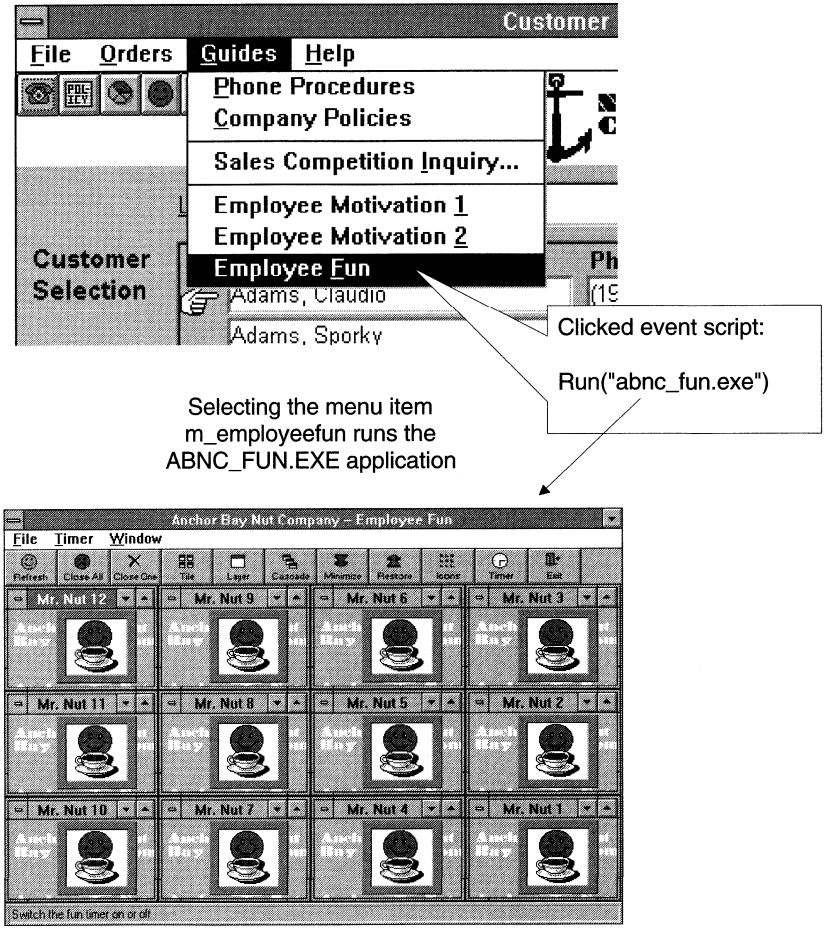
Doing some data sharing To enable a user to read about phone procedures, the Order Entry application needs to display various Microsoft Word documents in a window and let the user edit any one of them in Microsoft Word just by double-clicking that document. This is to be accomplished by:

1 Creating a DataWindow that includes an OLE column to display the Word documents

2 Using this DataWindow in the appropriate window (w_guides)

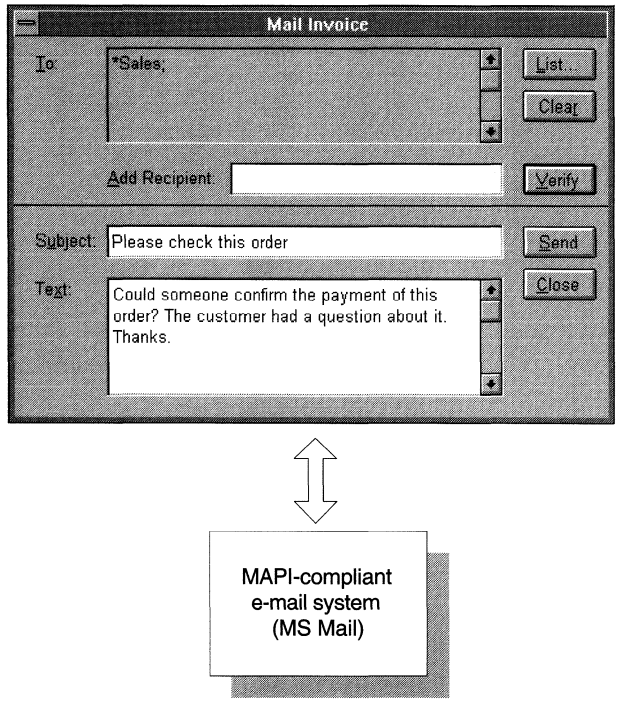


Running another of their applications While using the Order Entry application, users need to be able to run a small motivational application (ABNC_FUN.EXE) that was previously constructed with PowerBuilder. This is to be accomplished by providing a menu item whose Clicked event script calls the Run function to start that application.



Accessing their e-mail system Sometimes users of the Order Entry application will need to e-mail order invoice information to other people in the company. Since the company uses a MAPI-compliant e-mail system (Microsoft Mail), this can be done with the mail functions that PowerBuilder provides.

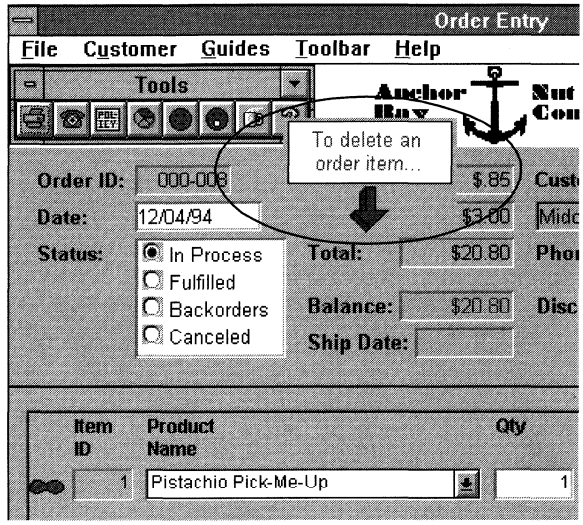
The application should display the following window (w_mail) to let users compose and send their mail messages. Event scripts of this window and its controls should be coded to call the appropriate mail functions.



The scripts in this window work with the e-mail system to let users send messages

Using an external function The Order Entry window of the application is designed to display an animated helper that users can select from the File menu to get assistance deleting an order item. This helper needs to play sound files to talk the user through the deletion process.

Playing sound files requires a DLL function (in this particular operating environment, the sndPlaySound function in MMSYSTEM.DLL). To execute that function from inside the application, the helper window (w_itemdelete_helper) must define and then call an external function.



This window uses a DLL function by first defining it as an external function

External function declaration:

FUNCTION integer sndPlaySound...

and then calling that function from one of its event scripts

Timer event script:

sndPlaySound("todelete.wav",3)

MMSYSTEM.DLL

DLL function: sndPlaySound

Routing application output: reports, files, and pipelines

As you've already learned, a lot of the output requirements of an application involve displaying windows to users and applying user updates to database tables. But many applications need to produce one or more additional kinds of output as well. These may include:

- Printed reports
- Nondatabase files
- New or copied database tables

What features does PowerBuilder provide to fulfill these output requirements?

Printing reports

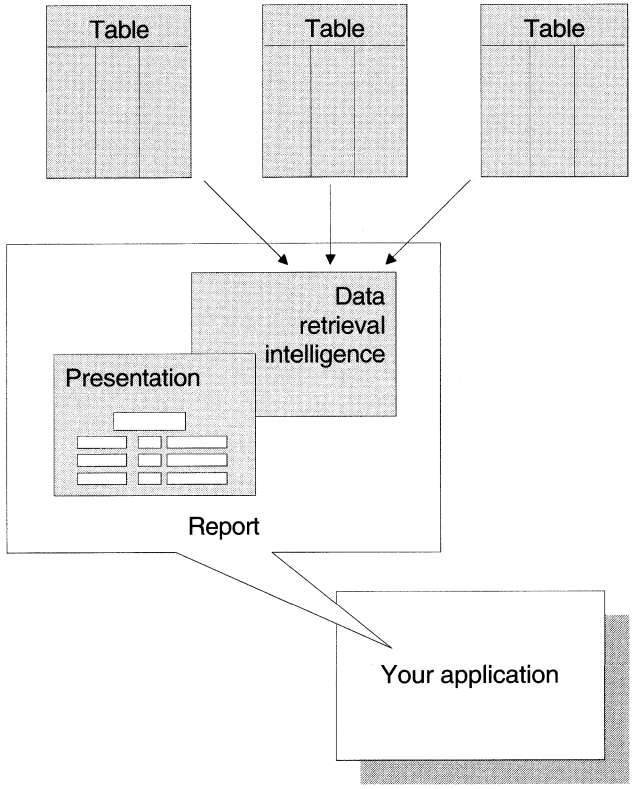
Users often want an application to generate one or more paper reports about its data that they can distribute, mark up, or simply read later away from the computer. For instance, maybe they want to print an expense report, a parts list, a form letter, or a bill.

If your application needs to generate any reports, you've got to figure out what data to include in each one and how to format that data given the report's purpose.

Designing reports in PowerBuilder The reports you design in PowerBuilder are actually specialized DataWindows. Specifically, reports are just like the other DataWindows you read about earlier, except that they only retrieve data and don't perform any update operations.

The advantage of this approach is that you can create reports that contain both:

- The **intelligence** to access the appropriate database tables and retrieve the rows you want, *and*
- The **presentation abilities** to format this row data as you require



Kinds of reports PowerBuilder supports a wide variety of report formats and also gives you the ability to customize just about any aspect of a particular report. Here are a few simple examples that show some of the kinds of reports you can design.

Anchor Bay Nut Company

Customer List 7/13/94

Last Name	First Name	Street	City	State	Zipcode	Phone
Adams	Claudio	90 Pear Place	San Francisco	CA	79802	(192)555-2630
Adams	Sporky	1 Apple Alley	Battle Creek	MI	40098	(221)555-0098
Bush	Bosworth	2 Raspberry Row	Boston	MA	02108	(617)555-5631
Carter	Kiki	43 Orange Street	Kansas City	MO	26800	(306)555-0077
Ford	Fred	900 Mango Road	Toledo	OH	30290	(488)555-0600
Grant	Gomer	8900 Papaya Lane	Honolulu	HI	91100	(809)555-3900
Harding	Stoiko	7 Apricot Avenue	Baton Rouge	LA	32522	(670)555-8100
Jackson	Svetlana	501 Lemon Lane	Salt Lake City	UT	60330	(390)555-1233
Jefferson	May-Lin	19 Plum Drive	Seattle	WA	72891	(373)555-5878
Johnson	Jean-Claude	505 Prune Highway	Joliette	IL	30022	(400)555-8004
Lincoln	Lana	2 Grape Way	Buzzard Gulch	MT	64689	(223)555-1110
Lincoln	Suzie	93 Strawberry Street	Sebastian	FL	20988	(344)555-8390
Monroe	Midori	30 Kiwi Terrace	Phoenix	AZ	83111	(500)555-3002
Roosevelt	Pablo	4 Peach Street	Atlanta	GA	53994	(209)555-2645

A report with a tabular layout

Product Report 7/13/94

Product Category	Product Name	ID	Price	Weight
Assortment				
	Filberts and Friends	17	$19.95	5.1
	Nuts of the World	14	$25.95	6.8
	Nuts-To-You Holiday Assortment	9	$29.95	7.4
	Nutty Neighbor Welcome Basket	16	$23.95	6.2
		Averages:	**24.95**	**6.3**
Children's				
	Cashew Caboose Kid's Pack	6	$13.95	3.5
	Dinosaur Adventure Mix	19	$9.95	2.4
		Averages:	**11.95**	**2.9**
General				
	Almonds Ahoy!	3	$15.95	4.4
	Brazilnut Bonanza	7	$17.95	5.3
	Carnival of Cashews	12	$21.95	4.7
	Filbert Fiesta	10	$18.95	5.5

Page 1 of 2

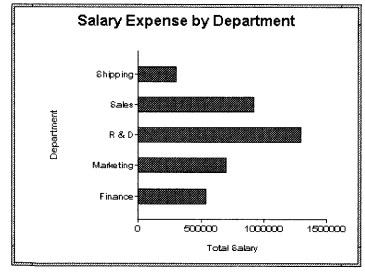
A report with grouped data

Last Name	First Name	Team	Quarter	Quota	Actual
Bean	Edna	D	Q1	$64,000	$62,299
			Q3	$64,000	$64,330
			Q2	$64,000	$66,870
			Q4	$64,000	$65,023
Broccoli	Antonio	C	Q4	$30,000	$31,755
			Q3	$30,000	$25,700
			Q2	$30,000	$31,290
			Q1	$0	$0
Corn	Carlos	A	Q1	$50,000	$53,780
			Q2	$50,000	$55,892
		B	Q4	$55,000	$56,821
			Q3	$55,000	$56,100

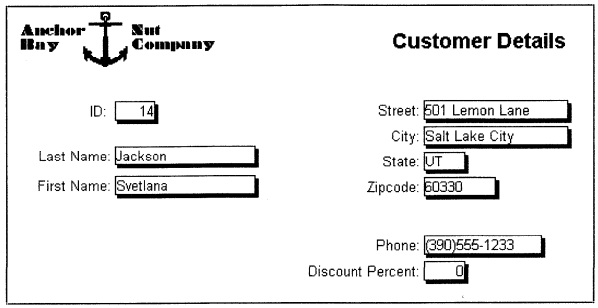
A report in a grid

Breakdown of Salary Expenses for 1995

Department	Total Salary
Finance	$535,500.00
Marketing	$698,250.75
R & D	$1,292,198.19
Sales	$919,428.00
Shipping	$303,769.80



A report with a graph



A report with a freeform layout

Mailing labels

In addition to these formats, you can design crosstab reports, reports with newspaper-style columns, reports nested within other reports, and more.

What you can do with reports Once you've designed a report in PowerBuilder, you're ready to use it in an application. This involves:

1 Displaying the report in a window through a DataWindow control (just as you do for any other DataWindow)

2 Letting users do one or more of the following:

- **Examine** the formatted report output in print preview mode
- **Send** the formatted report output to a printer
- **Save** the formatted report output as a Powersoft report (PSR) file (which can later be printed or displayed)

PowerBuilder provides built-in functions that you can code in scripts to implement these services.

For more information on designing and using reports, see Chapter 10, "Using DataWindow Objects."

Producing files

Although most of the data output from a typical application is directed at databases, you may also need to write data to other kinds of files. For example, users might want the application to save certain data in spreadsheet files that they can use later in another program.

PowerBuilder provides built-in functions that you can code in scripts to write (and in some cases read) many different kinds of files from your application, including:

- Text files
- Microsoft Excel spreadsheets
- Lotus 1-2-3 spreadsheets
- dBASE files
- The clipboard (of your operating environment)

as well as various others.

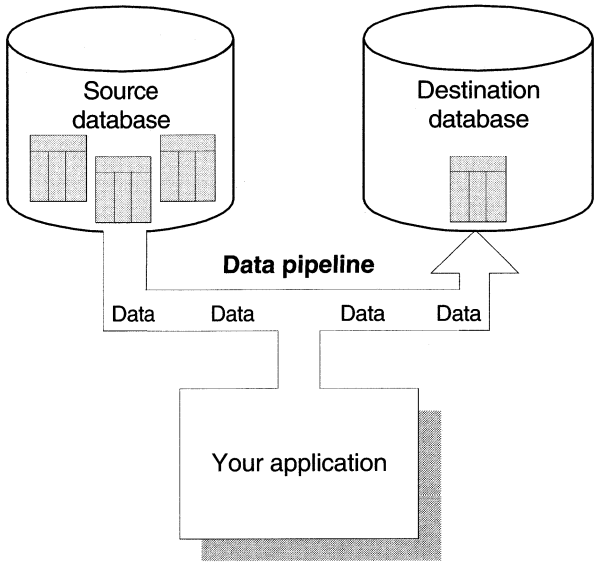
To learn more about functions for producing files, see the *Function Reference*.

Piping data between tables

Sometimes applications require the ability to migrate data from one place to another in the course of their work. For example, you might need to:

- **Create a new sales summary table** in your database by joining various regional sales tables and extracting particular data from them, or
- **Copy a payroll table** from the server database to a local database so that the application can access it without needing the network

To handle requirements such as these, PowerBuilder provides **data pipelines**. A data pipeline is an application component you can design to pump data from one or more source tables to a new or existing destination table. The source and destination tables you specify can either be in the same database or in separate databases (even if they're different kinds of databases that involve different DBMSs).



Data pipelines are a lot like DataWindows when it comes to their built-in data access and manipulation intelligence. And the steps you need to take to make them work in your application are similar in several ways too.

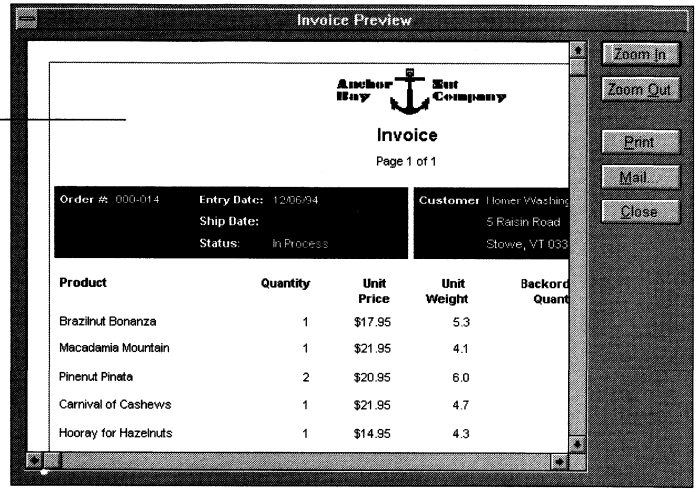
For more information on using data pipelines in your application, see Chapter 12, "Piping Data Between Data Sources."

Sample application

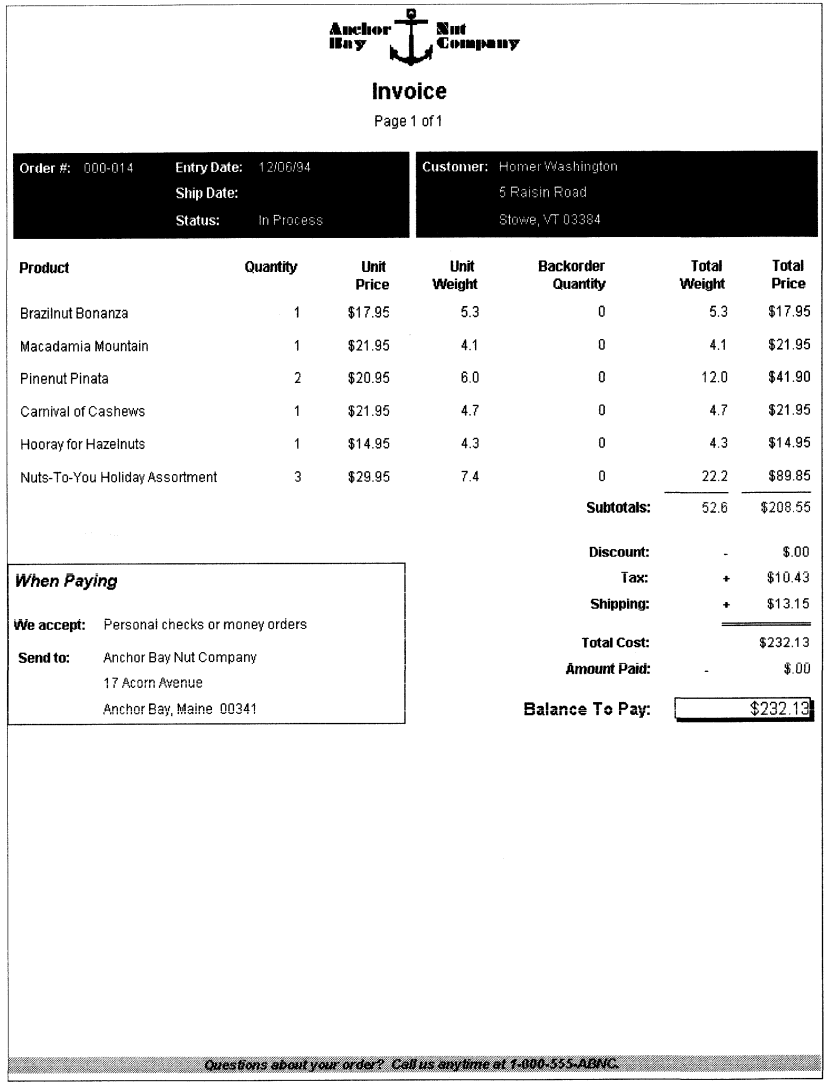
Here's how the system designers at the Anchor Bay Nut Company mapped their additional output requirements to PowerBuilder features.

Printing order invoices The Order Entry application needs to provide users with a way to preview and then print invoices of particular orders so that customers can be billed. To accomplish this, the application should use the following window (w_invoice), which displays the invoice report in a DataWindow control:

Previewing a report in a DataWindow control



When a user clicks the Print command button, the corresponding event script should send this invoice to the printer (by calling the built-in Print function). The following sample shows what the printed report will look like.



Anchor Bay Nut Company

Invoice

Page 1 of 1

Order #: 000-014 **Entry Date:** 12/06/94
Ship Date:
Status: In Process

Customer: Homer Washington
5 Raisin Road
Stowe, VT 03384

Product	Quantity	Unit Price	Unit Weight	Backorder Quantity	Total Weight	Total Price
Brazilnut Bonanza	1	$17.95	5.3	0	5.3	$17.95
Macadamia Mountain	1	$21.95	4.1	0	4.1	$21.95
Pinenut Pinata	2	$20.95	6.0	0	12.0	$41.90
Carnival of Cashews	1	$21.95	4.7	0	4.7	$21.95
Hooray for Hazelnuts	1	$14.95	4.3	0	4.3	$14.95
Nuts-To-You Holiday Assortment	3	$29.95	7.4	0	22.2	$89.85
				Subtotals:	52.6	$208.55
				Discount:	-	$.00
				Tax:	+	$10.43
				Shipping:	+	$13.15
				Total Cost:		$232.13
				Amount Paid:	-	$.00
				Balance To Pay:		$232.13

When Paying

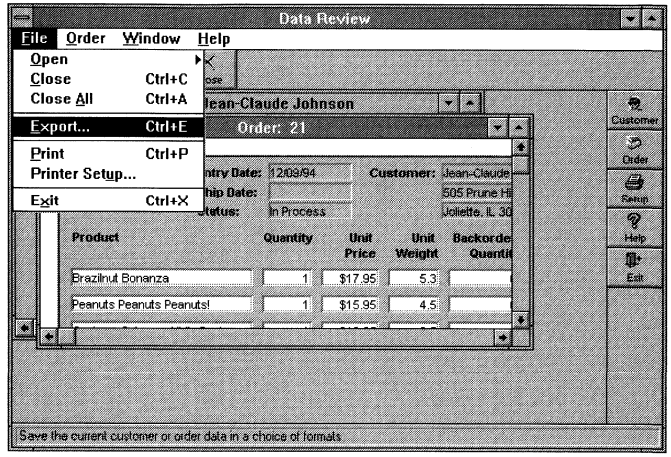
We accept: Personal checks or money orders

Send to: Anchor Bay Nut Company
17 Acorn Avenue
Anchor Bay, Maine 00341

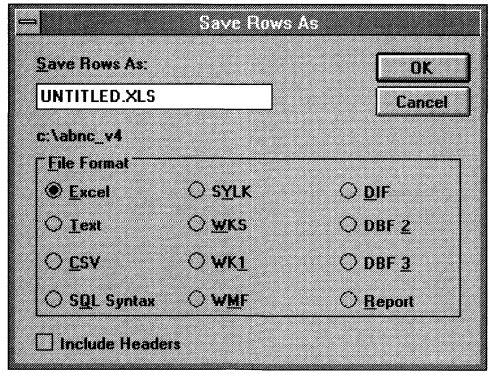
Questions about your order? Call us anytime at 1-000-555-ABNC.

Letting their users export files When users are working in the Data Review portion of the Order Entry application, they have to be able to export data about a particular customer or order to a file of their choosing. Additionally, they must be able to specify a file format that suits their needs at that moment.

To support these requirements, the Data Review window should offer an Export menu item that prompts for the filename and format to save the current data to:

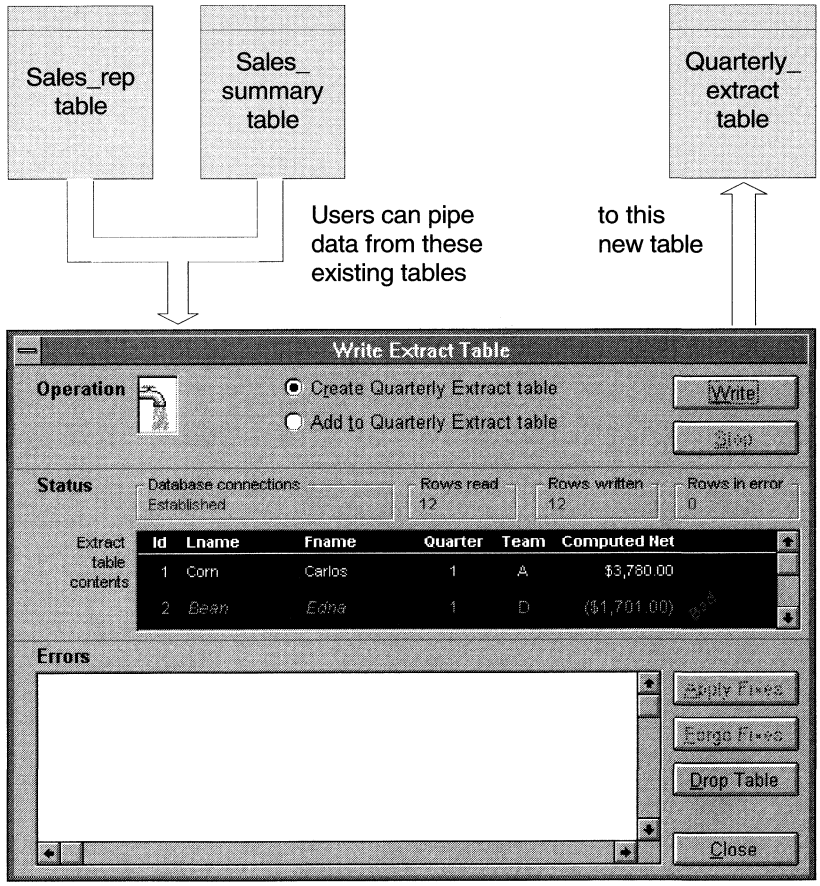


Fortunately, one of the **common dialog boxes** that PowerBuilder provides (namely, the Save Rows As dialog box) is appropriate for handling this job. Displaying this dialog box is just a matter of coding a built-in function (the SaveAs function) in the Export menu item's Clicked event script. Here's what the user will see when that script executes:



Piping sales data to an extract table As users work in the Sales Competition Inquiry portion of the Order Entry application, they will sometimes want to extract quarterly data from the Sales_rep and Sales_summary tables of the company's sales database (ABNCSALE.DB) and store that data in a new table (which they can use later for querying and reporting).

To provide this capability, the application should display a window (w_sales_extract) that lets users execute an appropriate data pipeline:



Where to go from here

Once you've completed your design work and have a blueprint for the application, the next phase of the project involves taking care of a few important setup chores before you start developing the application's detailed features. These setup chores require some up-front time from you, but they provide a big payoff later by making your application better and easier to maintain.

To learn about the setup phase, turn to Chapter 3, "Setting Up an Application."

CHAPTER 3

Setting Up an Application

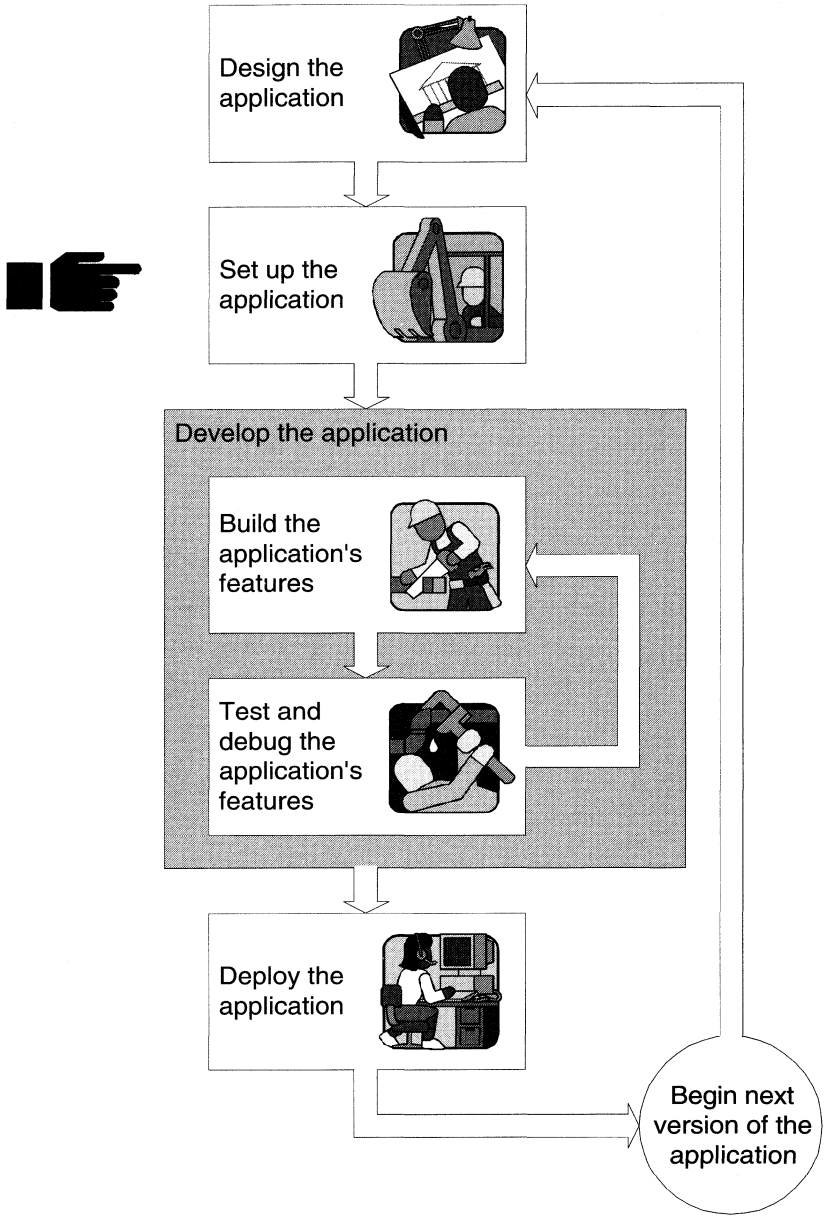
About this chapter

This chapter tells you about the process of setting up an application that you've designed.

Contents

Orientation

Here's where you are now in the lifecycle of your application development project:


Design the application
Set up the application
Develop the application
Build the application's features
Test and debug the application's features
Deploy the application
Begin next version of the application

Choosing how to set up

You can approach the setup phase of your project in a couple of different ways, depending on your goals for the application.

The comprehensive way

If the aim of your project is to deliver a full-scale, *production-quality* application, then you'll normally want to perform the comprehensive series of setup steps outlined in this chapter. This approach involves some short-term costs, because you'll be investing time and effort on such things as:

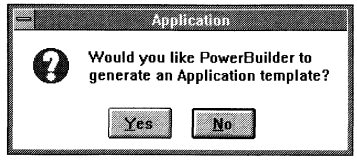
- Standards and conventions for your application
- Organization of the various application components you'll be building
- Reusability of those components
- Source control

But your reward for this up-front work will be a neat, well-crafted application—one that can be maintained and extended later with as little difficulty as possible.

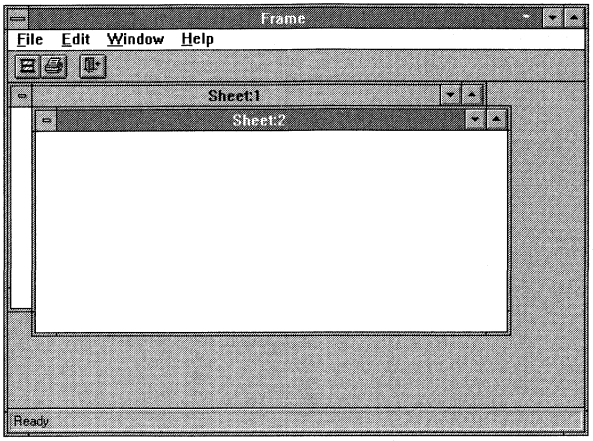
The quick way

On the other hand, suppose you just want to put together a small-scale personal application, or an application that's just for practice or testing. To handle these cases, PowerBuilder provides the **quick application** feature—an express route for setting up that gets you going with a minimum of time and effort.

How it works Whenever you ask to create a new application that you want to start building, PowerBuilder gives you the option of automatically generating an **application template**:



If you say yes, PowerBuilder creates the basic components of a skeletal MDI-style application for you, including several windows and their menus along with various event scripts. You can run this application to see how it works:



Then you can begin enhancing it to include the particular features you need.

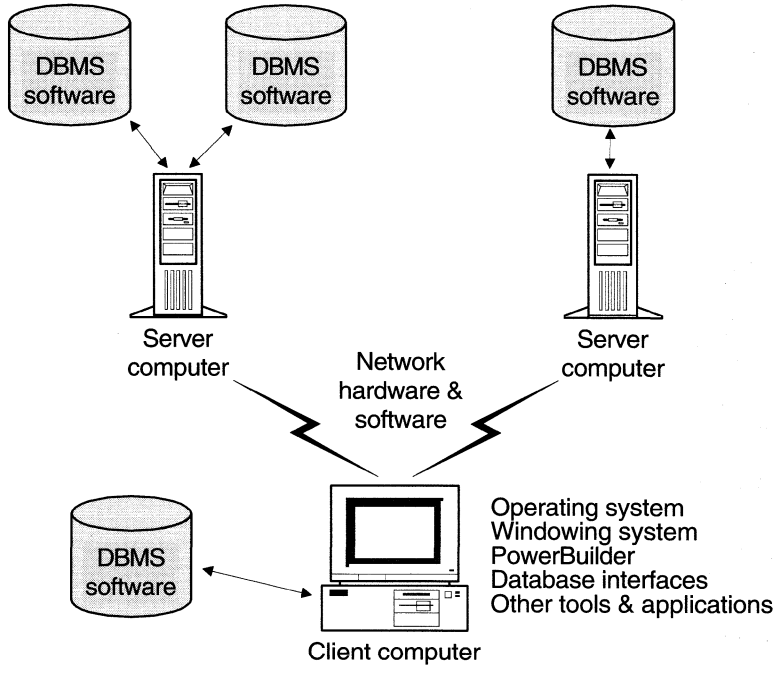
What it does for you The advantage of this quick application approach is that it instantly gives your project a head start so that you can jump into the actual development phase sooner. Just keep in mind that, depending on how you then want to tailor the application, you may still need to take care of a few setup chores yourself (such as configuring your hardware and software environment or creating any databases to be accessed).

For more information on using the quick application feature of PowerBuilder, see the *User's Guide*.

Configuring your environment

No matter how you plan to set up your application, you need to begin by configuring the hardware/software environment in which it is to be developed. This environment typically includes:

- Server computers
- DBMS and other server software
- Network hardware and software
- Client computers for developers
- Software for those client computers, including: the operating system, windowing system, PowerBuilder, database interfaces, and other tools or applications



Making sure it all works

It's important to get these elements installed and working properly as early as possible in the lifecycle of your project. If you wait too long, you may complicate the development and debugging of the application needlessly.

For instance, tracking down a performance problem in a particular part of your application can be difficult if the environment in which you're working is unstable. You may have to spend considerable time trying to determine whether the problem lies in the application itself or somewhere outside of it—in the operating system, or the network, or the server.

Configuring user computers

Later on in the project as you prepare to deploy your completed application, you'll also have to make sure that the client computers of the application's end users are properly configured and connected.

Establishing standards and conventions

If the goal of your project is to build a production-quality application, you're bound to have a variety of standards and conventions that should be followed once work begins. As you start setting up the application, it's a good idea to make a list of them for reference during the project.

Kinds of standards and conventions

The kinds of standards and conventions you'll be dealing with include:

- User-interface conventions
- Naming conventions
- Programming conventions
- Documentation standards
- Standard error and status routines

Who is responsible for them

If you're working on a project team, it's common to have one or more team members who are responsible for maintaining and/or enforcing particular sets of standards and conventions. For example, a DBA might handle the ones related to databases while someone in the role of object manager might oversee most of the others.

If you're working solo, there are still plenty of reasons for coming up with standards and conventions for yourself to follow. They can help keep your work consistent and orderly, especially on projects that involve lots of components and code or that extend over long periods of time. And they can make it easier for you to maintain or enhance an application later on.

User-interface conventions

PowerBuilder provides a wide variety of components and options that you can use to construct the user interface of your application. While this gives you a great deal of flexibility, it also requires you to make choices about what is appropriate for your project. By making these choices early on and establishing them as conventions, you can make the application more consistent, attractive, and usable while also saving yourself from a lot of cleanup work later.

Starting to gather UI conventions

To gather intelligent user-interface conventions, you must either learn something about good graphical design or draw upon the expertise of someone who does. Here are some suggestions for doing both.

Finding good examples You can find lots of good ideas for your user-interface conventions by looking at existing graphical applications, especially some of the more popular commercial products (for instance, Microsoft Word or Excel, Lotus 1-2-3, or PowerBuilder itself). You probably already use several of these applications and know which aspects of their user interfaces you like and which you don't.

If the intended users of your application work frequently with a particular product, you might even consider adopting a similar user interface in order to take advantage of their experience and lower the learning curve.

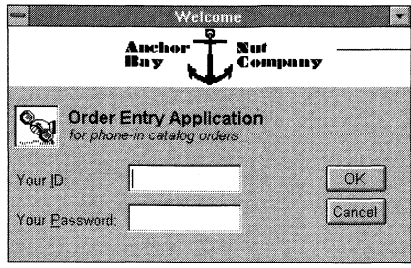
Reading about good graphical design There are plenty of informative books you can read to learn about designing graphical user interfaces. Here are some you might start with:

For this platform	Read
Microsoft Windows	*The Windows Interface: An Application Design Guide* from Microsoft Press
Apple Macintosh	*Human Interface Guidelines: The Apple Desktop Interface* from Addison-Wesley Publishing Company
UNIX Motif	*Motif Style Guide* from Prentice Hall

Adhering to shop standards It's quite possible that you've already got a set of general user-interface conventions that all of your application development projects must adhere to. Such shop standards can be very helpful in:

- **Establishing** consistency across multiple applications
- **Facilitating** the reuse of application components
- **Avoiding** redundant design and development work

Here's a brief example. At the Anchor Bay Nut Company, shop user-interface standards require that all major windows in applications display the company's logo (consisting of a Picture control and a Rectangle control) at the top:



Displaying the company logo as a shop UI standard

When you have shop standards to follow, you'll just need to determine any extensions or exceptions that are required for the user interface of each particular application you develop.

Some common UI conventions

Here are some commonly used conventions that you might consider adopting for the user interfaces you design and build.

User-interface styles Earlier you learned that an application can employ one of three different styles of user interface: SDI (Single Document Interface), MDI (Multiple Document Interface), or a combination. MDI is the most popular choice because of the flexibility it gives the user and all of the built-in services it provides. You should probably think about using it by default.

SDI may be appropriate if your application is very simple, especially if it deals with only one kind of data and the user needs to perform only one operation at a time. A combination interface may be useful in an application that performs very diverse operations, but you must design it with great care to avoid confusing the user.

> **Consider this**
> In the world of graphical applications, many of the more prevalent commercial products (including PowerBuilder) use an MDI interface.

Window conventions Conventions you might apply to the windows in your application include:

Topic	Convention
Kinds of controls	To keep the interface intuitive, use an appropriate kind of control for each feature you want to implement in a window. For instance, don't use a RadioButton to initiate a command—use a CommandButton or PictureButton instead.

Topic	Convention
Number of controls	Limit the number of controls you place in a window to avoid overwhelming the user. Use multiple windows instead of trying to cram too many controls into one.
Spacing of controls	Leave plenty of white space in a window. Don't crowd controls together.
Borders of controls	Display editable or selectable controls with 3D lowered borders (or with box or shadow box borders) to make them stand out. Display other controls (such as StaticText) without any borders.
Availability of controls	Gray out (disable) a control when it is not available to the user.
Length of list controls	Limit the number of items in a ListBox or DropDownListBox control to prevent it from becoming too long to scroll through. You'll usually want to keep the number of items well under 50.
Buttons and MDI sheets	Avoid placing CommandButton or PictureButton controls in MDI sheets. Use menu items instead.
Keyboard support	Provide keyboard access to every control in a window.[M1]
Colors	Use color judiciously. It is most effective for bringing out those portions of a window that you want the user to see most readily. Where possible, use color defaults so that the user's settings (from the windowing system) can take effect.
Fonts	Limit the number of fonts to one or two (and make sure that users will have those fonts installed on their computers). Keep font size large enough to be easily legible and use a very limited number of different font sizes.
Modality	Try to use nonmodal windows whenever possible to give the user maximum control over the interface. Use modal (response) windows only when the application must focus the user's attention.

[M1] Doesn't apply on the Macintosh.

Menu conventions Conventions you might apply to the menus in your application include:

Topic	Convention
Number of menu items	Limit the number of menu items you place in a menu to avoid overwhelming the user with choices.
Depth of menu items	If you use a cascading menu, go down only one level. Deeper menus confuse users.
Availability of menu items	Gray out (disable) a menu item when it is not available to the user.
Length of menu item names	Try to make menu items just one or two words long (and never more than four).
Wording of menu item names	Make sure the name you use is descriptive of what that menu item does. To indicate a menu item that displays a dialog box, include ellipsis points at the end of its name (for example, *Find...*).[M1] Be consistent in your use of either nouns or verbs for menu items.
Standard menu item names	Where possible, use standard names and positions for menu items (for instance, a typical menu bar should begin with *File Edit*).[M1]
Toggles	If a menu item serves as a toggle between one state and another (such as on or off), indicate the current state to the user either by displaying a checkmark (for example, ✓ *On*) or by switching the menu item name (such as from *On* to *Off*).[M1]
Help	Make sure that every menu bar provides menu items that display your application's Help system and information about the application.
Keyboard support	Provide keyboard access to every menu item.[M2]
MDI frame and sheet menus	In an MDI application, provide a menu for the frame (which displays when no sheets are open) and a separate menu for each kind of sheet (which displays when a sheet of that kind is active).
MDI toolbars	In an MDI application, display toolbar buttons for a few of the most common menu items.

Topic	Convention
MDI MicroHelp	In an MDI application, provide MicroHelp text (which displays in the frame's status area) for every menu item.

M1 Underlining of mnemonic characters in menu items doesn't apply on the Macintosh.

M2 Doesn't apply on the Macintosh.

Limitations of user computers As you decide on user-interface conventions, keep in mind the display capabilities of the user computers on which the application will ultimately run. For instance:

- **If users may have monochrome monitors**, don't rely too heavily on color to provide important cues in your application's user interface.
- **If users may have small low-resolution monitors**, make sure you design your windows for them and not just for larger high-resolution monitors, such as those that developers might have.

Platform specifics The choice of some user-interface conventions will depend on the particular platform on which your application is to be deployed. For example, certain conventions that may be appropriate for Microsoft Windows users might not make sense for Apple Macintosh users. Make sure you are familiar enough with your target platform to choose conventions that suit it.

If you're planning to deploy an application on multiple platforms, your user-interface conventions should take any platform differences into account and specify how to handle them.

Naming conventions

While using PowerBuilder to develop your application, you'll create lots of different components and have to specify names for them too. These components include: objects such as windows and menus, controls that go into your windows, and variables for your event and function scripts.

To keep those names straight, you should devise a set of naming conventions and follow them faithfully throughout your project. This is critical when you're working on a team (to enforce consistency and enable others to understand your code), but it's even important if you work on your own (so that you can easily read your own code).

In general

All component names in PowerBuilder can be up to 40 characters long. Developers typically use the first few characters to specify a prefix that identifies the kind of component it is. Then they type an _ (underscore) character, followed by a string of characters that uniquely describes this particular component.

For objects

Whenever you save a new object in PowerBuilder, you'll need to provide a name for that object. PowerBuilder doesn't force you to use any particular prefixes for different types of objects, but most developers follow these conventions:

Type of object	Prefix	Example
Window	w_	w_customer
Window function	wf_	wf_newcustnum
Window structure	ws_	ws_address
Menu	m_	m_custmenu
Menu function	mf_	mf_print
Menu structure	ms_	ms_address
User object	u_	u_custom_toolbar
User object function	uf_	uf_set_menu
User object structure	us_	us_comminfo
DataWindow	d_	d_custdetail
Pipeline	pipe_	pipe_sales_extract
Function	f_	f_checkrequired
Function structure	fs_	fs_checkdata
Structure	s_	s_selectinfo
Query	q_	q_custorders
Project	proj_	proj_ord_v4
Application	*None*	abnc_ord
Application function	appf_	appf_confirm
Application structure	apps_	apps_loginfo

For controls

Whenever you place a new control in a window, PowerBuilder provides a default name for that control. The default name consists of a recommended prefix for that type of control, an _ (underscore) character, and then a number. For instance:

- **cb_1** (a CommandButton control)
- **sle_4** (a SingleLineEdit control)

- **dw_2** (a DataWindow control)

Most developers keep the prefix portion suggested by PowerBuilder but replace the number with a more meaningful string of characters (to describe the control's purpose in the window):

Type of control	Prefix	Example
CheckBox	cbx_	cbx_overwrite_mode
CommandButton	cb_	cb_close
DataWindow	dw_	dw_list
DropDownListBox	ddlb_	ddlb_flavors
EditMask	em_	em_price
Graph	g_	g_revenues
GroupBox	gb_	gb_styles
HScrollBar	hsb_	hsb_volume
Line	ln_	ln_divider
ListBox	lb_	lb_months
MultiLineEdit	mle_	mle_message_text
OLE 2.0	ole_	ole_spreadsheet
Oval	oval_	oval_spotlight
Picture	p_	p_logo
PictureButton	pb_	pb_newitem
RadioButton	rb_	rb_sumactuals
Rectangle	r_	r_border_area
RoundRectangle	rr_	rr_background
SingleLineEdit	sle_	sle_lname
StaticText	st_	st_tolabel
UserObject	uo_	uo_toolbar
VScrollBar	vsb_	vsb_intensity

Naming DataWindows and user objects
It's especially helpful to follow the recommended naming conventions when it comes to DataWindows and user objects. Make sure you use the prefix *d_* in the names of *DataWindow objects* and the prefix *dw_* in the names of *DataWindow controls*. Make sure you use the prefix *u_* in the names of *user objects* and the prefix *uo_* in the names of *UserObject controls*. That way you'll always be able to distinguish between those objects and their corresponding controls.

For variables

Whenever you need a variable for reference in one or more scripts, you must declare a data type and name for that variable. PowerBuilder supports a wide variety of data types, including standard data types (such as string, decimal, and integer) and system object data types (such as window, menu, transaction, and mailsession).

One common approach for naming a variable is to specify a prefix that indicates the scope and data type, followed by an _ (underscore) character and then a string of characters that uniquely describes the variable's purpose. Under this scheme, you select the first letter of the prefix based on the scope of the variable:

If the scope is	The first letter of the prefix should be
Global	g...
Shared	s...
Instance	i...
Local	l...
Function argument	a...

And then you select the remaining letter(s) of the prefix based on the data type of the variable. For instance, here are some of the more common ones:

Category	If the data type is	The remaining letters of the prefix should be
Standard data types	Blob	...bb_
	Boolean	...b_
	Character	...ch_
	Date	...d_

Category	If the data type is	The remaining letters of the prefix should be
	DateTime	...dt_
	Decimal	...c_
	Double	...db_
	Integer	...i_
	Long	...l_
	Real	...r_
	String	...s_
	Time	...t_
	UnsignedInteger	...ui_
	UnsignedLong	...ul_
System object data types	DataWindow	...dw_
	DataWindowChild	...dwc_
	MailSession	...ms_
	Menu	...m_
	Structure	...str_
	Transaction	...trans_
	User object	...uo_
	Window	...w_

Examples The following examples show how you put this all together to devise appropriate names for your variables:

This variable	Is a
gi_num_lines	Global integer
sstr_data_points[ ]	Shared structure array
iul_pointer	Instance unsigned long
lb_finished	Local boolean
adw_source_rows	Argument DataWindow
gs_line_buffer[ ]	Global string array

This variable	Is a
sdt_last_update	Shared datetime
idb_last_amount	Instance double
lw_launch_win	Local window

Programming conventions

To encourage the use of good and consistent coding techniques over the course of your project(s), you should establish a set of basic programming conventions. Such conventions can greatly enhance the quality of your scripts, making them more readable, debuggable, maintainable, and reusable.

Here are some commonly used programming conventions that you might consider adopting.

Typing script code

These conventions have to do with how you type code in your scripts.

Spacing Type a space before and after any of the following:

- All operators (such as +, *, <, =)
- The Assignment symbol (=)

Separate the parameters that you supply for a function by typing a space between each one. Also, insert a blank line between statements. For example:

```
decimal lc_ordertax

lc_ordertax = lc_discounted_price * .05

dw_ordmaster.SetItem(1, "order_header_order_tax", &
lc_ordertax)
```

Indenting Improve the readability of complex statements by indenting where appropriate (through the use of tabs). This is especially helpful with statements such as these:

- CHOOSE CASE
- DO...LOOP
- FOR...NEXT

- IF...THEN

For example:

```
IF ldwis_itemprod = datamodified! OR &
   ldwis_itemprod = newmodified!  OR &
   ldwis_itemqty  = datamodified! OR &
   ldwis_itemqty  = newmodified!  THEN

   lb_itemchange = true

      // Exit from the loop as soon as a changed row
      // is found.

   EXIT

ELSE

   lb_itemchange = false

END IF
```

Case Follow these guidelines:

Type these	In this case
Variable names	Lower
Names of built-in PowerScript functions	Mixed
PowerScript statement keywords	Upper

For example:

```
integer li_fetched_count = 0

SELECT Count("order_header"."order_id" )
   INTO :li_fetched_count
   FROM "order_header"
   WHERE "order_cust_id" = :ai_custid
   USING sqlca;

IF sqlca.sqlcode = -1 THEN li_fetched_count = -1

RETURN li_fetched_count
```

Declaring variables

These conventions have to do with how you declare variables for your scripts.

Placement Be consistent about where you declare the local variables in your scripts. Many developers place all local variable declarations for a particular script at the beginning of the script:

```
// First, declare all locals for this script.

   decimal lc_balance, lc_grandtotal_price

   string ls_highlight_format, ls_regular_format

   boolean lb_first_time
```

Some others prefer to organize their scripts into sections and place the appropriate variable declarations at the start of those sections, closer to where they are needed.

Scope Declare each variable you need at the *narrowest* level of scoping that makes sense for that variable:

Level	Scope
Narrowest	Local
↑	Instance
↓	Shared
Broadest	Global

For example, if a variable needs to exist only during the execution of a particular script, then declare it as a local in that script (as opposed to an instance, shared, or global variable). Avoid declaring variables at the global level (unless they really need to exist for the entire application).

Working with objects and controls

These conventions have to do with how you work with objects and controls in your scripts.

References When you need to refer to objects or controls, use PowerScript pronouns (such as *This*, *Parent*, and *ParentWindow*) whenever possible instead of typing actual object/control names explicitly. For example, suppose you're coding the Clicked event script for the cb_close CommandButton in the w_orderentry window. Instead of typing:

```
Close (w_orderentry)
```

You should type:

```
Close (parent)
```

This helps make your script code more reusable because it won't be tied to particular objects and controls.

Object-oriented programming (OOP) Make sure you follow good OOP practices as you work with the objects and controls in your application. This involves taking advantage of the object encapsulation, inheritance, and polymorphism features supported by PowerBuilder.

You'll learn more about OOP and good OOP practices later in this manual. For general background information, you might also read:

- *Object-Oriented Technology: A Manager's Guide* from Addison-Wesley Publishing Company
- *Object-Oriented Analysis and Design with Applications* from the Benjamin/Cummings Publishing Company

Documentation standards

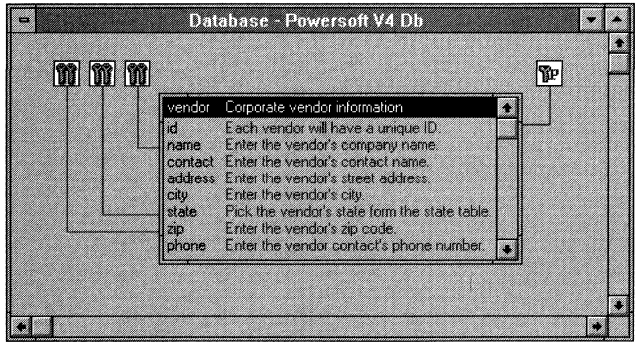
Providing good documentation is essential for any significant application. This includes both *internal documentation* for developers and *external documentation* for end users.

You should think about developing standards that specify how this documentation is to be done to ensure that it meets the needs of these audiences. Among the topics your documentation standards should address are: comments, online Help, and printed documentation.

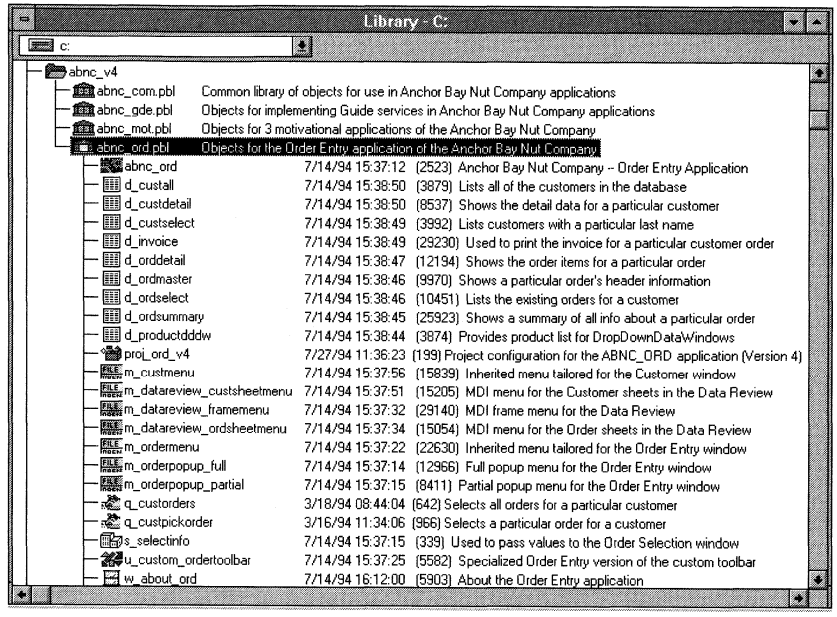
Commenting your work

In PowerBuilder, developers can supply comments for most anything they create, including tables, columns, objects, and scripts. Your documentation standards should probably require comments in all of these places.

Table and column comments Developers can enter these comments and read them in the Database painter:



Object comments Developers can enter and read these comments when editing objects in the various object painters. They can see them all at a glance in the Library painter:



Script comments It's wise to have your documentation standards call for thorough commenting in all event scripts and function scripts. After all, even the best code can be difficult to figure out on its own.

One good approach is to begin each script with a header comment that explains the script's purpose and describes any notable characteristics or requirements (such as arguments and return values for function scripts):

```
////////////////////////////////////////////////////
//                                                /
// Function: wf_warndataloss                      /
//                                                /
// Log:                                           /
//      5/20/94 (George) Initial version          /
//      9/13/94 (Elaine) Improved message text    /
//                                                /
////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////
//                                                /
// Purpose:                                       /
// This window-level user function is called from /
// various command buttons (Retrieve List, New    /
// Customer, Close) and menu items (Retrieve, New, /
// Exit Application) to check whether the user did /
// any typing in the dw_detail DataWindow control. /
// If so, it warns them that any unsaved data will /
// be lost if they proceed.                       /
//                                                /
// Argument:  as_titletext (a string that gets    /
//            inserted into the title of the      /
//            warning messagebox)                 /
//                                                /
// Return values:                                 /
//      *  1 if the user wants to proceed and lose /
//         the data.                              /
//      * -1 if the user wants to cancel the      /
//         operation and return to the dw_detail  /
//         DataWindow control.                    /
//                                                /
////////////////////////////////////////////////////
```

Then you can insert more specific comments throughout the body of the script to explain individual sections of code and what they do:

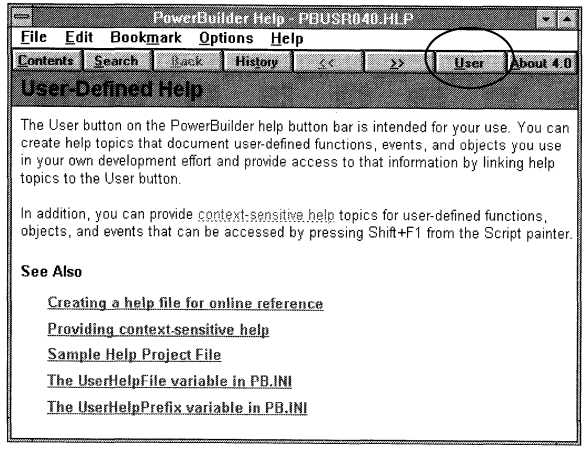
```
   // Test the flag that we set in the EditChanged
   // event of dw_detail.

IF ib_any_typing = true THEN

   Beep(1)
   IF MessageBox("Confirm " + &
       as_titletext + " Request", &
       "OK to lose unsaved values you've typed?", &
       question!, yesno!) = 2 THEN

         // If the user says no, set focus back on
         // the dw_detail DataWindow control and
         // then return -1.

      dw_detail.SetFocus( )
      RETURN -1

   END IF

END IF

   // If there was no typing in dw_detail or if the
   // user says yes (to proceed anyway and discard
   // the unsaved values), then return 1.

RETURN 1
```

Providing online Help

Your documentation standards should also specify what you want to do about online Help for a project. PowerBuilder enables you to provide two different kinds:

- Internal online Help that developers can display while using PowerBuilder
- External online Help that end users can display while running the completed application

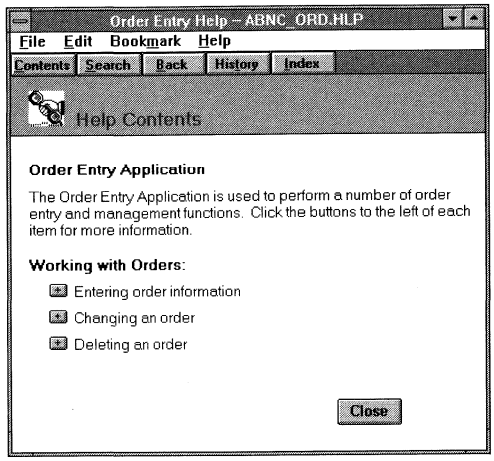
Online Help for developers A very useful way to document the various application components that your developers create is to write online Help modules about them. Once you do, you can link these modules right into the PowerBuilder online Help system through the special *User* button it provides:



Then developers can display these Help modules as they work in PowerBuilder to read information about their user-defined functions, user objects, user events, and whatever else you want to document.

Online Help for end users Users typically expect to be able to access online Help modules when running an application to get information about what that application does and how to use it. You should determine what these Help modules need to contain as well as how they should be organized and formatted.

Once your end-user Help modules are written, you can display them in an application by using a built-in function that PowerBuilder provides. For instance, here's a module that end users see when they ask for online Help in the Anchor Bay Nut Company's Order Entry application:



For more information on furnishing online Help for developers or end users, see Chapter 8, "Providing Online Help for an Application."

Providing printed documentation

In addition to comments and online Help, you may want to address printed documentation in your standards as well. For instance, you might consider requiring one or more of the following:

- **Developer reports** You can use the Library painter of PowerBuilder to print reports about the application components you've developed. You should determine which reports you need to print and how often.
- **Developer manuals** Would it be useful to have someone write one or more internal documents that developers could then reference for information on application components? If so, you need to specify what is to be described as well as how it is to be organized and formatted.
- **End-user manuals** Do users need training or reference documents to assist them as they run your application? If so, you should indicate what these documents are to cover as well as how they are to be organized and formatted.

Standard error and status routines

Housekeeping chores such as error and status checking are often good candidates for standardization. That's because you'll usually want to handle them the same way across many applications and because you'll want to minimize the chance of accidentally omitting a particular test.

Examples

For example, you might consider standardizing the way you check for:

- Network connection errors
- Database access errors
- Data entry errors

How you can handle them

Fortunately, the *object-oriented* features of PowerBuilder make it easy to standardize and implement common chores like these.

For instance, you'll usually want the DataWindow controls in your windows to check for various data entry and database access errors. The hard way to accomplish this is to write the appropriate event scripts in each individual DataWindow control. The easy, object-oriented way is to:

1 **Create** your own version of a generic DataWindow control by defining a user object for it.

2 **Write** your error-processing event scripts in this user-object DataWindow control.

3 **Inherit** the DataWindow controls you want to place in windows from your user-object DataWindow control. These inherited controls will *automatically* be able to perform your error processing.

You'll learn more about object orientation and its benefits later in this manual.

Designing and defining your databases

One of the most crucial setup steps for a development project involves preparing the database (or databases) required by your application. It can also be one of the most complicated and time-consuming steps.

Because database design and definition is such a broad topic, this section walks you through the main highlights that you need to think about right now. You'll read about:

- Starting with a good data model
- Assigning responsibility for a database
- Choosing tools for your database work
- Using the database tools in PowerBuilder
- Setting up for test versus production

Starting with a good data model

You should always begin any database development work by coming up with a model that accurately represents the nature of the data you're interested in. A good **relational data model** will enable you to understand how your data must be organized (into columns and tables) and related (through keys) before you actually start defining the database.

In many ways, a database serves as the foundation for every application that uses it. Unless that database is developed from a solid and sensible data model, you could be building your applications on a house of cards.

Further design questions Once you have the basic data model, you need to answer a few more specific questions, including these:

- What **kind of DBMS** is best for implementing this database?

  Different DBMSs (such as SQL Server, Oracle, or Watcom SQL) have different strengths. If you're in a position to choose a DBMS, you should select one that suits the needs of your database and its applications (in terms of scale, speed, features, cost, and ease of use).

- Do you need to provide **network access** to this database?

  Although client/server applications typically require their databases to be accessible through networks, you may sometimes find that a local database is all that's needed. When you do need a network database, the next question can be particularly important.

- What **processing capabilities** do you want to build into the database (instead of into each of your applications)?

Many DBMSs provide features such as stored procedures and referential integrity checking that enable you to perform a great deal of processing at the database level. You'll need to decide how much to exploit these features depending on factors like: the degree of centralization you want to institute, whether a particular feature is DBMS-specific (and might affect portability to a different DBMS in the future), and whether you're trying to reduce network traffic (by figuring out which processing is best to do on the client computer and which is best to do on the server computer).

> **Keep this in mind**
> The issue of whether to handle processing in your client application or on the server database isn't just important when you're developing that database. It also comes up when you're actually building the application and need to decide whether it or the database should perform a particular selection, sort, join, or other operation. More about this later.

Reading about good database design In addition to the manuals that accompany your DBMS, there are plenty of informative books you can read to learn about designing databases. Here are some you might start with:

- *Relational Database Design: An Introduction* from Prentice Hall
- *The Design of Relational Databases* from Addison-Wesley Publishing Company

Assigning responsibility for a database

If you're working on a project team, it's wise to assign responsibility for all database design and definition to one person (or a small group). You may even have a dedicated database administrator (DBA) to handle this aspect of your project.

This will help ensure that your database is organized intelligently and implemented in a consistent fashion.

Choosing tools for your database work

You'll usually have a lot of options when it comes to choosing tools for designing and defining a database. These options include:

- **The tools provided with the DBMS you're using**

  You might use these tools to fully develop a database or maybe just to take care of essential administrative chores—in particular, creating or deleting a database.

- **The database tools provided in PowerBuilder**

  Once you've created a database with the tools of your DBMS, you can use the database tools in PowerBuilder to develop that database the rest of the way, and then populate and maintain it. (For a Watcom SQL database, you can have PowerBuilder do the database creation step for you.)

  At minimum, you'll want to use these tools to take advantage of some very useful extensions they let you add to the definitions of your tables and columns. You'll learn more about those in just a moment.

- **CASE and other supplementary tools**

  These tools are particularly useful in large-scale environments where you must develop entire systems of databases and applications. There are many such tools available, but you'll benefit most by using those that are made to work with PowerBuilder.

  For a list of those tools, look in your PowerBuilder package or talk to a Powersoft sales representative.

Using the database tools in PowerBuilder

PowerBuilder includes several different **painters** that you can use to handle various kinds of database development chores.

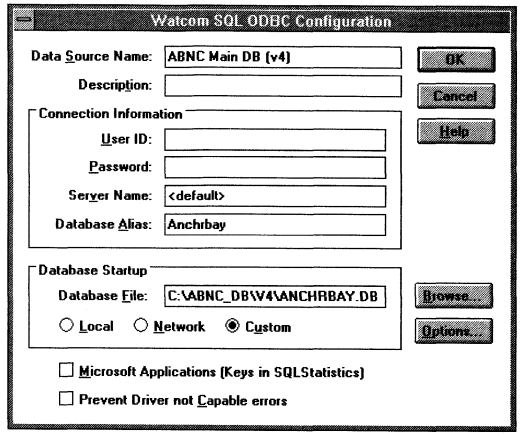
Preparing to use them Before you can use these painters to work on a particular database, you must be able to access that database. This involves the following steps:

1 Making sure the appropriate **database interface** is installed

  In the previous chapter, you learned about the database interfaces that PowerBuilder uses to talk to each kind of DBMS you access. You probably installed the databases interface(s) you usually need when you installed PowerBuilder. But if you decide to access a new DBMS, you must install the appropriate database interface for it before you can work with that DBMS in the PowerBuilder painters.

2 Specifying **ODBC configuration information** (*for ODBC-compliant databases only*)

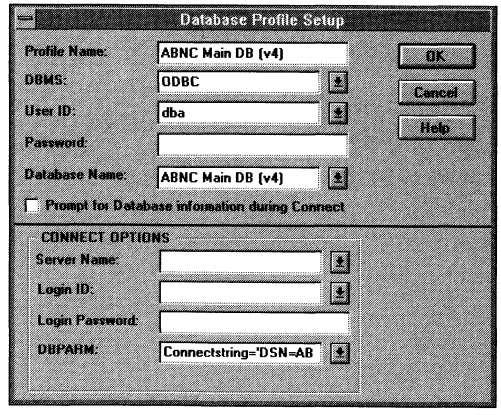
  If you want to access a particular database through the ODBC interface (as opposed to one of the other Powersoft database interfaces), you must define it to ODBC as a data source. For example, the main database used in the Anchor Bay Nut Company's Order Entry application is a Watcom SQL database that's defined to ODBC as follows:



3 Defining a **database profile** for the database

When you use one of the painters to work on a database, PowerBuilder must first connect to that database. If you want, you can have PowerBuilder prompt you for the parameters it needs to make the connection (such as the database name, user ID, password, and so on). But if you plan to work with that database more in the future, it's much better to define a database profile (which saves your parameters and enables you to connect automatically).

For example, at the Anchor Bay Nut Company people define a database profile named *ABNC Main DB (v4)* for their main database:



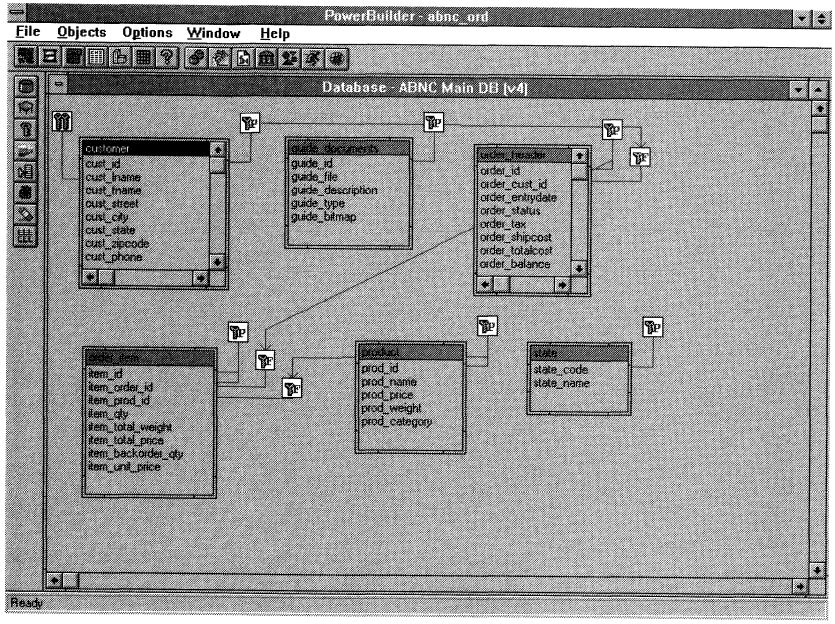
Shortcut for Watcom SQL databases
When you use PowerBuilder to create a new Watcom SQL database, it takes care of specifying the needed ODBC configuration information and defining a database profile.

For more information on the preceding steps, see *Connecting to Your Database.*

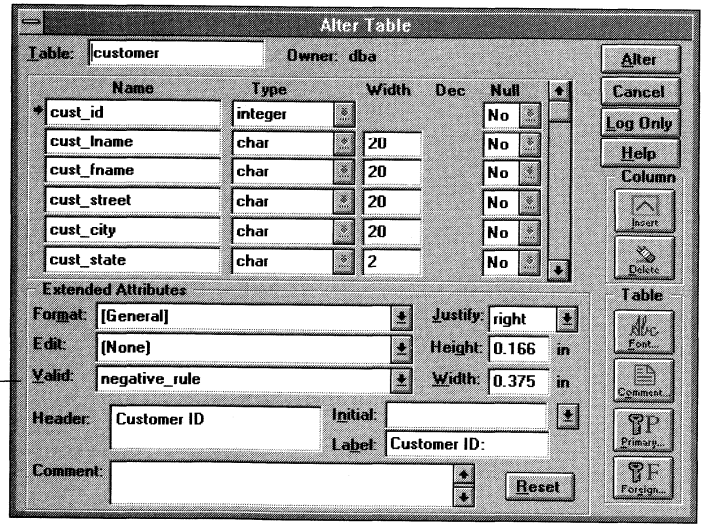
What they can do for you Once your environment is set up properly, you can begin your database development work. Here's a brief introduction to each of the painters that PowerBuilder provides to help you do that work:

Use this	When you want to
Database painter	Define tables, columns, keys, or indexes
	Create or delete Watcom SQL databases
View painter	Define views
Data Manipulation painter	Retrieve, insert, update, delete, or save table rows
Database Administration painter	Define database security or send SQL statements to the DBMS for immediate execution
Data Pipeline painter	Migrate data from one or more source tables to a new or existing destination table (either within a database or across databases)

A closer look For example, at the Anchor Bay Nut Company they used the *Database painter* to define the various tables, columns, keys, and indexes for their main database:



In particular, here's how they defined the Customer table and its columns:



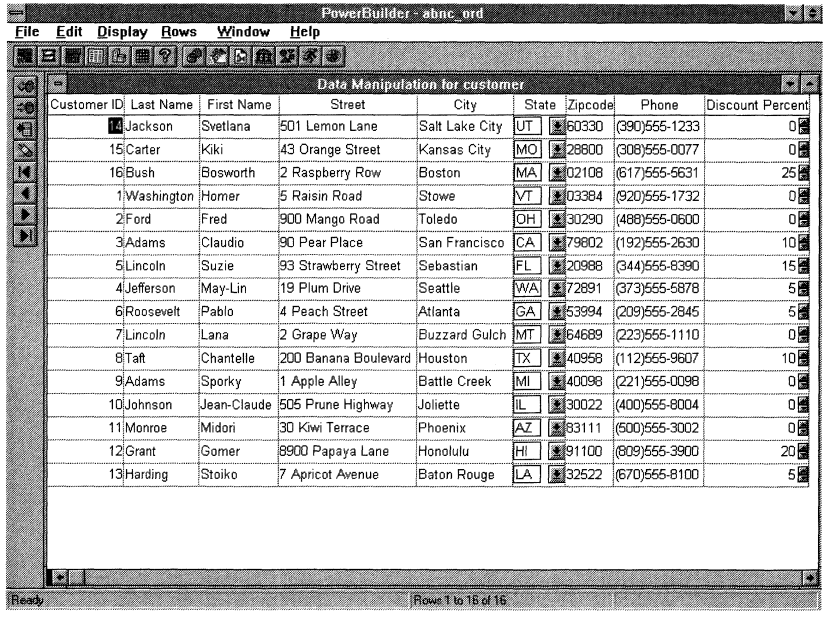
Extended attributes for the cust_id column

About extended attributes

Notice that the Database painter provides a number of extended attributes that you can specify for each column. This enables you to attach application-oriented information (such as display formats, validation rules, initial values, and header or label text) to columns for use throughout your applications. As a result, extended attributes provide you with an excellent mechanism for implementing certain kinds of standards.

PowerBuilder maintains extended attributes in a **repository** (something like a data dictionary), which it stores in some special system tables inside your database.

The Anchor Bay staff also used the *Data Manipulation painter* to populate the tables in their test version of the main database. For instance, here's how they populated the Customer table:



Customer ID	Last Name	First Name	Street	City	State	Zipcode	Phone	Discount Percent
14	Jackson	Svetlana	501 Lemon Lane	Salt Lake City	UT	60330	(390)555-1233	0
15	Carter	Kiki	43 Orange Street	Kansas City	MO	28800	(308)555-0077	0
16	Bush	Bosworth	2 Raspberry Row	Boston	MA	02108	(617)555-5631	25
1	Washington	Homer	5 Raisin Road	Stowe	VT	03384	(920)555-1732	0
2	Ford	Fred	900 Mango Road	Toledo	OH	30290	(488)555-0600	0
3	Adams	Claudio	90 Pear Place	San Francisco	CA	79802	(192)555-2630	10
5	Lincoln	Suzie	93 Strawberry Street	Sebastian	FL	20988	(344)555-8390	15
4	Jefferson	May-Lin	19 Plum Drive	Seattle	WA	72891	(373)555-5878	5
6	Roosevelt	Pablo	4 Peach Street	Atlanta	GA	53994	(209)555-2845	5
7	Lincoln	Lana	2 Grape Way	Buzzard Gulch	MT	64689	(223)555-1110	0
8	Taft	Chantelle	200 Banana Boulevard	Houston	TX	40958	(112)555-9607	10
9	Adams	Sporky	1 Apple Alley	Battle Creek	MI	40098	(221)555-0098	0
10	Johnson	Jean-Claude	505 Prune Highway	Joliette	IL	30022	(400)555-8004	0
11	Monroe	Midori	30 Kiwi Terrace	Phoenix	AZ	83111	(500)555-3002	0
12	Grant	Gomer	8900 Papaya Lane	Honolulu	HI	91100	(809)555-3900	20
13	Harding	Stoiko	7 Apricot Avenue	Baton Rouge	LA	32522	(670)555-8100	5

For more information on all of the database-related painters in PowerBuilder, see the *User's Guide*.

Setting up for test versus production

Once you have a database, you need to decide how you want developers to access it during their application development project. Consider these scenarios:

- **Protecting production data** Suppose you've got an existing production database that's currently accessed by applications and users, or at least contains live data. You certainly won't want developers to be testing their new work against this database.
- **Developing without the network** Suppose you've defined your database under a server DBMS (such as SQL Server or Oracle) but you don't want to have to be connected to the network to do application development work (maybe because you want to be able to work at home).

The solution in either case is to create a separate *test version* of your database.

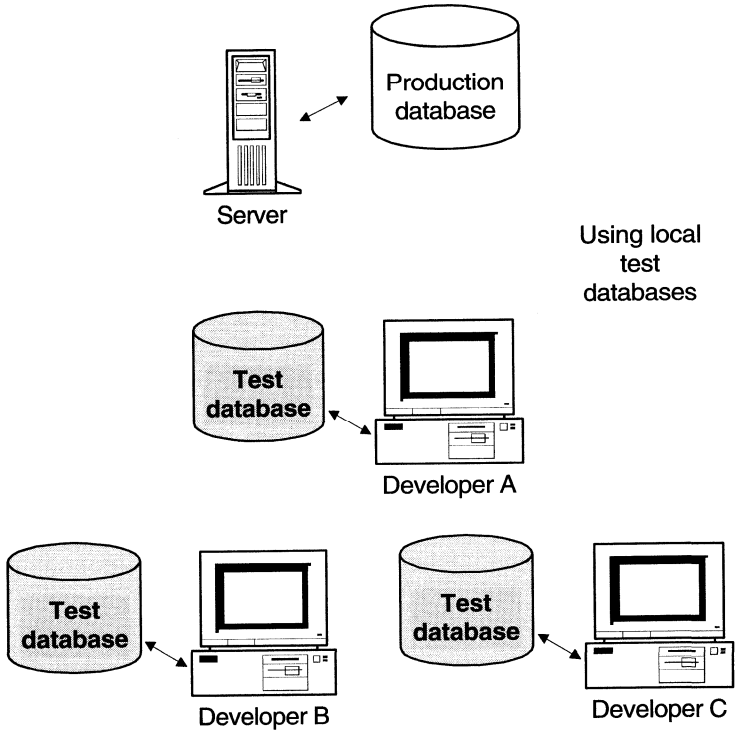
Guidelines for a test database Setup (and maintenance) of a test database is best done by the same people who are responsible for the production database itself. And it should be complete before the development of application components begins.

When setting up a test database, you need to decide where it should be implemented. Here are a couple of common approaches:

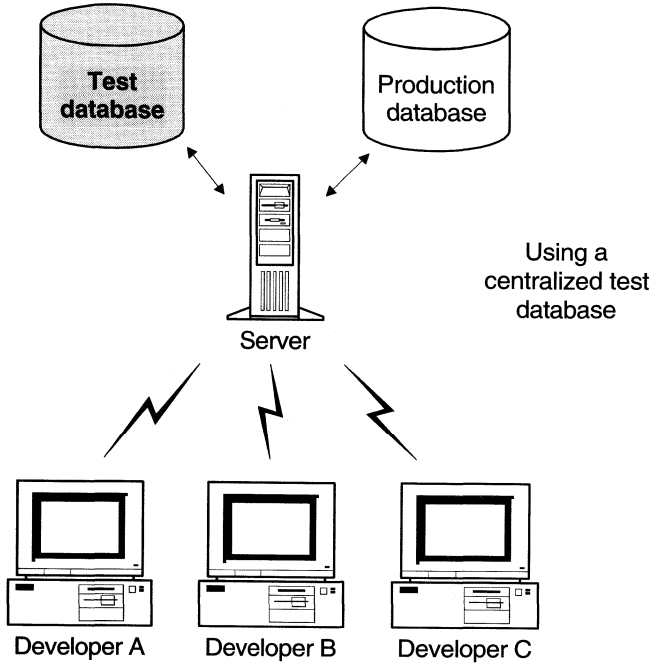
Approach	What it involves
Local test databases	With this approach, you create a local Watcom SQL version of the database, which developers can then install on their own computers. Consider doing this if you want to be able to develop and test application components in a decentralized way—without being connected to the network. Just be aware that: ♦ **You may need to work around DBMS-related differences** if your test and production databases don't use the same DBMS. Such differences might involve data types and supported features (such as stored procedures). ♦ **The application's performance may be different** when accessing the local test database as opposed to the network server database. So you should leave time at the end of the project for tuning.

Approach	What it involves
Centralized test database	With this approach, you create the test database under the same server DBMS as the production database so that all developers can access it from the network. Consider doing this if you want to keep your test environment as similar to the production environment as possible (to avoid the need for changes at the end of the project when you switch over to the production database).

Here's an illustration of the *local approach* (you may recall from the previous chapter that the Anchor Bay Nut Company used this approach for their Order Entry application):



In contrast, this is what the *centralized approach* looks like:



Accessing a test database Once your test database is set up, you can use it in PowerBuilder simply by connecting to it instead of to the production database. This will enable you to access the test database automatically while you're working in the PowerBuilder painters to develop application components.

Similarly, you can point your application itself to the test database during the development phase of your project. That way, whenever you run the application to check your work and look for bugs, it will access the test database instead of the production one. Later in the project as you approach deployment, you can point your application back to the production database.

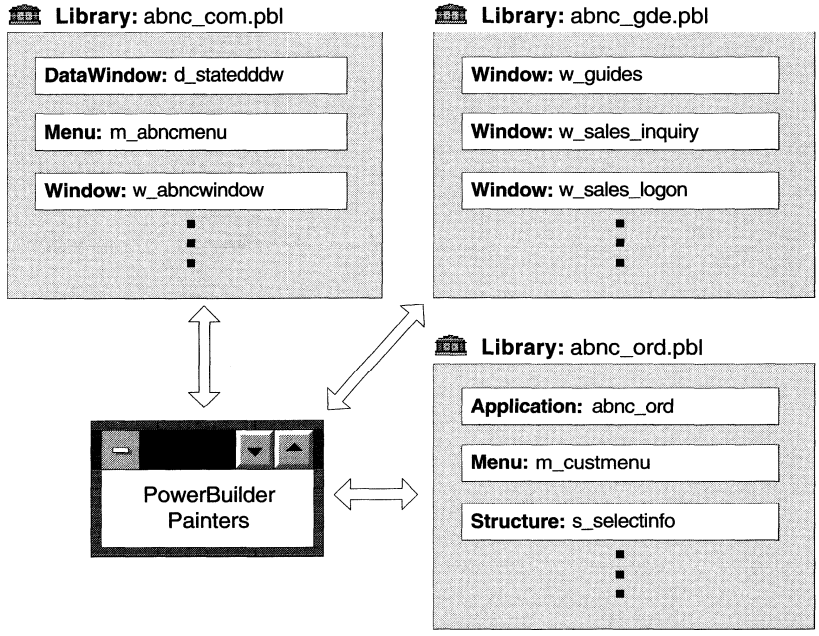
You'll learn the details of how to connect to databases from an application later in this manual.

Organizing your work into libraries

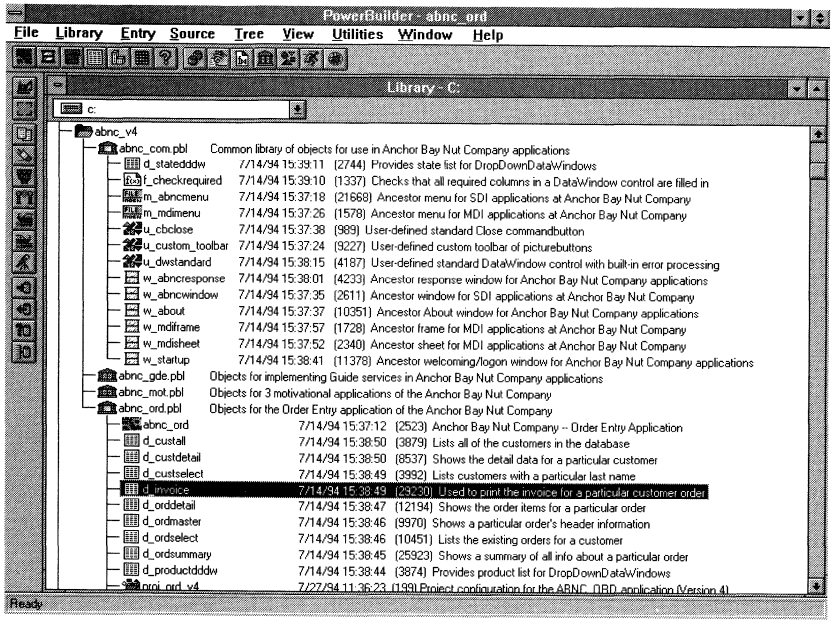
In addition to its database-related painters, PowerBuilder provides several other painters that you use to create the components (such as windows and menus) of your application. When you're working in these painters and ask to save components, PowerBuilder needs a place to store and organize them. That's where **libraries** come in.

Introducing libraries

A library is the container in which you collect a particular set of components for use in your applications. It's where a component goes when you save it in a painter and where a painter gets a component you ask to open. PowerBuilder maintains each library you use in its own **PBL file** in your operating system.



To help you organize these libraries and control the components they contain, PowerBuilder includes a tool called the **Library painter**:



How you organize your libraries is up to you. For instance, you can:

- Create one or more libraries for a particular application
- Use the same library in more than one application
- Store libraries in different locations, including your computer or a server

We'll offer you some guidelines on what's good to do in just a moment.

Introducing objects

Now it's time to take a closer look at the components you store in your libraries. Up to this point, you've learned what several of these components are used for, but not much about how they work. The truth is that things like windows and menus are not merely pieces of a particular application, but self-contained **objects**.

That means each one *encapsulates* the particular characteristics and behaviors (attributes, events, and functions) that are appropriate to it. It also means that they support other standard object-oriented capabilities, including *inheritance* and *polymorphism*. Later you'll learn how to take advantage of these features in the applications you build. But for the moment you just need to know that they'll benefit you in several important ways—in particular by making your work more modular, reusable, extensible, flexible, and powerful.

Kinds of objects

Here's a list of the PowerBuilder painters that keep their objects in your libraries:

Painter	Kind of object it stores in libraries
Application painter	 Application object
Data Pipeline painter	 Pipeline object
DataWindow painter	 DataWindow object
Function painter	 Function object
Menu painter	 Menu object
Project painter	 Project object
Query painter	 Query object
Structure painter	 Structure object
User Object painter	 User object
Window painter	 Window object

For more information on using these painters, as well as the Library painter, see the *User's Guide*.

Other kinds of objects

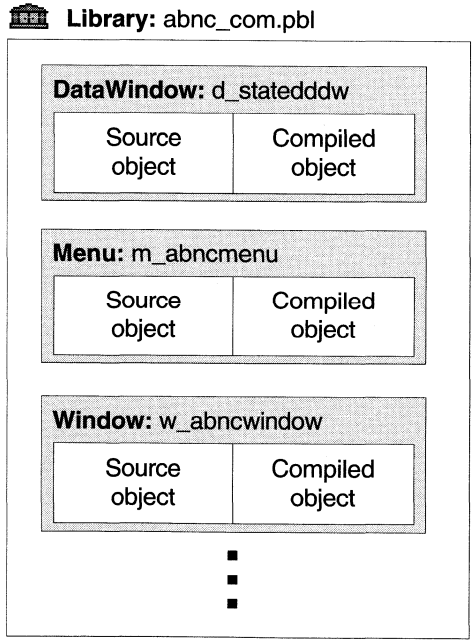
In addition to the objects you create in the painters, PowerBuilder provides a number of **system objects**. These objects aren't maintained in libraries. Instead, they're managed by PowerBuilder itself (although you can work with them in your scripts). You'll learn more about examining and using system objects later.

How libraries store objects

Whenever you save an object in a library, PowerBuilder actually stores *two* forms of that object:

This form of an object	Is
Source	A syntactic representation of the object, including any script code it contains. You can think of this as the object's definition.
Compiled	A binary representation of the object (similar to an OBJ file in the C language) that's compiled and ready for execution. Every time you save an object in a painter, PowerBuilder automatically compiles that object for you.

Here are some examples:



This is important to you because it means that libraries play a central role not only during the development of objects, but during their execution as well.

When building a new application

If your project involves building a new application, you'll want to create at least one library for it. This library is needed to hold the application object for your application (but you can put any other objects you want in it too).

Creating a library for your application

You can create this library in either of the following ways:

- **By using the Library painter**

  Once the library exists, you can: go to the Application painter, create your application object, and then store it in that library.

- **By using the Application painter**

  The Application painter lets you specify the name of a library you want to create in case you don't already have one to hold your new application object.

For more information on the application object, see "Specifying the logistics of your application" on page 136.

Using additional libraries

Next, you can use the Library painter to create any number of additional new libraries to hold the objects you plan to develop for your application. You may not need any more or you may need several—it all depends on how many objects you think you'll be storing and how you want them organized.

Fortunately, you're not required to figure this out right now. You can reorganize at any point in a project (by creating libraries, or deleting them, or moving objects among them). In fact developers often do this, because it's only as an application evolves that its optimal organization becomes clear.

Taking advantage of existing libraries Of course you aren't limited to only new libraries in your application. You might also want it to access one or more *existing* libraries in order to use the objects they contain.

To get the most out of PowerBuilder, you should try to take advantage of this reusability feature whenever possible. It will save you a great deal of time as well as improve the quality and consistency of your applications.

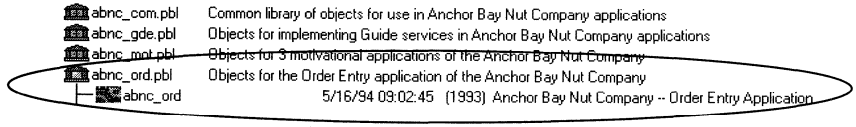
For more information on using objects in existing libraries, see "Using an application framework" on page 130.

Listing all of your libraries Whenever you want to use multiple libraries in your application (whether they're new ones, existing ones, or both), you need to specify a list of their names. You can go to the Application painter to do this.

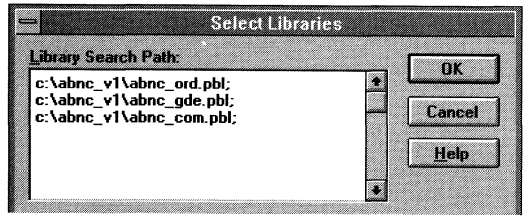
For more information, see "Specifying the logistics of your application" on page 136.

Example

Here's a typical case. When the developers at the Anchor Bay Nut Company set up their initial version of the Order Entry application, they created a library called ABNC_ORD.PBL and placed their new application object (abnc_ord) in it:



But they also wanted to use various objects from a couple of existing libraries (ABNC_COM.PBL and ABNC_GDE.PBL), so they added these names to the list of libraries for the application (in the Application painter):



When updating an existing application

If your project involves building a new version (release) of an application, you'll normally want to make copies of the existing libraries and then use those copied libraries as your starting point. That way, you can implement whatever enhancements are required while still preserving the source from the previous release. This can be particularly important if that previous release of the application is currently in production use and needs to be maintained independent of the new release.

Copying libraries for your new release

The approach you'll take to do this copying depends on whether or not you're using a version control system with PowerBuilder for your project:

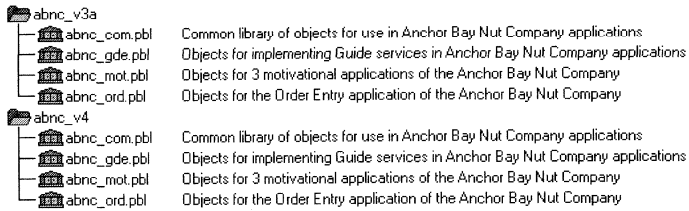
If you	Then
Are using a version control system	Go to the Library painter and use its Create New Release facility. This facility lets you: ♦ Copy the application's PBL files ♦ Copy and adjust any corresponding version control files
Aren't using one	Go to your operating (windowing) system and use it to make new copies of the application's PBL files.

For more information on the Library painter, see the *User's Guide*. To learn more about version control, see "Maintaining versions of objects" on page 129.

Example Not long ago, the developers at the Anchor Bay Nut Company were assigned the project of creating an updated release of their Order Entry application. As part of their setup phase for this project, they copied the libraries from the previous release of the application (Version 3a) for use in the new release (Version 4):

Libraries for previous version

Copied libraries for new version



Going back to work on an earlier release

Sometimes while you're working on a new release of an application, you'll need to go back to a previous release and make changes (typically, bug fixes). This is often just a matter switching to the libraries of that earlier release, editing the appropriate objects from them, and then building a new executable for that release.

If you're using a version control system, PowerBuilder provides additional tools to help you go back to earlier work. In particular, you can use the Restore Libraries facility in the Project painter to recreate all of your application's libraries as they were when you built the executable for a previous project.

For more information on the Project painter, see the *User's Guide*. To learn more about version control, see "Maintaining versions of objects" on page 129.

Guidelines for organizing libraries

The organization of your libraries is important because it affects both:

- **How easy** it is for you to develop and maintain your applications
- **How well** your applications perform when users run them

In fact, project teams often have someone dedicated to the job of organizing libraries and managing the objects they contain. If that person is you or if you're working independently, you should observe the following guidelines to ensure that you're using your libraries appropriately.

Library size

Although there's no limit to how large a library can be, it's wise to keep them *smaller than about 800K*.

As a library gets larger than that, performance can suffer—because PowerBuilder has to search more in order to save or open an object. To find out how big your libraries are, look up the size of their PBL files in your operating system.

Number of objects

You'll usually want to store *no more than about 50 or 60* objects in a particular library.

This limit is simply for your convenience (the number of objects doesn't affect performance). When you have more objects than that in a library, you'll find that it's tedious to look through library-object listboxes that PowerBuilder displays and that it's not as easy to work in the Library painter.

Number of application objects in a library

In most cases, you should store *no more than one application object* in a particular library. (Of course you'll probably have many libraries that don't contain any application objects at all.)

It is acceptable to store multiple application objects in a library if they all have the same global declarations.

Number of libraries

Your goal should be to use the *minimum number of libraries of reasonable size* that still allows you to organize your objects appropriately.

Maintain a balance between the size and number of your libraries. You don't want to have to manage a large number of libraries containing just a few objects each. That makes the library search path too long and can affect performance by forcing PowerBuilder to look through many libraries to find an object.

Library optimization

Make a habit of optimizing your libraries (in the Library painter) *on a regular basis.*

Library (PBL) files can become internally fragmented over time, especially when you make a lot of changes to the objects in them. That means they can end up containing gaps of unused space and maintaining some objects in noncontiguous areas of storage. This can affect performance when you're working in the PowerBuilder development environment.

Optimizing unfragments the internal storage of PBL files. (It doesn't alter the contents of objects or cause them to be recompiled.) To ensure good performance, consider optimizing any libraries you're actively working on about once a week.

> **Tip**
> It's *not* necessary to optimize your libraries before creating an executable (EXE) file or dynamic libraries (PBD files) for your application.

Library organization schemes

If your application is fairly simple, you may need only one library to hold all of its objects. But many typical business applications will involve a large number of objects and require you to organize them into multiple libraries. If that describes your application, here are a couple of organization schemes you should consider.

Organizing by object type One obvious approach is to divide your objects into different libraries based on their type. For instance, you could set up:

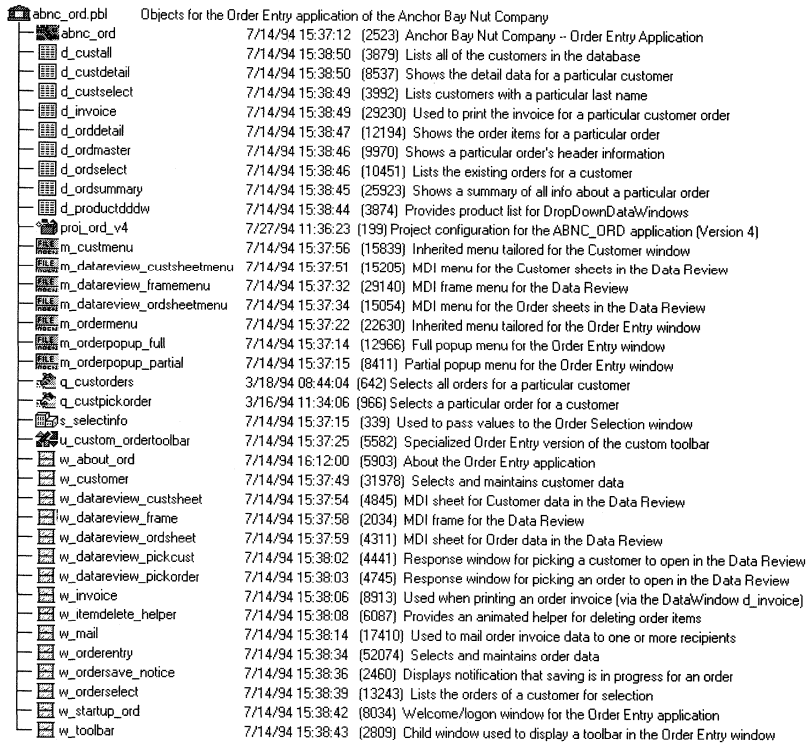
- **One library** to hold all of your window objects, DataWindow objects, and function objects, *and*
- **Another library** to hold your application object as well as all of your menu objects, structure objects, and user objects

Because this approach is straightforward, it can be an acceptable choice for small- or medium-size applications. But it's usually *not recommended for large-scale applications*. That's because it doesn't consider which objects are most commonly used or used together, and so it often forces PowerBuilder to do more searching through libraries. This can affect performance.

Organizing by role A more useful and efficient approach is to organize objects according to the role they play in your application. This involves grouping together all of the objects that participate in a particular feature or subsystem.

The Anchor Bay Nut Company's Order Entry application is a good example of this scheme. In it, objects are organized into three different libraries:

- **ABNC_ORD.PBL** This library contains objects used to implement order entry activities:



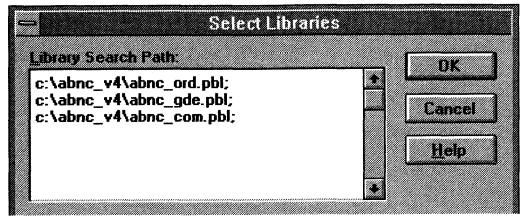
- **ABNC_GDE.PBL** This library contains objects used to implement some peripheral activities (such as the display of guidance information), which may be needed in other applications too:

```
abnc_gde.pbl     Objects for implementing Guide services in Anchor Bay Nut Company applications
  d_guidedocs          9/30/94 08:15:23  (5841) Lists quide documents (policies and procedures) that users can access
  d_sales_actsumgraph  9/30/94 08:15:23  (6476) Graph showing the sum of actual sales by Rep
  d_sales_extract      9/30/94 08:15:23  (7418) Lists rows currently in the Quarterly_Extract table of the Sales database
  d_sales_graphdata    9/30/94 08:15:22  (6628) Supplies selected data to be shared by graph DataWindows
  d_sales_repcrosstab  9/30/94 08:15:22  (15087) Crosstab showing rep actual sales compared to quota
  d_sales_repgraph     9/30/94 08:15:20  (6552) Graph showing rep actual sales compared to quota
  d_sales_teamcrosstab 9/30/94 08:15:20  (14738) Crosstab showing team actual sales compared to quota
  d_sales_teamgraph    9/30/94 08:15:19  (6597) Graph showing team actual sales compared to quota
  m_guides             9/30/94 08:14:36  (6116) Menu used with the w_guides window
  m_sales_inquiry      9/30/94 08:14:35  (11059) Menu used with the w_sales_inquiry window
  pipe_sales_extract1  9/14/94 14:37:09  (1961) Pipeline data definitions for creating new Quarterly_Extract table in Sales database
  pipe_sales_extract2  9/14/94 14:39:17  (1967) Pipeline data definitions for inserting new rows into existing Quarterly_Extract table
  u_sales_pipe_logistics 9/30/94 08:14:36  (1552) Class inherited from pipeline system object (to handle pipeline logisitics)
  w_guides             9/30/94 08:14:41  (12320) Displays guide info about company policies or phone procedures
  w_sales_extract      9/30/94 08:14:55  (41266) Dialog box for writing Quarterly_Extract table via pipelines
  w_sales_graphrotate  9/30/94 08:14:57  (10221) Lets the user rotate a 3D graph displayed in w_sales_inquiry
  w_sales_graphtopic   9/30/94 08:14:59  (9246) Lets the user change the graph topic displayed in w_sales_inquiry
  w_sales_graphtype    9/30/94 08:15:02  (13853) Lets the user change the graph type displayed in w_sales_inquiry
  w_sales_inquiry      9/30/94 08:15:16  (25550) Displays inquiry information from the Sales database (ABNCSALE.DB)
  w_sales_logon        9/30/94 08:15:18  (10307) Prompts for logon to the Sales database (ABNCSALE.DB)
```

- **ABNC_COM.PBL** This library serves as the *application framework* for this and other Anchor Bay Nut Company applications. That means it contains various *ancestor objects* from which many of the objects in the other libraries are inherited:

```
abnc_com.pbl     Common library of objects for use in Anchor Bay Nut Company applications
  d_statedddw          7/14/94 15:39:11  (2744) Provides state list for DropDownDataWindows
  f_checkrequired      7/14/94 15:39:10  (1337) Checks that all required columns in a DataWindow control are filled in
  m_abncmenu           7/14/94 15:37:18  (21668) Ancestor menu for SDI applications at Anchor Bay Nut Company
  m_mdimenu            7/14/94 15:37:26  (1578) Ancestor menu for MDI applications at Anchor Bay Nut Company
  u_cbclose            7/14/94 15:37:38  (989) User-defined standard Close commandbutton
  u_custom_toolbar     7/14/94 15:37:24  (9227) User-defined custom toolbar of picturebuttons
  u_dwstandard         7/14/94 15:38:15  (4187) User-defined standard DataWindow control with built-in error processing
  w_abncresponse       7/14/94 15:38:01  (4233) Ancestor response window for Anchor Bay Nut Company applications
  w_abncwindow         7/14/94 15:37:35  (2611) Ancestor window for SDI applications at Anchor Bay Nut Company
  w_about              7/14/94 15:37:37  (10351) Ancestor About window for Anchor Bay Nut Company applications
  w_mdiframe           7/14/94 15:37:57  (1728) Ancestor frame for MDI applications at Anchor Bay Nut Company
  w_mdisheet           7/14/94 15:37:52  (2340) Ancestor sheet for MDI applications at Anchor Bay Nut Company
  w_startup            7/14/94 15:38:41  (11378) Ancestor welcoming/logon window for Anchor Bay Nut Company applications
```

This organization makes sense from a business point of view. But it also enables them to list their libraries (in the Application painter) according to frequency of use:



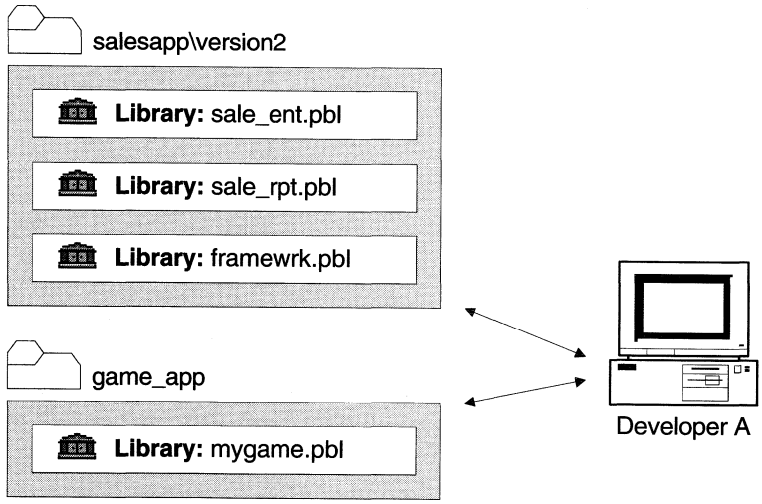
That ensures optimal performance. (Notice that the library containing the ancestor objects is listed last.)

To learn more about using a library like ABNC_COM.PBL, see "Using an application framework" on page 130.

Library management during a project

Another setup chore for your project involves how you want to manage your libraries and their objects throughout the development, testing, and deployment phases. Many variations of library management models are possible, but here are a couple of the typical ones.

Single-developer model This model is good for independent developers or for development team members who are working on small, self-contained applications. It simply involves setting up the application's libraries on the developer's computer so that the developer can work with them locally.



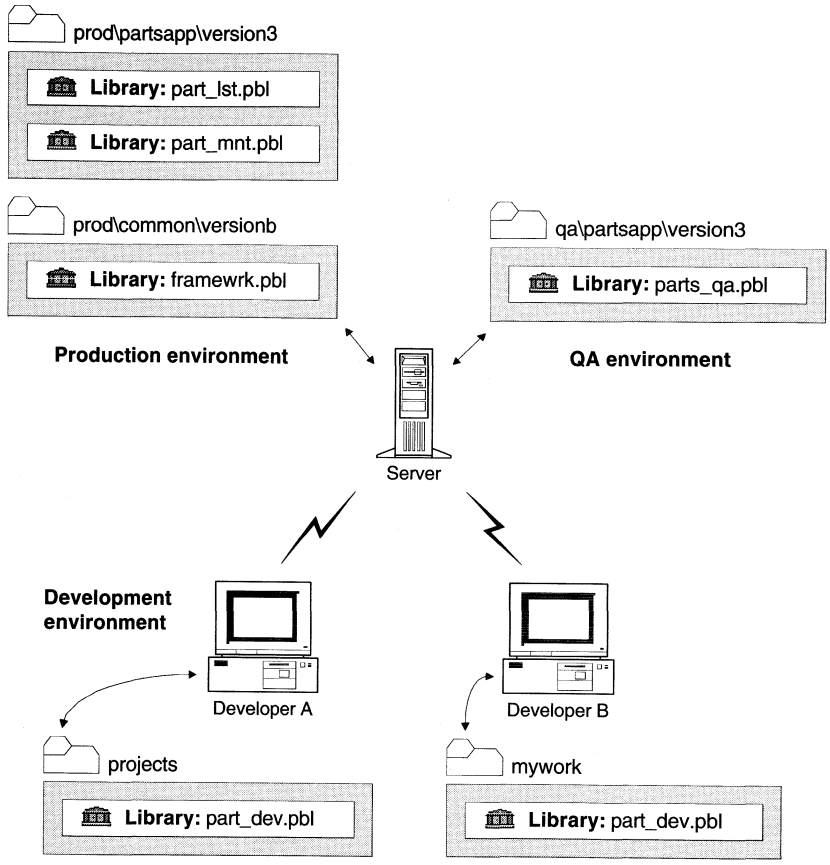
If you have a version control system, you might consider using it in this model (in order to gain some extra control over your source).

For more information, see "Maintaining versions of objects" on page 129.

Multiple-developer model To manage a project in which multiple developers will take part, you'll usually want to set up a three-tier model that includes a production environment, a quality-assurance (QA) environment, and a development environment. Here are the details:

For	Do this
Production	Set up your production libraries on the server and organize them as described in "Library organization schemes" on page 122. Make sure they are read-only and under the control of a particular person (such as a project manager, object manager, or QA manager). Objects to be edited are *checked out* of the production libraries and placed into a QA library. Completed objects are *checked in* to the production libraries from that QA library. For more information on checking out and checking in objects, see "Controlling access to objects" on page 128.
QA	Set up one QA library for the project and locate it on the server. (You might choose to use multiple QA libraries for a project—it's up to you.) Keep it under the control of a particular person (a QA manager). Define the QA *library path* for the application in this order: 1 QA library 2 Production libraries Any object to be edited is *copied from* the QA library by an individual developer and placed into that developer's local development library for the project. When done with the object, the developer *copies it back* into the QA library (so that it can be tested and approved before checking it into the appropriate production library).
Development	Have each developer set up one local development library (or possibly more) for the project. The *library path* they define for the application should be in this order: 1 Local development library 2 QA library 3 Production libraries The local development library of a particular developer contains only the objects that developer is working on. When done with those objects, the developer copies them back into the QA library.

Here's an illustration of this model:



A multiple-developer model such as this typically calls for the use of source control. You'll learn about that in just a moment.

Cross-platform use of libraries

Libraries and the objects in them are *platform independent*. That means you can move or share your PBL files across platforms (such as Microsoft Windows and Apple Macintosh) during an application development project.

Managing your work with source control

If your project involves a team of developers, it's a good bet that you'll want to employ some form of source control to maintain the integrity of your work. But even a developer working solo can benefit from certain source-control features (such as the ability to get back to an earlier version of an object).

To support your requirements for source control, PowerBuilder provides:

- Some basic features of its own that enable you to control access to objects
- Interfaces to external version control systems, which enable you to take advantage of their features as you work within PowerBuilder

Controlling access to objects

As you just saw in the previous section, controlling access to the objects in your libraries is crucial on any multiple-developer project. PowerBuilder helps give you this control by providing the ability to **check out** and **check in** objects.

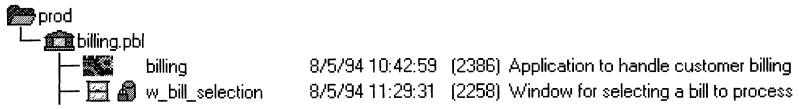
How it works

Suppose you want to edit an object that's stored in a public (shared) library on a server. You can use the Library painter to check out that object from the public library, placing a copy of it in one of your own libraries.

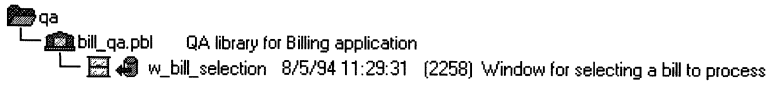
As long as you have that object checked out, no one else can update the original in the public library (although they can open it as read-only). When you finish working on the object and check it back in, PowerBuilder replaces the original in the public library with your copy, and then deletes the copy from your library.

Example

Here's an example in which a developer checks the window object w_bill_selection out of the production library BILLING.PBL and places a working copy in the QA library BILL_QA.PBL. Notice that the Library painter indicates the object is checked out of BILLING.PBL by displaying a picture of a lock beside it:



If you look in BILL_QA.PBL, you'll see the copy of w_bill_selection. You'll know it's an object that must eventually be checked in because the Library painter displays a picture of an arrow beside it:



For all of the details on using the Library painter to check out and check in objects, see the *User's Guide*.

Maintaining versions of objects

Many developers rely on version control systems to track and manage the changes made to their application source over time. Such systems provide a fine level of object control, enabling you to move back and forth through the history of a project.

PowerBuilder supports a variety of popular version control systems, primarily through its Library painter. If you install one of these systems and set it up in PowerBuilder, you'll be able to use its features in the Library painter during your projects. Those features include: registration, check-out/check-in, reporting, and application-specific configuration.

For all of the details on setting up and using version control in PowerBuilder, see *Version Control Interfaces*.

Using an application framework

To get the most out of PowerBuilder, you've got to take full advantage of its object-oriented features. And the best way to do that is to base each project you begin on an **application framework**.

What it is

There's some variation in how people use the term application framework, but it essentially refers to a collection of **base classes** (ancestor objects) from which you can inherit most of the application-specific objects you need to develop for your project.

The idea is that you define these base classes with the characteristics and behaviors (attributes, events, and functions) you commonly want in the objects you construct. That way, when you create an object inherited from one of them, you get all of the characteristics and behaviors of that base class in your new object automatically.

Benefits of using one

This is much more than just a convenient way to copy features from one object to another. That's because your inherited objects don't duplicate the features of their base classes, but *refer* to them. This makes it easy for you to apply global changes, since modifications you make in a base class are automatically picked up by its descendants. And when you execute your application, it can perform more efficiently because those inherited features are loaded just once (via the base class).

In other words, using an application framework can help make your applications better in a variety of ways (more consistent and modular, easier to fix and extend, faster and leaner). It can also make them much quicker to build, since you'll be able to reuse so much of your work on each one.

An example of one

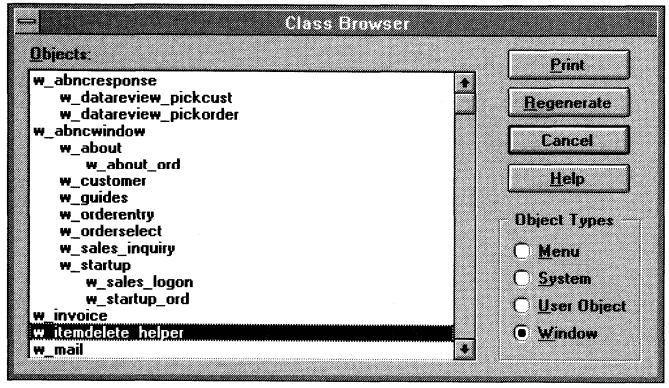
Earlier you learned that the Anchor Bay Nut Company used an application framework to build their Order Entry application. This framework consists of one library named ABNC_COM.PBL, which contains these objects:

abnc_com.pbl	Common library of objects for use in Anchor Bay Nut Company applications	
d_statedddw	7/14/94 15:39:11	(2744) Provides state list for DropDownDataWindows
f_checkrequired	7/14/94 15:39:10	(1337) Checks that all required columns in a DataWindow control are filled in
m_abncmenu	7/14/94 15:37:18	(21668) Ancestor menu for SDI applications at Anchor Bay Nut Company
m_mdimenu	7/14/94 15:37:26	(1578) Ancestor menu for MDI applications at Anchor Bay Nut Company
u_cbclose	7/14/94 15:37:30	(909) User-defined standard Close commandbutton
u_custom_toolbar	7/14/94 15:37:24	(9227) User-defined custom toolbar of picturebuttons
u_dwstandard	7/14/94 15:38:15	(4187) User-defined standard DataWindow control with built-in error processing
w_abncresponse	7/14/94 15:38:01	(4233) Ancestor response window for Anchor Bay Nut Company applications
w_abncwindow	7/14/94 15:37:35	(2611) Ancestor window for SDI applications at Anchor Bay Nut Company
w_about	7/14/94 15:37:37	(10351) Ancestor About window for Anchor Bay Nut Company applications
w_mdiframe	7/14/94 15:37:57	(1728) Ancestor frame for MDI applications at Anchor Bay Nut Company
w_mdisheet	7/14/94 15:37:52	(2340) Ancestor sheet for MDI applications at Anchor Bay Nut Company
w_startup	7/14/94 15:38:41	(11378) Ancestor welcoming/logon window for Anchor Bay Nut Company applications

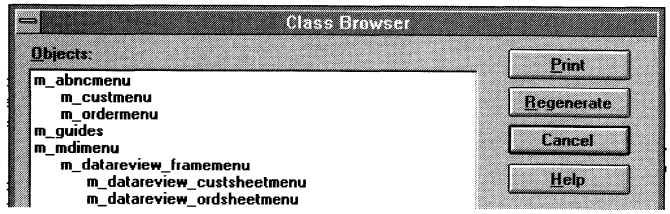
They created most of these objects to serve as base classes.

How they used it While developing objects for their other libraries (ABNC_GDE.PBL and ABNC_ORD.PBL), they inherited those objects from the base classes in ABNC_COM.PBL whenever possible. You can see which objects they inherited from these base classes by displaying the Class Browser in the Library painter.

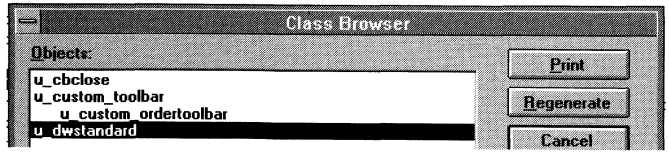
For instance, here you can see all of the *window objects* they inherited from the window base classes w_abncresponse and w_abncwindow:



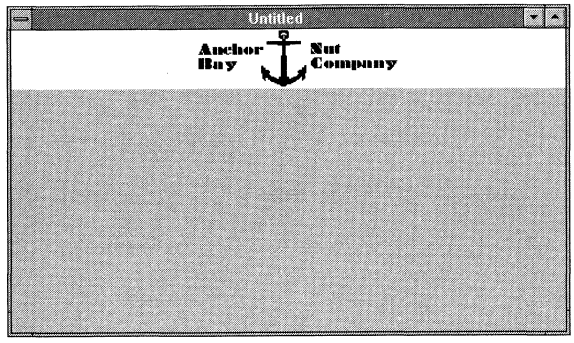
Next, here are the *menu objects* they inherited from the menu base classes m_abncmenu and m_mdimenu:



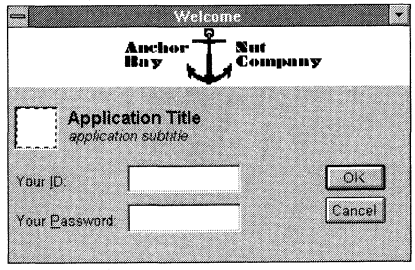
And here you can see that the *user object* named u_custom_ordertoolbar is inherited from the user-object base class u_custom_toolbar:



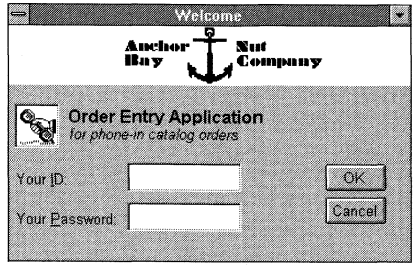
A closer look If you want a better idea of what happens when you define base classes and then inherit objects from them, take a look at the base class *w_abncwindow*. It includes just a couple of controls that display the company's logo:



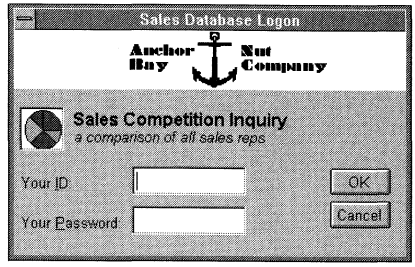
The window *w_startup* is also a base class stored in ABNC_COM.PBL. It is inherited from w_abncwindow but is specialized to include several additional controls (which applications need when prompting users to log on to a server database):



The window *w_startup_ord* is inherited from w_startup and stored in ABNC_ORD.PBL for use in the Order Entry application. As you can see, its controls are even more specialized than those in w_startup, and its scripts are tailored to handle the specific processing required in that application:



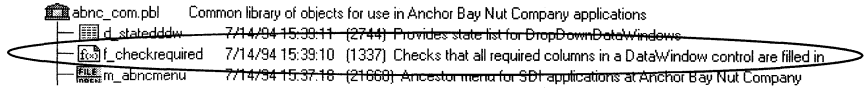
The window *w_sales_logon* is also inherited from w_startup. But it is stored in ABNC_GDE.PBL and tailored for a different use (that is, to enable users to access the company's sales database):



What else it can include

In addition to base classes, an application framework can also include frequently needed, finished objects that you simply use as they are. For instance, you might include a window that displays a clock or calendar. Or maybe a function that performs some specific error processing that many of your applications require.

An example of such a function is f_checkrequired in the Anchor Bay Nut Company's application framework:



How you get one

If you don't already have an appropriate application framework available to use on your project(s), you can either:

- Develop one yourself, *or*
- Acquire one

Developing an application framework This involves using the PowerBuilder painters to create one or more libraries in which you then store the ancestor objects you want to serve as base classes. You can think of these libraries as **class libraries**.

As you develop your ancestor objects, try to include only those features that will be needed in *most* of the descendent objects you'll inherit from them. This will help ensure efficiency. Also, avoid creating ancestor objects that don't really serve some purpose (because you don't want to introduce superfluous levels of inheritance into your applications).

If you're part of a team, you may have a particular person in the role of object manager. This person is typically responsible for overseeing the development and maintenance of an application framework for your organization.

Acquiring an application framework There are a variety of application framework products available commercially for use with PowerBuilder. If you don't have the time or resources to develop one yourself, you might consider one of these. This includes the PowerBuilder Application Library, produced by Powersoft.

For a list of some of the commercial application frameworks that are available, look in your PowerBuilder package or talk to a Powersoft sales representative.

Adding special-purpose libraries

Sometimes you may need to extend your application framework to include one or more special-purpose libraries. These are libraries whose objects can be used in an application to enable it to perform some specialized kind of processing, such as interaction with a particular software product or service.

From Powersoft

For example, some of the special-purpose libraries available from Powersoft are:

Libraries	About them
PowerBuilder Library for NetWare	Included in the PowerBuilder Advanced Developer Toolkit
PowerBuilder Library for Pen Computing	Included in the PowerBuilder Advanced Developer Toolkit
PowerBuilder Library for Lotus Notes	Also provides support for accessing VIM-compliant electronic mail systems
FUNCky for PowerBuilder	Provides hundreds of additional functions you can use in your PowerBuilder applications

From other vendors

Other vendors also offer special-purpose libraries that you can use in your PowerBuilder applications.

For a list of some of the commercial special-purpose libraries that are available, look in your PowerBuilder package or talk to a Powersoft sales representative.

Specifying the logistics of your application

By this point, you've set up most of the foundation for your project (including the hardware/software environment, standards and conventions, databases, libraries, source control, and an application framework). Now it's time to put the last piece of that foundation in place by creating an **application object**. It will enable you to specify the logistics of the particular application you want to build.

About the application object

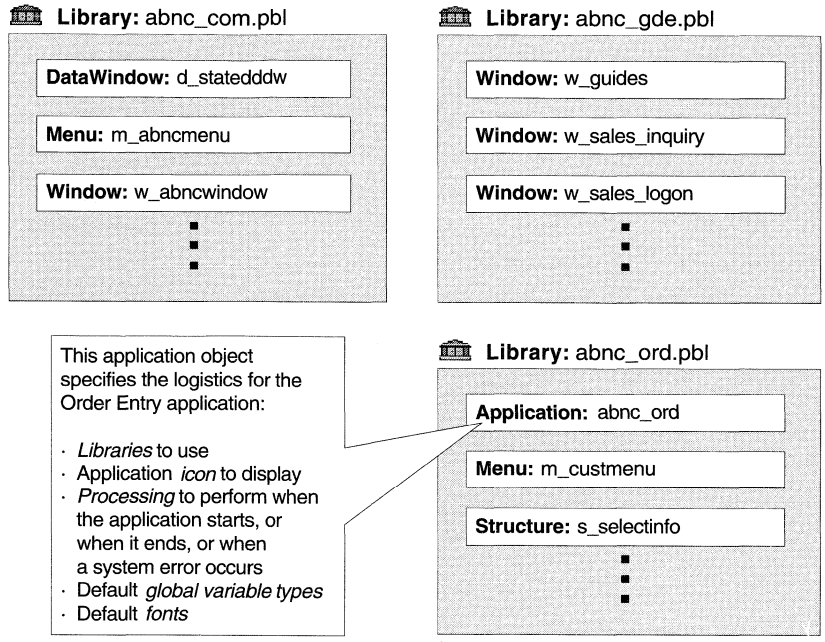
It's in the application object that you define such application-level characteristics and behaviors as these:

- The **list of libraries** (library search path) for your application

  PowerBuilder uses this list to find objects when you're developing the application and when you're executing the application. It searches through your libraries in the order you list them.

- The **icon** that your operating system is to display to represent the executable form of your application
- The **processing** that you want to perform when:
  - A user starts (opens) your application
  - A serious (system) error occurs while a user is in your application
  - A user ends (closes) your application
- The **global variable types** you want to use by default in your application for certain system objects
- The **fonts** you want to use by default in your application for various purposes (including text, data, headings, and labels)

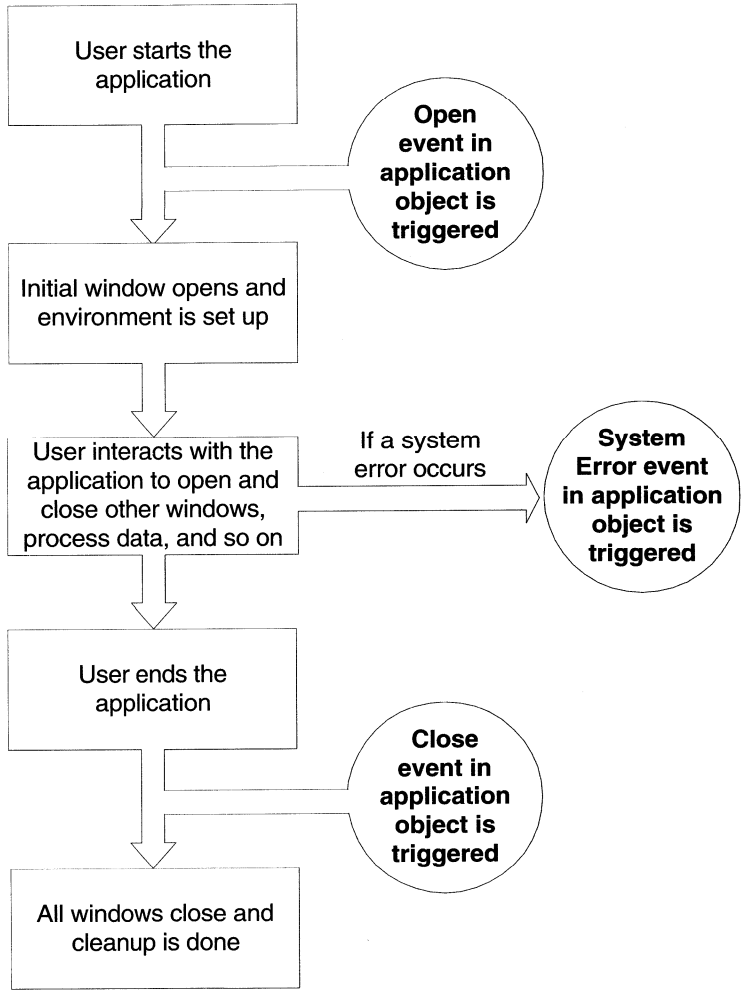
For example, when the developers at the Anchor Bay Nut Company were setting up their Order Entry application, they created an application object named *abnc_ord* to define its characteristics and behaviors. They placed this application object in the library ABNC_ORD.PBL:



More about the processing it performs The role that your application object plays at execution time is particularly important, so it's worth a closer look:

- **Opening your application** The application object provides an Open event that's triggered when a user starts your application. You must write a script for this event to specify the initial processing you want the application to perform (which typically includes opening a window).
- **Trapping system errors** The application object provides a SystemError event that's triggered when a serious application error occurs during execution. By default, PowerBuilder displays a message box (with the appropriate error number and text) in response to such an error. But if you prefer, you can write a script for this event to perform your own error processing.
- **Closing your application** The application object provides a Close event that's triggered when a user ends your application. You should write a script for this event to specify any cleanup processing you want the application to perform (which might include disconnecting from a database or writing to an INI file).

Here's an illustration of how it all works:



Creating an application object

When you're ready to create an application object for your project, you'll use the **Application painter**. It will begin by asking you to specify:

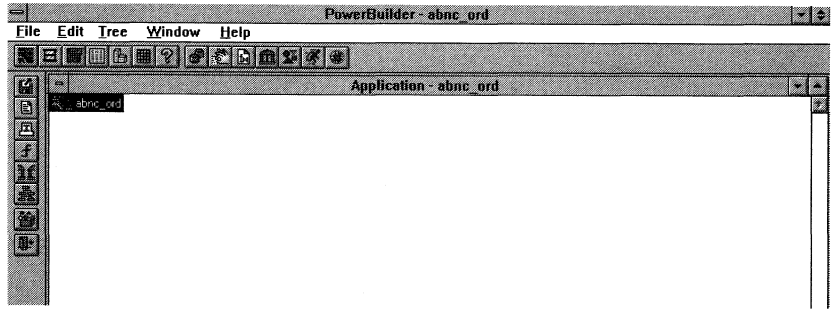
- **A new or existing library** in which you want your application object stored

  For guidance on working with libraries, see "When building a new application" on page 118.

- **A name** for your application object
- **Whether you want an application template** to be generated automatically for your new application

  For guidance on generating an application template, see "Choosing how to set up" on page 81.

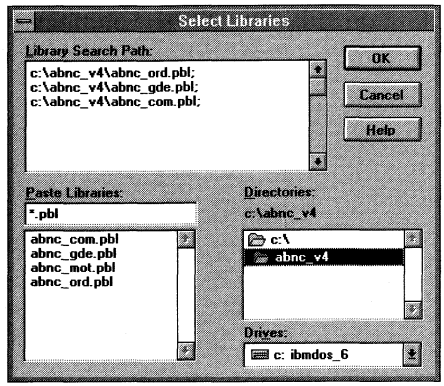
Once you specify this information, the Application painter creates and displays your new application object. For instance:



Defining the details

Now you're ready to define the details of your application object. To get an idea of what's involved here, consider what they specified at the Anchor Bay Nut Company for their abnc_ord application object:

Specifying the list of libraries for the application

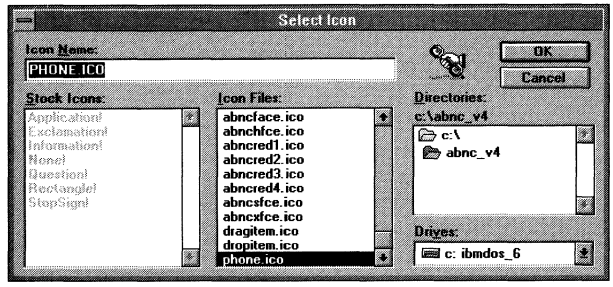


Tips for listing libraries

Make sure you list your libraries in the order you want PowerBuilder to search through them. Put the most frequently accessed libraries at the top of the list. Put class libraries (those containing ancestor objects) at the bottom of the list.

If you need to list libraries in a multiple-tier development environment (such as one including production, QA, and development versions of libraries), see "Library management during a project" on page 125 for advice.

Specifying the icon for the application

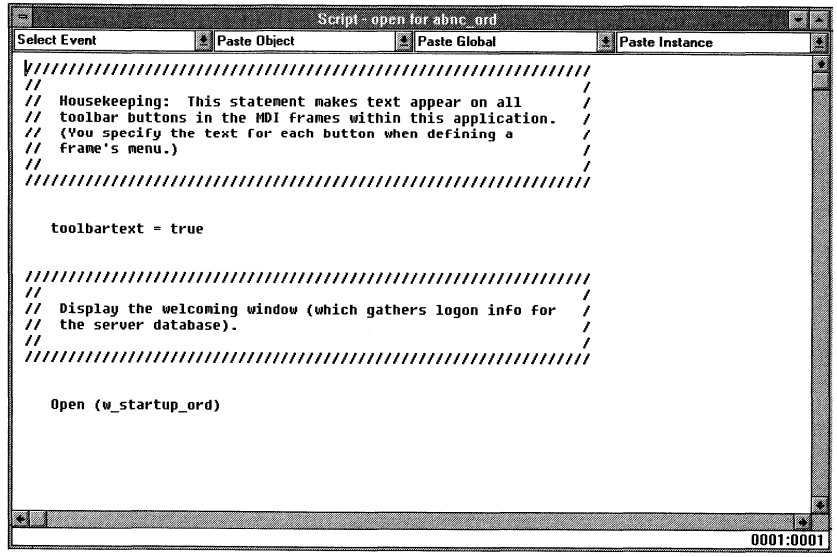


Tip for specifying an icon name

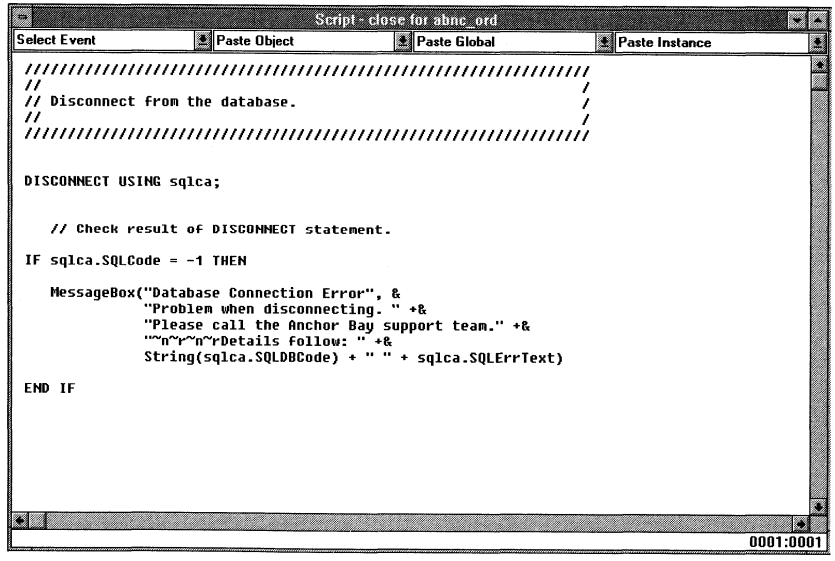
When you specify the icon you want, enter just its filename. Leave out information about where it is stored. This will make your application more transportable.

You should also follow this advice when specifying other kinds of *resources* (such as picture files and pointer files).

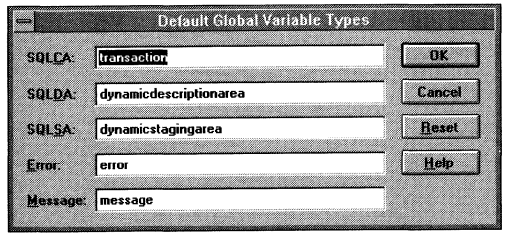
Coding the Open event script for the application



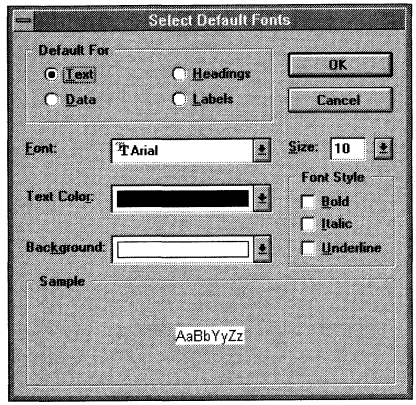
Coding the Close event script for the application



Specifying default global variable types for system objects in the application



Specifying default fonts for use in the application



Coming back to make changes If after setting up your application object you need to change one or more of these details, you can always return to the Application painter and make whatever modifications you want. For instance, you'll probably come back more than once to work on the application object's event scripts. You might also return to alter the library list (if, for instance, you want the application to access additional libraries or search through its libraries in a different order).

Even if you don't need to make any of these changes, you can come back to the Application painter later in your project for another reason: to create the executable version of your application.

To learn about creating the executable version of your application, see Chapter 5, "Deploying an Application." For more information on using the Application painter, see the *User's Guide*.

Where to go from here

Now that you've finished with all of the preliminaries, it's time to start developing the details of your application.

To learn how to get the development phase of your project under way, turn to Chapter 4, "Developing an Application."

CHAPTER 4

Developing an Application

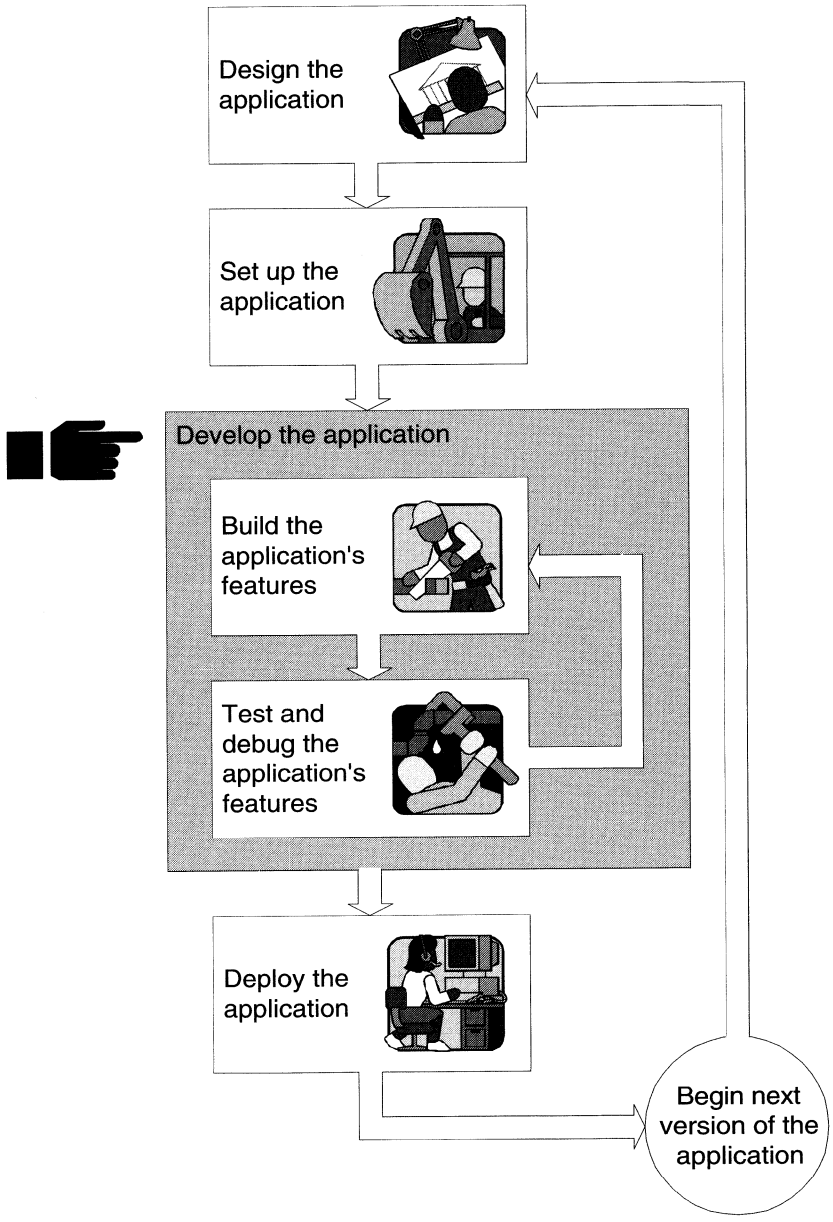
About this chapter

This chapter tells you about the process of developing an application that you've set up.

Contents

Orientation

Here's where you are now in the lifecycle of your application development project:


Design the application
Set up the application
Develop the application
Build the application's features
Test and debug the application's features
Deploy the application
Begin next version of the application

Prototyping the user interface

Now that you've taken care of all the up-front planning and setup for your project, you're ready to start painting objects and coding scripts to develop the features your application requires. But which features should you work on first?

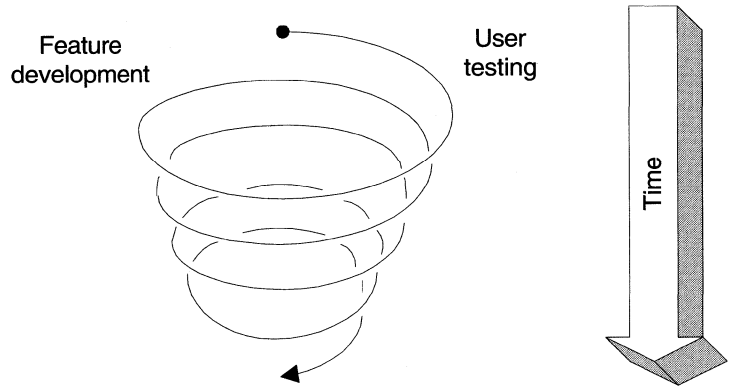
For graphical applications such as those you develop in PowerBuilder, it's usually best to begin by focusing on the user interface. Much of the effort in a typical project is spent trying to get the interface right—making it easy for users to understand and operate, bullet-proofing it to handle user errors and accidents, even ensuring that it's attractive enough to encourage user acceptance. So you'll want to have a *prototype* of it that can be demonstrated and tested as early as possible. That way if changes are necessary, you'll have time to deal with them.

Prototyping the RAD way

PowerBuilder and its painters are designed to support **rapid application development** (RAD), a highly iterative approach to working that focuses on:

- Building up application features quickly,
- Testing them immediately, and
- Responding to comments and new requirements as they arise

One of the key ways you can take advantage of RAD during a project is to use your application as its own working (and evolving) prototype. This involves implementing the basic skeleton of the application's user interface at the start of development and subjecting it to user testing right away. Then as the project proceeds and more features are added, you can periodically return to user testing, ensuring that the application never strays far from what people want:



It's possible to do this because PowerBuilder lets you construct and change your application's user-interface components so swiftly.

How to do it

Getting a new application to the point where it can be used as a prototype is a matter of:

- Creating **the major windows** it is to use
- Painting most of **the important visual features** of these windows—namely, their controls and menus
- Coding **just enough logic** (in scripts) to display these windows and let users navigate among them

Even at this simple level of implementation, users should be able to get a sense of what the application will look like and what it will be like to work with. And as a result, you'll probably be able to gather some valuable feedback that can help you steer your project in the right direction.

For information on developing fundamental user-interface features such as those you want when prototyping, see *Getting Started*. If your application uses MDI (Multiple Document Interface), you should also look at Chapter 6 of *Building Applications*, "Building an MDI Application."

Prototyping even faster—with an application framework

If you use an application framework on your project, you can speed up your prototyping even more. That's because it will enable you to reuse existing objects to develop the features of your user interface, saving you from having to create everything from scratch. It will probably even help you to make your prototype more robust right from the start.

For more information on using an application framework, see Chapter 3, "Setting Up an Application."

Choosing techniques to implement your features

With your prototyping strategy in order, you can turn your attention to implementing the various features that your project plan specifies for this application. The key to success here is to find the best technique for each particular feature, one that meets your requirements for:

- Appearance
- Performance
- Security
- Ease of implementation
- Elegance of design
- Extensibility
- Portability

Helping you find the right technique

To help you do this, the next pages list many of the techniques you'll often want to implement in your applications. You'll see that these techniques are grouped into several different categories according to their purpose:

Looking up technique information For each technique, you'll see one or two kinds of cross-references to places where you can look to get information about it:

These cross-references point you to specific libraries and objects in the sample Order Entry application of the Anchor Bay Nut Company.

Use PowerBuilder to open the sample application and examine those components. Pay particular attention to the comments in the application's scripts, because they contain a great deal of technique information. Then run the application to try out the technique.

> **Installing the sample application**
> You probably already installed the sample Order Entry application when you installed PowerBuilder. But if not, you can do so now.
>
> For more information, see the *Installation and Deployment Guide*.

These cross-references point you to specific chapters in the *Building Applications* manual as well as to other manuals in the PowerBuilder documentation set.

If you need additional information, here are a few general cross-references that you should always keep in mind:

To learn about	Look here
Implementing basic application features	*Getting Started*
Painting the details of any kind of object (such as a window, menu, or DataWindow)	*User's Guide*
Coding a particular piece of syntax (such as a variable declaration or statement)	*PowerScript Language*
Coding a particular built-in function	*Function Reference*
Using the built-in characteristics and behaviors (attributes, events, functions) of a particular object or control	*Objects and Controls*

For the basics

To help you find the techniques you want in this category, they are organized into:

- Housekeeping techniques
- Display techniques
- Database techniques

Housekeeping techniques

To do this	Look here
Access an INI file	 Library: abnc_ord.pbl  Window object: w_startup_ord  CommandButton control: cb_ok  Clicked event script **Notes** Look near the beginning of this script. See Chapter 10, "Using DataWindow Objects"
Clean up when an application ends	 Library: abnc_ord.pbl  Application object: abnc_ord  Close event script **Notes** This event script executes automatically when a user closes the last open window in the application. (You can also explicitly halt the application and cause this event script to execute or not.) See Chapter 3, "Setting Up an Application" See also *Getting Started*
End an application and perform cleanup processing	 Library: abnc_com.pbl  Menu object: m_abncmenu  Menu item: m_file.m_exit  Clicked event script **Notes** You can see how this technique works in two different menu objects that are inherited from m_abncmenu. They are: m_custmenu (used in the w_customer window and stored in abnc_ord.pbl) and m_ordermenu (used in the w_orderentry window and stored in abnc_ord.pbl). See *Getting Started*

To do this	Look here
End an application but skip cleanup processing	Library: abnc_com.pbl Window object: w_startup CommandButton control: cb_cancel Clicked event script **Notes** You can see how this technique works in a window object that is inherited from w_startup. It is: w_startup_ord (in abnc_ord.pbl).
Start an application	Library: abnc_ord.pbl Application object: abnc_ord Open event script **Notes** This event script executes automatically when a user starts the application (by running the executable file, which in this case is ABNC_ORD.EXE). See Chapter 3, "Setting Up an Application" Also see *Getting Started*

Display techniques

To do this	Look here
Close a window	Library: abnc_ord.pbl Window object: w_invoice CommandButton control: cb_cancel Clicked event script Library: abnc_ord.pbl Window object: w_orderentry CommandButton control: cb_close Clicked event script **Notes** This example shows some interesting related processing (checking for unsaved data that might be lost, displaying another window after closing this one). See *Getting Started* Also see the *User's Guide*

To do this	Look here
Display a menu as a popup	Library: abnc_ord.pbl Window object: w_customer RButtonDown event script **Notes** When a user clicks the right mouse button in the w_customer window, this event script displays the m_guide menu item of the menu object m_custmenu (from abnc_ord.pbl) as a popup menu. Library: abnc_ord.pbl Window object: w_orderentry RButtonDown event script **Notes** This example shows more advanced popup menu techniques. It enables the user to display either a long popup menu (the menu object m_orderpopup_full from abnc_ord.pbl) or a shorter version of it (the menu object m_orderpopup_partial from abnc_ord.pbl). See the *User's Guide*
Display a menu on the menu bar	Library: abnc_ord.pbl Window object: w_customer **Notes** Examine the style attributes of this window and you'll see that it's defined to automatically display the menu object m_custmenu (from abnc_ord.pbl) on its menu bar. Library: abnc_gde.pbl Window object: w_sales_inquiry **Notes** Examine the style attributes of this window and you'll see that it's defined to automatically display the menu object m_sales_inquiry (from abnc_gde.pbl) on its menu bar. See *Getting Started* Also see the *User's Guide*

To do this	Look here
Display the initial window for an application	 Library: abnc_ord.pbl  Application object: abnc_ord  Open event script Chapter 3, "Setting Up an Application" Also see *Getting Started*
Open a window	 Library: abnc_ord.pbl  Window object: w_startup_ord  CommandButton control: cb_ok  Clicked event script **Notes** Look at the end of this script.  Library: abnc_com.pbl  Menu object: m_abncmenu ➤ Menu item: m_file.m_reviewdata  Clicked event script **Notes** You can see how this technique works in two different menu objects that are inherited from m_abncmenu. They are: m_custmenu (used in the w_customer window and stored in abnc_ord.pbl) and m_ordermenu (used in the w_orderentry window and stored in abnc_ord.pbl). The window that this script opens is an MDI frame. See *Getting Started* Also see the *User's Guide*

To do this	Look here
Open a window and pass values to it	 Library: abnc_ord.pbl  Window object: w_customer  CommandButton control: cb_orders  Clicked event script **Notes** Look near the end of this script.  Library: abnc_gde.pbl  Window object: w_sales_logon  CommandButton control: cb_ok  Clicked event script **Notes** Look at the end of this script. It shows how you can pass an entire instance of an object (in this case, an instance of the transaction object). See the *User's Guide*

Database techniques

To do this	Look here
Connect to a database	 Library: abnc_ord.pbl  Window object: w_startup_ord  CommandButton control: cb_ok  Clicked event script Chapter 9, "Using Transaction Objects" Chapter 10, "Using DataWindow Objects" Also see *Getting Started*
Connect to a second database	 Library: abnc_gde.pbl  Window object: w_sales_logon  CommandButton control: cb_ok  Clicked event script Chapter 9, "Using Transaction Objects" Chapter 10, "Using DataWindow Objects"

To do this	Look here
Disconnect from a database	 Library: abnc_ord.pbl  Application object: abnc_ord  Close event script Chapter 9, "Using Transaction Objects" Chapter 10, "Using DataWindow Objects" Also see *Getting Started*
Disconnect from a second database	 Library: abnc_gde.pbl  Window object: w_sales_inquiry  Close event script Chapter 9, "Using Transaction Objects" Chapter 10, "Using DataWindow Objects"
Prompt for server logon information	 Library: abnc_ord.pbl  Window object: w_startup_ord **Notes** The application displays this as its initial window (by opening it in the Open event script of the application object abnc_ord). Examine the Clicked event scripts of the two CommandButton controls in this window—that's where all of the processing is done.  Library: abnc_gde.pbl  Window object: w_sales_logon **Notes** This is what the application displays when a user wants to log on to a second database (specifically, the sales database). Compare the Clicked event scripts of this window's two CommandButton controls with those in the w_startup_ord window.

For presenting the user interface

To help you find the techniques you want in this category, they are organized into:

- Basic window techniques
- MDI techniques
- Control and menu techniques
- Other handy techniques

Basic window techniques

To do this	Look here
Display a dialog box	 Library: abnc_ord.pbl  Window object: w_orderentry CommandButton control: cb_retrieve  Clicked event script **Notes** Look at the end of this script. This script displays the response window object w_orderselect (from abnc_ord.pbl), which serves as a dialog box in which the user selects an order to retrieve.  Library: abnc_gde.pbl  Window object: w_sales_inquiry  CommandButton control: cb_graphtype  Clicked event script **Notes** This script displays the response window object w_sales_graphtype (from abnc_gde.pbl), which serves as a dialog box in which the user selects the type of graph to plot.

(continued)

To do this	Look here
Display a dialog box *(continued)*	Library: abnc_ord.pbl Menu object: m_datareview_framemenu ➤ Menu item: m_file.m_open.m_custopen Clicked event script **Notes** Here's an example of displaying a dialog box when you're using an MDI frame. This script displays the response window object w_datareview_pickcust (from abnc_ord.pbl), which serves as a dialog box in which the user selects the customer to retrieve. See the *User's Guide*
Display a message box	Library: abnc_ord.pbl Window object: w_customer User-defined window function: wf_warndataloss Library: abnc_ord.pbl Window object: w_customer CommandButton control: cb_delete Clicked event script **Notes** This script displays several different message boxes, some prompting for confirmation, others simply providing information. See *Getting Started* Also see the *User's Guide*
Display an about box	Library: abnc_ord.pbl Menu object: m_custmenu ➤ Menu item: m_help.m_about Clicked event script **Notes** This script displays the response window object w_about_ord (from abnc_ord.pbl), which presents information about the Order Entry application.

To do this	Look here
Hide a window	Library: abnc_ord.pbl Window object: w_customer CommandButton control: cb_orders Clicked event script **Notes** Look at the end of this script and you'll see that it contains code to the hide (make invisible) the w_customer window.
Open multiple instances of a window	Library: abnc_ord.pbl Window object: w_datareview_pickorder CommandButton control: cb_ok Clicked event script **Notes** This script shows how you can use a throw-away variable to open multiple instances of a window. Library: abnc_mot.pbl Window object: w_mdiframe_fun Uevent_createsheets event script **Notes** The script of this user event works with an array to open and manage multiple instances of a window. The advantage of using an array is that you can later refer to a particular instance (by identifying its element number from the array). See the *User's Guide*
Show a hidden window	Library: abnc_ord.pbl Window object: w_orderentry Close event script **Notes** This script shows (makes visible) the window w_customer (which was previously hidden). The w_customer window object is stored in abnc_ord.pbl.

To do this	Look here
Use child windows	 Library: abnc_ord.pbl  Window object: w_toolbar **Notes** This example of a child window is used to implement a custom floating toolbar in an SDI application.  Library: abnc_ord.pbl  Window object: w_itemdelete_helper **Notes** This example of a child window is used to implement an animated wizard (which in this case helps a user delete an order item). See the *User's Guide*
Use main windows	 Library: abnc_ord.pbl  Window object: w_customer **Notes** This example shows a typical use of a main window in an SDI application.  Library: abnc_ord.pbl  Window object: w_datareview_custsheet **Notes** This example shows how you implement sheets in an MDI application by defining them as main windows. Chapter 6, "Building an MDI Application" Also see the *User's Guide* and *Getting Started*
Use popup windows	See the *User's Guide*

To do this	Look here
Use response windows	 Library: abnc_ord.pbl  Window object: w_about_ord **Notes** This example of a response window is used to implement an about box (which displays information about an application).  Library: abnc_ord.pbl  Window object: w_datareview_pickcust **Notes** This example of a response window prompts the user to select a customer to retrieve and display. See the *User's Guide*

MDI techniques

To do this	Look here
Close an MDI frame	 Library: abnc_mot.pbl  Menu object: m_mdi_funmenu ➤ Menu item: m_file.m_exit  Clicked event script  Library: abnc_ord.pbl  Menu object: m_datareview_framemenu ➤ Menu item: m_file.m_exit  Clicked event script **Notes** Closing the MDI frame automatically closes all of its open sheets too. Chapter 6, "Building an MDI Application"

To do this	Look here
Close every open MDI sheet	Library: abnc_mot.pbl Menu object: m_mdi_funmenu ➤ Menu item: m_file.m_closeeverysheet Clicked event script Library: abnc_ord.pbl Menu object: m_datareview_framemenu ➤ Menu item: m_file.m_closeall Clicked event script **Notes** This example uses a global variable populated by the sheet windows (window object w_datareview_custsheet and window object w_datareview_ordsheet in abnc_ord.pbl) in their event scripts to keep track of which sheet is active. Chapter 6, "Building an MDI Application"
Close the active MDI sheet	Library: abnc_mot.pbl Menu object: m_mdi_funmenu ➤ Menu item: m_file.m_closeactivesheet Clicked event script Library: abnc_ord.pbl Menu object: m_datareview_framemenu ➤ Menu item: m_file.m_close Clicked event script **Notes** This example uses a global variable populated by the sheet windows (window object w_datareview_custsheet and window object w_datareview_ordsheet in abnc_ord.pbl) in their event scripts to keep track of which sheet is active. Chapter 6, "Building an MDI Application"

To do this	Look here
Display an MDI toolbar	Library: abnc_ord.pbl Menu object: m_datareview_framemenu Menu item: m_window Selected event script Menu item: m_window.m_showtoolbar Clicked event script **Notes** These scripts are used to show or hide the toolbar for the frame or sheet menu that is currently displayed. Library: abnc_ord.pbl Application object: abnc_ord Open event script **Notes** This script contains code to make all MDI toolbars in the Order Entry application display text on their buttons by default. Chapter 6, "Building an MDI Application"
Display different menu bars for an MDI frame and its sheets	Library: abnc_ord.pbl Window object: w_datareview_frame Window object: w_datareview_custsheet Window object: w_datareview_ordsheet **Notes** Look at the style attributes for each of these windows and you'll see that: *w_datareview_frame* is defined to display the menu object m_datareview_framemenu (from abnc_ord.pbl) automatically on its menu bar. *w_datareview_custsheet* is defined to display the menu object m_datareview_custsheetmenu (from abnc_ord.pbl) automatically on its menu bar. *w_datareview_ordsheet* is defined to display the menu object m_datareview_ordsheetmenu (from abnc_ord.pbl) automatically on its menu bar. Chapter 6, "Building an MDI Application"

To do this	Look here
Manipulate MDI sheets	Library: abnc_ord.pbl Menu object: m_datareview_framemenu ➤ Menu item: m_window ➤ Menu item: m_tile Clicked event script ➤ Menu item: m_layer Clicked event script ➤ Menu item: m_cascade Clicked event script ➤ Menu item: m_arrangeicons Clicked event script **Notes** The first three scripts enable a user to tile, layer, and cascade the open sheets in an MDI frame. The last script lets a user arrange the icons of any minimized sheets in the frame. Chapter 6, "Building an MDI Application"
Position an MDI toolbar	Library: abnc_ord.pbl Menu object: m_datareview_framemenu ➤ Menu item: m_window Selected event script ➤ Menu item: m_window.m_floattoolbar Clicked event script Menu object: m_datareview_custsheetmenu ➤ Menu item: m_window.m_floattoolbar Clicked event script Menu object: m_datareview_ordsheetmenu ➤ Menu item: m_window.m_floattoolbar Clicked event script **Notes** These scripts are used to float or fix the position of the toolbar for the frame or sheet menu that is currently displayed. Notice that the sheet menus have their own specialized versions of the Clicked event script for the m_floattoolbar menu item. *(continued)*

To do this	Look here
Position an MDI toolbar *(continued)*	Library: abnc_ord.pbl Window object: w_datareview_frame Open event script **Notes** The code in this script sets the frame's toolbar to be right-aligned by default. Chapter 6, "Building an MDI Application"
Use MDI frames (with MicroHelp)	Library: abnc_ord.pbl Window object: w_datareview_frame **Notes** This example shows a typical use of a frame window in an MDI application. Chapter 6, "Building an MDI Application"

Control and menu techniques

To do this	Look here
Disable controls or menu items	Library: abnc_ord.pbl Window object: w_customer CommandButton control: cb_delete Clicked event script **Notes** Look near the end of this script. When you paint a particular control or menu item, you can specify whether it is to be enabled initially (when the window opens). Examine the controls in the w_customer window and the menu items in its menu (m_custmenu) to see how they are defined. See the *User's Guide*

To do this	Look here
Display a toolbar in an SDI window	Library: abnc_ord.pbl Window object: w_customer UserObject control: uo_toolbar **Notes** This UserObject control is inherited from the user object u_custom_toolbar (stored in abnc_com.pbl). The Open event script of the w_customer window calls a function that was defined in this user object (to pass needed information to the UserObject control). Library: abnc_ord.pbl Window object: w_orderentry Open event script **Notes** Here's a more sophisticated approach. This script opens a child window object named w_toolbar (stored in abnc_ord.pbl), which contains a UserObject control to display the user object u_custom_ordertoolbar (inherited from u_custom_toolbar and stored in abnc_ord.pbl). That child window enables a user to float the toolbar around on the w_orderentry window.
Enable controls or menu items	Library: abnc_ord.pbl Window object: w_customer CommandButton control: cb_retrieve Clicked event script **Notes** Look near the end of this script. When you paint a particular control or menu item, you can specify whether it is to be enabled initially (when the window opens). Examine the controls in the w_customer window and the menu items in its menu (m_custmenu) to see how they are defined. See the *User's Guide*

To do this	Look here
Hide a control in a window	 Library: abnc_ord.pbl  Window object: w_customer  CommandButton control: cb_delete  Clicked event script **Notes** Look near the end of this script and you'll see that it contains code to the hide (make invisible) the dw_list DataWindow control. See the *User's Guide*
Show a hidden control in a window	 Library: abnc_ord.pbl  Window object: w_customer  CommandButton control: cb_retrieve  Clicked event script **Notes** Look near the end of this script and you'll see that it contains code to the show (make visible) the dw_list DataWindow control. See the *User's Guide*
Trap control menu commands issued by the user	 Library: abnc_ord.pbl  Window object: w_orderselect  Uevent_ctrlmenu_close event script
Trap particular keystrokes typed by the user	 Library: abnc_ord.pbl  Window object: w_customer  DataWindow control: dw_list  Uevent_keypressed event script
Use CheckBox controls	See the *User's Guide*

To do this	Look here
Use CommandButton controls	Library: abnc_ord.pbl Window object: w_orderentry CommandButton control: cb_save CommandButton control: cb_delete_order CommandButton control: cb_retrieve CommandButton control: cb_new_order CommandButton control: cb_close **Notes** Examine the Clicked event scripts of these CommandButton controls to see how they are used. See the *User's Guide*
Use DataWindow controls	Library: abnc_ord.pbl Window object: w_customer DataWindow control: dw_list DataWindow control: dw_detail **Notes** In this example, dw_list is used for selecting a row to display in dw_detail. Then dw_detail can be used to perform maintenance operations on that row. Library: abnc_ord.pbl Window object: w_orderentry DataWindow control: dw_ordmaster DataWindow control: dw_orddetail **Notes** These two DataWindow controls are used to let users display and maintain master and detail data (for a particular order and its order items). Library: abnc_ord.pbl Window object: w_invoice DataWindow control: dw_invoice **Notes** This DataWindow control is used to preview a report before printing it. Chapter 10, "Using DataWindow Objects" Also see *Getting Started*

To do this	Look here
Use DropDownListBox controls	See the *User's Guide*
Use EditMask controls	See the *User's Guide*
Use Graph controls	See the *User's Guide*
Use GroupBox controls	Library: abnc_gde.pbl Window object: w_sales_graphtype GroupBox control: gb_graphtype GroupBox control: gb_graphstyle **Notes** Each of these GroupBox controls is used to group together a few particular RadioButton controls. See the *User's Guide*
Use HScrollBar controls	See the *User's Guide*
Use Line controls (drawing objects)	Library: abnc_ord.pbl Window object: w_customer Line control: ln_divider **Notes** This Line control is used for cosmetic purposes only. It simply divides the window visually to make it easier to understand. See the *User's Guide*
Use ListBox controls	See the *User's Guide*
Use MultiLineEdit controls	Library: abnc_ord.pbl Window object: w_mail MultiLineEdit control: mle_to MultiLineEdit control: mle_text **Notes** These MultiLineEdit controls are used to address and compose an e-mail message. To learn how, explore the user-defined functions of the w_mail window as well as the Clicked event scripts of the window's CommandButton controls. See the *User's Guide*
Use OLE 2.0 controls	Chapter 15, "Using OLE in an Application"

To do this	Look here
Use Oval controls (drawing objects)	See the *User's Guide*
Use Picture controls	Library: abnc_ord.pbl Window object: w_startup_ord Picture control: p_image **Notes** This Picture control is used for cosmetic purposes only. Library: abnc_ord.pbl Window object: w_orderentry Picture control: p_trashcan **Notes** This Picture control is used as a trashcan into which users can drag and drop rows they want to delete (from the DataWindow control dw_orddetail). Examine the event scripts of p_trashcan to learn how it works. See the *User's Guide*
Use PictureButton controls	Library: abnc_ord.pbl Window object: w_orderentry PictureButton control: pb_new_item **Notes** Examine the Clicked event script of this PictureButton control to see how it is used. Library: abnc_gde.pbl Window object: w_sales_graphrotate PictureButton control: pb_up PictureButton control: pb_down PictureButton control: pb_left PictureButton control: pb_right **Notes** These PictureButton controls are used to rotate a graph. For details on how they work, look at their Clicked event scripts. See *Getting Started* Also see the *User's Guide*

To do this	Look here
Use RadioButton controls	Library: abnc_gde.pbl Window object: w_sales_graphtype RadioButton control: rb_area RadioButton control: rb_column RadioButton control: rb_line RadioButton control: rb_2d RadioButton control: rb_3d RadioButton control: rb_solid **Notes** To see how these RadioButton controls are used, look at the Clicked event script of the window's cb_ok CommandButton control. Also, notice how these RadioButton controls are organized within two GroupBox controls. See the *User's Guide*
Use Rectangle controls (drawing objects)	Library: abnc_com.pbl Window object: w_abncwindow Rectangle control: r_1 **Notes** This Rectangle control is used for cosmetic purposes only. See the *User's Guide*
Use RoundRectangle controls (drawing objects)	See the *User's Guide*
Use SingleLineEdit controls	Library: abnc_ord.pbl Window object: w_startup_ord SingleLineEdit control: sle_logonid SingleLineEdit control: sle_logonpass **Notes** These SingleLineEdit controls enable the user to enter an ID and password for logging on to a server database (if appropriate). Look at the Clicked event script of the window's cb_ok CommandButton control to see how they are handled. *(continued)*

To do this	Look here
Use SingleLineEdit controls *(continued)*	Library: abnc_ord.pbl Window object: w_customer SingleLineEdit control: sle_lname **Notes** This SingleLineEdit control is used to get selection criteria for a retrieval operation. If the user types a name in sle_lname, the application uses it (in the Clicked event script of the cb_retrieve CommandButton control) to select customers to display in the dw_list DataWindow control. See the *User's Guide*
Use StaticText controls	Library: abnc_ord.pbl Window object: w_startup_ord StaticText control: st_apptitle StaticText control: st_appsubtitle StaticText control: st_id StaticText control: st_password **Notes** These StaticText controls are used to display constant information to the user and to label other controls in the window. Library: abnc_ord.pbl Window object: w_startup_ord StaticText control: st_message **Notes** This StaticText control is used to dynamically display a message to the user (to indicate that the application is trying to connect to a database). To see how this is handled, examine the Clicked event script of the window's cb_ok CommandButton control. See the *User's Guide*

To do this	Look here
Use UserObject controls	Library: abnc_ord.pbl Window object: w_customer UserObject control: uo_toolbar **Notes** This UserObject control is inherited from the user object u_custom_toolbar (stored in abnc_com.pbl). The Open event script of the w_customer window calls a function that was defined in this user object (to pass needed information to the UserObject control). See the *User's Guide*
Use VScrollBar controls	See the *User's Guide*

Other handy techniques

To do this	Look here
Display online Help	Library: abnc_ord.pbl Menu object: m_custmenu Menu item: m_help.m_contents Clicked event script **Notes** This example takes a user to the index of the application's Help file. Library: abnc_ord.pbl Window object: w_orderselect CommandButton control: cb_context_help Clicked event script **Notes** Here's how you can implement context-sensitive Help in a dialog box (to take the user to an appropriate topic in the application's Help file). Chapter 8, "Providing Online Help for an Application"

To do this	Look here
Drag and drop in a window	Library: abnc_ord.pbl Window object: w_orderentry DataWindow control: dw_orddetail DoubleClicked event script Picture control: p_trashcan DragDrop event script DragEnter event script DragLeave event script **Notes** In this example, a user can delete a row in the dw_orddetail DataWindow control by: double-clicking it, dragging it to the trashcan (the p_trashcan Picture control), and then dropping it there. Chapter 7, "Using Drag and Drop in a Window"

For accessing data

To help you find the techniques you want in this category, they are organized into:

- Retrieval techniques
- Update techniques
- Everyday DataWindow techniques
- Dynamic DataWindow techniques
- Error and validation techniques
- Other handy techniques

Retrieval techniques

To do this	Look here
Prompt the user to enter selection criteria for a retrieval	 Library: abnc_ord.pbl  Window object: w_customer  SingleLineEdit control: sle_lname **Notes** Here's an example of a manual way to get selection criteria. If the user types a name in sle_lname, the application uses it (in the Clicked event script of the cb_retrieve CommandButton control) to select customers to display in the dw_list DataWindow control.  Library: abnc_gde.pbl  Window object: w_sales_inquiry  CommandButton control: cb_graphdata  Clicked event script **Notes** Now here's a more automated approach that uses the Prompt for Criteria feature you can specify in the definition of a DataWindow object (in this case d_sales_graphdata, which is stored in abnc_gde.pbl). The cb_graphdata CommandButton retrieves rows for the dw_sales_graphdata DataWindow control, which uses d_sales_graphdata. During this retrieval operation, the Prompt for Criteria feature causes it to display a dialog box in which the user must enter any selection criteria. See the *User's Guide*

To do this	Look here
Query by example (via query mode) in a DataWindow	Chapter 11, "Using Dynamic DataWindow Objects"
Retrieve from a database table	Library: abnc_ord.pbl Window object: w_datareview_pickcust Open event script **Notes** Here's an example of a retrieval that uses a DataWindow. Look near the beginning of the Open event script for w_datareview_pickcust, and you'll see code that retrieves all rows from the Customer table and displays them in the dw_choice DataWindow control (which uses the d_custall DataWindow object from abnc_ord.pbl). Library: abnc_ord.pbl Window object: w_customer User-defined window function: wf_newcustnum **Notes** This retrieval example uses embedded SQL (to retrieve some specific data from the Customer table). Chapter 10, "Using DataWindow Objects" Also see *Getting Started* (for more on DataWindows) and *PowerScript Language* (for information on using embedded SQL)
Retrieve from multiple database tables	Library: abnc_ord.pbl Window object: w_datareview_custsheet Open event script **Notes** Look near the beginning of this script, and you'll see that it retrieves a particular row from the Customer table into the dw_custdetail DataWindow control (which uses the d_custdetail DataWindow object from abnc_ord.pbl). But the d_custdetail DataWindow object includes a DropDownDataWindow that also automatically retrieves all rows from the State table. *(continued)*

To do this	Look here
Retrieve from multiple database tables *(continued)*	Library: abnc_ord.pbl Window object: w_datareview_ordsheet Open event script **Notes** If you look near the beginning of this script, you'll see that it retrieves information about a particular order into the dw_ordsummary DataWindow control (which uses a DataWindow object named d_ordsummary from abnc_ord.pbl). The d_ordsummary DataWindow object is defined to join and retrieve rows from four different tables: Order_header, Order_item, Product, and Customer. Chapter 10, "Using DataWindow Objects" Also see *PowerScript Language* (for information on using embedded SQL)

Update techniques

To do this	Look here
Delete a single row from a DataWindow control	Library: abnc_ord.pbl Window object: w_customer CommandButton control: cb_delete Clicked event script **Notes** This script deletes the current row from the dw_detail DataWindow control and applies this update to a single table (Customer). Chapter 10, "Using DataWindow Objects" Also see *Getting Started*

To do this	Look here
Delete multiple rows from DataWindow controls	 Library: abnc_ord.pbl  Window object: w_orderentry  Picture control: p_trashcan  DragDrop event script  CommandButton control: cb_save  Clicked event script **Notes** This example illustrates a two-step technique for deleting one or more rows: *First*, a user can drag each unwanted row from the dw_orddetail DataWindow control and drop it in p_trashcan. *Then*, the user can apply all of these deletions to the database by clicking cb_save. This updates two different tables: Order_header and Order_item.  Library: abnc_ord.pbl  Window object: w_orderentry  CommandButton control: cb_delete_order  Clicked event script **Notes** This script deletes *all* of the rows from the dw_orddetail DataWindow control and applies those deletions to the Order_item table. Then it deletes the current row from the dw_ordmaster DataWindow control and applies that deletion to the Order_header table. Chapter 10, "Using DataWindow Objects"
Delete rows with embedded SQL	See *PowerScript Language*

To do this	Look here
Insert rows in a DataWindow control	 Library: abnc_ord.pbl  Window object: w_customer  CommandButton control: cb_new  Clicked event script **Notes** If you look near the end of this script, you'll see code that inserts a new row in the dw_detail DataWindow control (which is designed to hold just one row).  Library: abnc_ord.pbl  Window object: w_orderentry  PictureButton control: pb_new_item  Clicked event script **Notes** This script inserts a new row in the dw_orddetail DataWindow control (which is designed to hold multiple rows). See Chapter 10, "Using DataWindow Objects" Also see *Getting Started*
Insert rows with embedded SQL	See *PowerScript Language*
Update a database table	 Library: abnc_ord.pbl  Window object: w_customer  CommandButton control: cb_save  Clicked event script **Notes** Look in the middle of this script and you'll see code that updates the Customer table from the dw_detail DataWindow control (which uses the d_custdetail DataWindow object). See Chapter 10, "Using DataWindow Objects" Also see *Getting Started* (for more on DataWindows) and *PowerScript Language* (for information on using embedded SQL)

To do this	Look here
Update multiple database tables	Library: abnc_ord.pbl Window object: w_orderentry CommandButton control: cb_save Clicked event script **Notes** Look in the middle of this script and you'll see code that updates two different tables: The *Order_header table* is updated from the dw_ordmaster DataWindow control (which uses the d_ordmaster DataWindow object). The *Order_item table* is updated from the dw_orddetail DataWindow control (which uses the d_orddetail DataWindow object). Chapter 10, "Using DataWindow Objects" Also see *PowerScript Language* (for information on using embedded SQL)

Everyday DataWindow techniques

To do this	Look here
Accept the user's last entry when leaving a DataWindow control	Library: abnc_ord.pbl Window object: w_customer CommandButton control: cb_orders Clicked event script **Notes** Look at the beginning of this script. It shows the technique of coding the AcceptText function in a control that a user might jump to after typing in a DataWindow control. *(continued)*

To do this	Look here
Accept the user's last entry when leaving a DataWindow control *(continued)*	Library: abnc_com.pbl User object: u_dwstandard LoseFocus event script Uevent_accepttext event script **Notes** The user object u_dwstandard is a custom DataWindow control that you could define to standardize your error checking. It includes code (in the two event scripts listed above) to call the AcceptText function itself (instead of waiting for other controls to call it). To see how this technique is used, look at the dw_orddetail DataWindow control in the w_orderentry window (stored in abnc_ord.pbl). It is inherited from u_dwstandard. See Chapter 10, "Using DataWindow Objects"
Cross-tabulate data	Library: abnc_gde.pbl Window object: w_sales_inquiry DataWindow control: dw_sales_teamcrosstab DataWindow control: dw_sales_repcrosstab **Notes** The w_sales_inquiry window retrieves the data for these two crosstab DataWindow controls in its Open event script. The dw_sales_teamcrosstab DataWindow control uses the d_sales_teamcrosstab DataWindow object. The dw_sales_repcrosstab DataWindow control uses the d_sales_repcrosstab DataWindow object. (Both of these objects are stored in abnc_gde.pbl.) See the *User's Guide*
Display master and detail DataWindow controls in a window	Library: abnc_ord.pbl Window object: w_orderentry DataWindow control: dw_ordmaster DataWindow control: dw_orddetail

To do this	Look here
Display selection and maintenance DataWindow controls in a window	Library: abnc_ord.pbl Window object: w_customer DataWindow control: dw_list DataWindow control: dw_detail **Notes** In this example, dw_list is used for selecting the row to display in dw_detail. Then dw_detail can be used to perform maintenance operations on that row. See *Getting Started*
Graph data	Library: abnc_gde.pbl Window object: w_sales_inquiry DataWindow control: dw_sales_repgraph **Notes** The dw_sales_repgraph DataWindow control gets its data by sharing the data from the dw_sales_graphdata DataWindow control. Look near the end of the Open event script for the w_sales_inquiry window to see how data is retrieved for dw_sales_graphdata and then shared. See the *User's Guide*
Indicate to the user which row in a DataWindow control is the current one	Library: abnc_ord.pbl Window object: w_datareview_pickcust Open event script **Notes** Look near the end of this script and you'll see code that displays a standard indicator (a *hand*) to let the user know which row in the dw_choice DataWindow control is current. *(continued)*

To do this	Look here
Indicate to the user which row in a DataWindow control is the current one *(continued)*	Library: abnc_ord.pbl Window object: w_orderentry DataWindow control: dw_orddetail Constructor event script **Notes** This script shows how you can display a picture of your own as a *custom row indicator*. Library: abnc_ord.pbl Window object: w_datareview_pickorder DataWindow control: dw_choice RowFocusChanged event script **Notes** This script uses the technique of *highlighting* the current row to indicate it to the user. See *Getting Started*
Preview a report	Library: abnc_ord.pbl Window object: w_invoice Open event script Chapter 10, "Using DataWindow Objects"
Search through a DropDownDataWindow for a particular value	Library: abnc_ord.pbl Window object: w_customer DataWindow control: dw_detail ItemChanged event script **Notes** Look near the beginning of this script.
Share data between two DataWindow controls	Library: abnc_gde.pbl Window object: w_sales_inquiry CommandButton control: cb_graphdata Clicked event script Chapter 10, "Using DataWindow Objects"

To do this	Look here
Switch the DataWindow in a DataWindow control dynamically	Library: abnc_ord.pbl Window object: w_customer CommandButton control: cb_retrieve Clicked event script **Notes** Look near the beginning of this script. Library: abnc_gde.pbl Window object: w_sales_graphtopic CommandButton control: cb_ok Clicked event script **Notes** This script changes the DataWindow displayed in the w_sales_inquiry window by the dw_sales_repgraph DataWindow control. (You'll find w_sales_inquiry in abnc_gde.pbl.) Chapter 10, "Using DataWindow Objects"

Dynamic DataWindow techniques

To do this	Look here
Create a DataWindow object dynamically during execution	Chapter 11, "Using Dynamic DataWindow Objects"
Dynamically change the appearance of a DataWindow	Library: abnc_ord.pbl Window object: w_orderselect DataWindow control: dw_choices RetrieveEnd event script Library: abnc_gde.pbl Window object: w_sales_extract DataWindow control: dw_review_extract **Notes** This DataWindow control uses the d_sales_extract DataWindow object. That object includes several attribute conditional expressions to modify attributes of its columns dynamically at execution time. Chapter 11, "Using Dynamic DataWindow Objects" Also see the *User's Guide*

To do this	Look here
Dynamically change the appearance of a graph	Library: abnc_gde.pbl Window object: w_sales_inquiry DataWindow control: dw_sales_repgraph Clicked event script See the *User's Guide*
Dynamically change the rotation of a graph	Library: abnc_gde.pbl Window object: w_sales_graphrotate PictureButton control: pb_up Clicked event script PictureButton control: pb_down Clicked event script PictureButton control: pb_left Clicked event script PictureButton control: pb_right Clicked event script **Notes** These scripts contain code to rotate the graph displayed in the dw_sales_repgraph DataWindow control of the w_sales_inquiry window (from abnc_gde.pbl). The w_sales_graphrotate window is opened from the w_sales_graphtopic window (which itself is opened when the user clicks the cb_graphtopic CommandButton control in the w_sales_inquiry window). See the *User's Guide*
Dynamically change the type of a graph	Library: abnc_gde.pbl Window object: w_sales_graphtype CommandButton control: cb_ok Clicked event script **Notes** This script changes the type of graph displayed in the w_sales_inquiry window by the dw_sales_repgraph DataWindow control. (You'll find w_sales_inquiry in abnc_gde.pbl.) See the *User's Guide*

To do this	Look here
Dynamically change the WHERE clause of a DataWindow	Library: abnc_gde.pbl Window object: w_guides Open event script **Notes** Look near the beginning of this script. To see the code used to open the w_guides window, look at the Clicked event scripts for the following two menu items in the menu object m_abncmenu (from abnc_com.pbl): m_guide.m_phoneprocedures m_guide.m_companypolicies Chapter 11, "Using Dynamic DataWindow Objects"
Zoom in or out with a DataWindow control	Library: abnc_ord.pbl Window object: w_invoice Open event script CommandButton control: cb_zoomin Clicked event script CommandButton control: cb_zoomout Clicked event script **Notes** This example shows how you can set up zooming in a DataWindow control (dw_invoice) that displays a report. The Open event script of w_invoice contains code to set the initial zoom factor. The two CommandButton controls let the user zoom in or zoom out from there.

Error and validation techniques

To do this	Look here
Check for referential integrity when updating database tables	Library: abnc_ord.pbl Window object: w_customer CommandButton control: cb_delete Clicked event script **Notes** Look near the beginning of this script. *(continued)*

To do this	Look here
Check for referential integrity when updating database tables *(continued)*	Library: abnc_ord.pbl Window object: w_orderentry CommandButton control: cb_save Clicked event script CommandButton control: cb_delete_order Clicked event script **Notes** Look in the middle of each of these scripts and you'll see the code used to ensure referential integrity when updating multiple tables at the same time.
Check for required values in a DataWindow control	Library: abnc_ord.pbl Window object: w_customer CommandButton control: cb_save Clicked event script **Notes** Look at the very beginning of this script and you'll see that it calls the function object f_checkrequired (from abnc_com.pbl) to do the appropriate checking.
Standardize database error processing	Library: abnc_com.pbl User object: u_dwstandard DBError event script LoseFocus event script Uevent_accepttext event script Uevent_dberr_reqmissing event script Uevent_dberr_initial event script **Notes** The user object u_dwstandard is a custom DataWindow control you could define to standardize your error checking. It includes code (in the event scripts listed above) you might want to use in many of your DataWindow controls. To see how you can take advantage of such standardized error-checking code, look at the dw_orddetail DataWindow control in the w_orderentry window (stored in abnc_ord.pbl). It is inherited from u_dwstandard. Chapter 10, "Using DataWindow Objects"

To do this	Look here
Test for database-specific errors	Library: abnc_com.pbl User object: u_dwstandard DBError event script **Notes** To see how you might use this script in an application, look at the dw_orddetail DataWindow control in the w_orderentry window (stored in abnc_ord.pbl). It is inherited from u_dwstandard. Chapter 10, "Using DataWindow Objects"

Other handy techniques

To do this	Look here
Number rows in a DataWindow control automatically for the user	Library: abnc_ord.pbl Window object: w_orderentry PictureButton control: pb_new_item Clicked event script **Notes** Look near the middle of this script.
Perform an outer join	Library: abnc_ord.pbl Window object: w_orderselect DataWindow control: dw_choices **Notes** To see how the outer join is defined, look at the d_ordselect DataWindow object (stored in abnc_ord.pbl). See the *User's Guide*
Read from nondatabase files	Chapter 13, "Reading and Writing Text or Binary Files"
Write to nondatabase files	Chapter 13, "Reading and Writing Text or Binary Files"

For interacting with other programs

To do this	Look here
Call a C++ class function	See the *C++ Class Builder* manual
Call a database stored procedure (that doesn't return a result set)	Chapter 9, "Using Transaction Objects"
Call a DLL function	 Library: abnc_ord.pbl  Window object: w_itemdelete_helper  Timer event script **Notes** Look near the beginning of this script. Chapter 17, "Adding Other Processing Extensions"
Communicate via DDE	Chapter 14, "Using DDE in an Application"
Communicate via OLE 2.0	Chapter 15, "Using OLE in an Application"
Display OLE 1.0 columns in a DataWindow	 Library: abnc_gde.pbl  Window object: w_guides  DataWindow control: dw_guidedocs **Notes** This DataWindow control uses the d_guidedocs DataWindow object (stored in abnc_gde.pbl) to display an OLE 1.0 column (which is labeled *Text*). For further details, look in the Clicked event script of the cb_edit CommandButton control (in the w_guides window). Chapter 15, "Using OLE in an Application"
Execute an AppleScript script	See the *User's Guide*

To do this	Look here
Execute other applications	Library: abnc_com.pbl Menu object: m_abncmenu ➤ Menu item: m_guide ➤ Menu item: m_employeemotivationa Clicked event script ➤ Menu item: m_employeemotivationb Clicked event script ➤ Menu item: m_employeefun Clicked event script **Notes** You can see how these scripts actually work in two different menu objects that are inherited from m_abncmenu. They are: m_custmenu (used in the w_customer window and stored in abnc_ord.pbl) and m_ordermenu (used in the w_orderentry window and stored in abnc_ord.pbl).
Log on to an electronic mail system	Library: abnc_ord.pbl Window object: w_invoice CommandButton control: cb_mail Clicked event script Chapter 16, "Building a Mail-Enabled Application"
Send electronic mail	Library: abnc_ord.pbl Window object: w_mail **Notes** To learn about implementing a window like this that lets users compose and send e-mail messages, look at: The Clicked event scripts for the CommandButton controls in w_mail The two user-defined functions in w_mail The Close event script of w_mail Chapter 16, "Building a Mail-Enabled Application"

For producing output

To do this	Look here
Export data to files (of various formats)	Library: abnc_ord.pbl Window object: w_invoice CommandButton control: cb_mail Clicked event script **Notes** Look at the end of this script to see code that exports the contents of a DataWindow to a specific file in a specific format. Library: abnc_ord.pbl Window object: w_datareview_custsheet User-defined window function: wf_export Window object: w_datareview_ordsheet User-defined window function: wf_export **Notes** These functions contain code that makes the application prompt users to specify the file and format to which they want the contents of a DataWindow exported.
Migrate tables within or between databases	Library: abnc_gde.pbl Window object: w_sales_extract **Notes** This window lets a user choose and then execute either of two pipeline objects (named pipe_sales_extract1 and pipe_sales_extract2). These pipeline objects migrate data from the Sales_rep and Sales_summary tables of the sales database to a new table (Quarterly_extract). For further details, look in the window's Open event script and in the Clicked event script of the window's CommandButton controls. Chapter 12, "Piping Data Between Data Sources"

To do this	Look here
Print a report	Library: abnc_ord.pbl Window object: w_invoice CommandButton control: cb_print Clicked event script Chapter 10, "Using DataWindow Objects" Chapter 18, "Printing from an Application"
Print the contents of a regular DataWindow	Library: abnc_ord.pbl Window object: w_datareview_custsheet User-defined window function: wf_print Window object: w_datareview_ordsheet User-defined window function: wf_print **Notes** You'll see that you can use the same technique for printing report DataWindows and regular DataWindows. Chapter 18, "Printing from an Application" Also see *Getting Started*
Prompt for printer setup information	Library: abnc_com.pbl Menu object: m_abncmenu Menu item: m_file.m_printersetup Clicked event script **Notes** You can see how this script actually works in two different menu objects that are inherited from m_abncmenu. They are: m_custmenu (used in the w_customer window and stored in abnc_ord.pbl) and m_ordermenu (used in the w_orderentry window and stored in abnc_ord.pbl). Chapter 18, "Printing from an Application" Also see *Getting Started*

For object-oriented programming

To help you find the techniques you want in this category, they are organized into:

- Basic inheritance techniques
- Script and function techniques

Basic inheritance techniques

To do this	Look here
Inherit from an ancestor menu object	Library: abnc_com.pbl Menu object: m_abncmenu **Notes** This is an ancestor menu object from which these others are inherited: *m_custmenu* (used in the w_customer window and stored in abnc_ord.pbl) *m_ordermenu* (used in the w_orderentry window and stored in abnc_ord.pbl) Library: abnc_com.pbl Menu object: m_mdimenu **Notes** This is an ancestor menu object from which these others are inherited: *m_datareview_framemenu* (stored in abnc_ord.pbl) and its two descendants: *m_datareview_custsheetmenu* (stored in abnc_ord.pbl) *m_datareview_ordsheetmenu* (stored in abnc_ord.pbl) m_mdimenu is a generic MDI menu. Its three descendants are specialized versions used by the window object w_datareview_frame (stored in abnc_ord.pbl) depending on which kind of sheet, if any, is active. See the *User's Guide*

To do this	Look here
Inherit from an ancestor user object	 Library: abnc_com.pbl  User object: u_dwstandard **Notes** This is a customized DataWindow control that you could define to standardize your error checking. It includes code that you might want to use in many of your DataWindow controls. For instance, look at the dw_orddetail DataWindow control in the w_orderentry window (stored in abnc_ord.pbl). It is inherited from u_dwstandard.  Library: abnc_com.pbl  User object: u_custom_toolbar **Notes** This is a custom control that you could use to provide your own toolbar in a window. For instance, look at the uo_toolbar UserObject control in the w_customer window (stored in abnc_ord.pbl) and you'll discover that it is inherited from u_custom_toolbar. Another thing to look at is the user object *u_custom_ordertoolbar* in abnc_ord.pbl. It is inherited from u_custom_toolbar and specialized for use in the order entry portion of the application. If you examine the w_toolbar window (in abnc_ord.pbl), you'll see a UserObject control (uo_toolbar) that is in turn inherited from u_custom_ordertoolbar. See the *User's Guide*

To do this	Look here
Inherit from an ancestor window object	 Library: abnc_com.pbl  Window object: w_startup **Notes** This is an ancestor window object from which these others are inherited: *w_sales_logon* (stored in abnc_gde.pbl) *w_startup_ord* (stored in abnc_ord.pbl) The ancestor is a generic server-database logon window, while the descendants are specialized versions for particular databases.  Library: abnc_com.pbl  Window object: w_mdiframe **Notes** This is an ancestor window object from which the following window object is inherited: *w_datareview_frame* (stored in abnc_ord.pbl) The ancestor is a generic MDI frame, while the descendant is an MDI frame specialized for use in the data review portion of the Order Entry application.  Library: abnc_com.pbl  Window object: w_mdisheet **Notes** This is an ancestor window object from which these others are inherited: *w_datareview_custsheet* (stored in abnc_ord.pbl) *w_datareview_ordsheet* (stored in abnc_ord.pbl) The ancestor is a generic MDI sheet, while the descendants are specialized versions for working with particular kinds of information within the w_datareview_frame window. See *Getting Started* Also see the *User's Guide*

Script and function techniques

To do this	Look here
Call an ancestor event script manually	Library: abnc_ord.pbl Menu object: m_custmenu ➤ Menu item: m_file.m_exit Clicked event script **Notes** The ancestor of the menu object m_custmenu is the menu object m_abncmenu (stored in abnc_com.pbl). See the *User's Guide*
Call an ancestor function	See the *User's Guide*
Extend an ancestor event script	Library: abnc_ord.pbl Window object: w_datareview_custsheet Open event script **Notes** This script extends the corresponding script from the window object w_mdisheet (stored in abnc_com.pbl), which is the ancestor of w_datareview_custsheet. That means it includes some specialized code of its own that is to execute *after* the code from the ancestor script. See the *User's Guide*
Overload user-defined functions	Library: abnc_ord.pbl Menu object: m_datareview_framemenu ➤ Menu item: m_file.m_export Clicked event script ➤ Menu item: m_file.m_print Clicked event script

To do this	Look here
Override an ancestor event script	 Library: abnc_gde.pbl  Window object: w_sales_logon  CommandButton control: cb_cancel  Clicked event script **Notes** This script overrides the corresponding script from the window object w_startup (stored in abnc_com.pbl), which is the ancestor of w_sales_logon. That enables w_sales_logon to do some specialized processing of its own instead when users click its cb_cancel CommandButton control. See the *User's Guide*

For other needs

To do this	Look here
Animate images by using a timer	Library: abnc_ord.pbl Window object: w_orderentry Picture control: p_trashcan DragDrop event script Library: abnc_mot.pbl Window object: w_mdiframe_fun Uevent_createsheets event script
Define and execute user events	Library: abnc_ord.pbl Window object: w_orderentry Uevent_listorders event script Library: abnc_com.pbl User object: u_dwstandard Uevent_accepttext event script Uevent_dberr_reqmissing event script Uevent_dberr_initial event script **Notes** To see how you can take advantage of these user events, look at the dw_orddetail DataWindow control in the w_orderentry window (stored in abnc_ord.pbl). It is inherited from u_dwstandard. See the *User's Guide*
Handle windowing-system messages	Chapter 17, "Adding Other Processing Extensions"
Implement a wizard to guide users through a procedure	Library: abnc_ord.pbl Menu object: m_ordermenu Menu item: m_file.m_delete.m_itemdel Clicked event script **Notes** When you read this script, you'll learn that it opens the w_itemdelete_helper window object (stored in abnc_ord.pbl) to implement a row-deletion wizard. Look at this window object and read its Timer event script.

To do this	Look here
Perform spreadsheet-style recalculation in a window	 Library: abnc_ord.pbl  Window object: w_orderentry  DataWindow control: dw_orddetail  ItemChanged event script
Play sound files	 Library: abnc_ord.pbl  Window object: w_itemdelete_helper  Timer event script **Notes** Look near the beginning of this script. Chapter 17, "Adding Other Processing Extensions"

Taking advantage of development support facilities

As you work in PowerBuilder and its painters to develop the features of an application, you'll want to take advantage of their many development support facilities. These facilities will help you get your tasks done more quickly and more easily.

The following sections list some of the facilities you'll find especially useful when you want to:

- Get information
- Code scripts
- Manage your work
- Customize your environment

For complete details on the PowerBuilder painters and facilities listed in these sections, see the *User's Guide*.

To help you get information

Here are some of the ways you can get information to assist your development work:

When you want to	You can
Display attributes and functions of the object you're currently editing	Use the Browse Object facility (Quick browser) in the PowerScript painter
Display attributes, functions, variables, names, and structures of various objects	Use the Browse Objects facility (Object browser) in the PowerScript painter or in the Library painter
Display inheritance information about a specific object	Go to the Application painter and view the Object Hierarchy
Display inheritance information about class hierarchies	Go to the Library painter and use the Class Browser
Display OLE class names	Use the Browse OLE Classes facility in the PowerScript painter
Find a particular text string in objects	Use the Browse Library Entries facility in the Library painter or the Search facilities in the PowerScript painter
Get reports about objects	Go to the Library painter and print particular objects or library directories

When you want to	You can
Read documentation about using PowerBuilder	Go to the online Help system
See the ancestor version of a script you're editing	Use the Display Ancestor Script facility in the PowerScript painter
Track object access in a multiple-developer environment	Go to the Library painter and view the check-out status
Track versions of objects	Go to the Library painter and display their registration directories and reports
View a complete or selected list of the objects in a library	Go to the Library painter
Walk the tree structure of an application	Go to the Application painter and expand or collapse branches

To help you code scripts

In addition to using the basic editing facilities in the PowerScript painter, here are some extra things you can do to facilitate your script coding work:

When you want to	You can
Borrow code from the ancestor version of a script you're editing	Copy it from the Display Ancestor Script facility in the PowerScript painter
Borrow information or syntax from PowerBuilder documentation	Copy it from the online Help system
Code DataWindow syntax (including the Describe, Modify, and SyntaxFromSQL functions)	Run DWSYN40.EXE, a utility application included with PowerBuilder
Get script code from a text file	Use the File Import facility in the PowerScript painter
Insert functions into your script code	Paste them by using the Paste Function facility in the PowerScript painter
Insert OLE class names into your script code	Paste them by using the Browse OLE Classes facility in the PowerScript painter
Insert PowerScript control statements into your script code	Paste them by using the Paste Statement facility in the PowerScript painter
Insert SQL statements into your script code	Paste them by using the Paste SQL facility in the PowerScript painter

When you want to	You can
Store script code in a text file	Use the File Export facility in the PowerScript painter
Use attributes and functions of the object you're currently editing	Paste them from the Browse Object facility (Quick browser) in the PowerScript painter
Use attributes, functions, variables, names, and structures of various objects	Paste them from the Browse Objects facility (Object browser) in the PowerScript painter or in the Library painter

To help you manage your work

Here are some of the things you can do to manage the objects and other components you're developing:

When you want to	You can
Access objects from shared libraries in a multiple-developer environment	Check them out (and later check them back in) in the Library painter
Access or store particular versions of objects	Use the version control facilities in the Library painter and the Project painter
Clean up the internal storage of your libraries (to ensure the best performance)	Optimize them in the Library painter
Create an object from a text file that contains its exported source code	Import that text file into one of your libraries in the Library painter
Document your libraries and their objects	Enter comments when saving them in their respective painters, or modify their comments in the Library painter
Edit text files	Go to the File Editor (such as by selecting it from the PowerPanel)
Jump to a painter to work on an existing object	Select that object from the tree in either the Application painter or the Library painter
Produce a text file that contains the source code for an object	Export that object in the Library painter
Recompile objects in your libraries (such as when you want descendants to pick up modifications made to their ancestors)	Regenerate them in the Library painter

When you want to	You can
Reorganize objects in your libraries	Copy, move, or delete them in the Library painter
Reorganize your libraries	Create or delete them in the Library painter

To help you customize your environment

Here are some of the ways you can customize your development environment to better suit your needs:

When you want to	You can
Customize PowerBuilder and its painters	Go to the Preferences painter
Customize the PowerBuilder toolbars (PowerBar and PainterBar)	Use the Toolbars facility, which is available in the painters as well as when no painters are open

Other tools that might help you

In addition to the development support facilities that are built right into PowerBuilder, Powersoft produces some supplementary tools that you may want to use.

Tools from Powersoft

These include:

- The **Powersoft Infobase CD-ROM**, which contains a wide variety of technical support information and other resources designed to help you get the most out of the Powersoft products you use
- The **PowerBuilder Advanced Developer Toolkit**, which includes utilities for testing, maintaining, deploying, and reporting on applications

For more information on these tools, look in your PowerBuilder package.

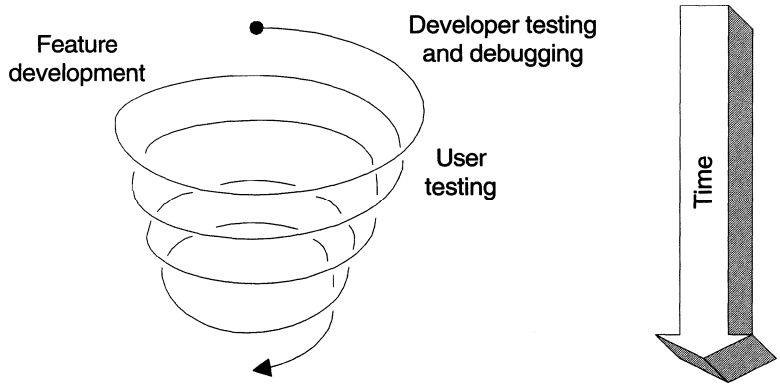
Tools from other vendors

You may also want to consider some of the development support tools that are available from other vendors for use with PowerBuilder. You'll find a lot of them to choose from, each providing one or more particular services.

For a list of these tools, look in your PowerBuilder package or talk to a Powersoft sales representative.

Testing what you implement

As you work with a rapid application development tool such as PowerBuilder, you should get into the habit of testing and debugging features in small increments as you implement them. A highly *iterative* approach will make it much easier to track down particular problems as well as help ensure that you're always building on a solid code-base.



PowerBuilder supports this approach by providing a variety of handy testing and debugging facilities. In just a moment you'll get an introduction to these facilities, but first it's worth taking a broader look at the topic of testing, including:

- A discussion of what you want to find out while testing your application
- Some guidelines about how to track down problems in client/server applications

Determining what you want to find out while testing

Each time you test a particular feature you've developed, you'll probably want to evaluate its success based on *several* different criteria. For example:

- **Are there any bugs in your implementation of the feature?**

  Sometimes these problems are obvious because they result in execution-time error messages or even crashes. Other times they are more subtle and show up only after thorough testing—this is especially true for problems caused by errors in your logic.

- **Did you choose an appropriate technique for implementing the feature?**

  You'll want to make sure that your technique provides the functionality you're trying to implement (or enough of it to be acceptable). If not, you may need to try a different technique.

- **Does the feature meet your requirements for performance?**

  Conduct any tests you need to look at speed, resource usage, network traffic, and whatever other measures of performance are important to you. Depending on your test results, you might have to make some adjustments to your feature, or maybe you'll want to consider a different technique altogether.

- **Does the feature meet the end user's needs?**

  You have to determine whether your completed feature is really what users are asking for. It may be necessary to conduct some user tests to find out.

Keep these questions in mind as you test, and add any others that apply to your specific situation. That way you'll avoid overlooking bugs or other deficiencies that could come back to bite you later.

Tracking down problems

Because client/server applications typically involve so many separate players in so many separate places, they *can* be difficult to debug. But if you follow these general guidelines, you should be able to analyze a given problem effectively and pinpoint its source.

Before you start debugging Here are some things you should do in preparation for tracking down a problem:

What to do	Why
Make sure your system is clean and stable. If the system has experienced a serious failure (such as one involving memory corruption), restart it first.	An unstable system can introduce a great deal of confusion. For example, it can cause you to waste time looking for problems that have nothing to do with your application.
Back up your libraries (PBL files).	This will ensure that you have a good copy of your application to return to in case you want to abandon changes made while debugging.

What to do	Why
Regenerate the application's objects (in the Library painter).	This can help uncover a variety of errors (such as object references that are no longer valid for one reason or another).
Check your test data to make sure it is still good. For instance, does it really contain the values you think it does?	Invalid test data can mislead you both by hiding application problems and by making you think there are application problems where none exist.

While you're debugging Once you're ready to start tracking down a problem, here are some tips for how you should proceed:

What to do	Why
Eliminate external factors that may be contributing to the problem. For example, you might try: ♦ *Eliminating network issues* by running the application locally ♦ *Eliminating database issues* by accessing simulated data ♦ *Eliminating configuration issues* by checking the resource usage and other aspects of your client computer	The problem might not actually reside in one of your objects or libraries, but in one or more other parts of your environment (such as the network, the database, or the client computer's configuration).
Isolate the problem within your application. For instance, you might try: ♦ *Commenting-out* code in one or more scripts ♦ *Copying* suspect objects, controls, or code for isolation testing in a simpler context	You won't want to start making fixes until you know exactly what portion of your application is causing the problem. So it's important that you know how to narrow your search as quickly as possible.
Fix only one problem at a time.	If you change too many things at once, you may frustrate or mislead yourself unnecessarily. Remember that it's easier to solve for one variable than for two.

Choosing the testing facilities to use

To help you do your testing and debugging work, PowerBuilder provides a number of facilities, including the following.

For syntax and reference checking

When you want to	Use
Catch syntax errors in a script	The PowerScript painter to compile that script
Catch reference and inheritance errors among objects	The Library painter to regenerate those objects

For execution-related testing

When you want to	Use
Preview an object as you paint it (to see how it will look when displayed by the application)	The Preview facility in the appropriate painter (such as in the Window painter, Menu painter, DataWindow painter, or Query painter)
Test run just a particular window (without running the whole application)	The Run Window facility in the Window painter
Test run the application in **regular mode**—to see it the way end users will see it	The Run facility, which is always available in PowerBuilder (for the current application)
Test run the application in **debug mode**—to be able to: ♦ *Stop* it at breakpoints ♦ *Step* through its script code line by line ♦ *Look* at the contents of variables ♦ *Modify* the contents of variables for test purposes ♦ *Watch* how variables change	The Debug facility, which is always available in PowerBuilder (for the current application)
Make the application beep to let you know when some test condition occurs	The Beep function in an appropriate script For more information, see the *Function Reference*

When you want to	Use
Make the application display a message to let you know when some test condition occurs or to show you some values	Either of the following: ♦ The *MessageBox function* in an appropriate script (but not in a focus-related event script) ♦ The *Title attribute* of a window (by assigning your message text to it in an appropriate script)
Trap serious errors when running the application (so that you can handle them with code of your own instead of relying on the PowerBuilder default execution-time error processing)	The SystemError event of the application object

For database-related testing

When you want to	Use
Test and analyze SQL statements interactively	The Database Administration painter
Examine the SQL statements generated by a DataWindow before they are sent to the database for execution	The GetSQLPreview function in the SQLPreview event script of the appropriate DataWindow control For more information, see the *Function Reference*
Trace the interactions between your application and a database it accesses	The Database Trace utility For more information, see *Connecting to Your Database*
Examine values in your test database	The Database Administration painter, Data Manipulation painter, or Query painter

For more information on these topics, see the *User's Guide* (except where noted).

Tools from other vendors You may also want to consider some of the testing and debugging tools that are available from other vendors for use with PowerBuilder. These include tools for automated testing of PowerBuilder applications.

For a list of these tools, look in your PowerBuilder package or talk to a Powersoft sales representative.

Where to go from here

Once you've implemented all of the features required for your application and they pass your tests, you can wrap up the application and deploy it to the people who need to use it.

To learn the details of what you need to do in the deployment phase of your project, turn to Chapter 5, "Deploying an Application."

CHAPTER 5

Deploying an Application

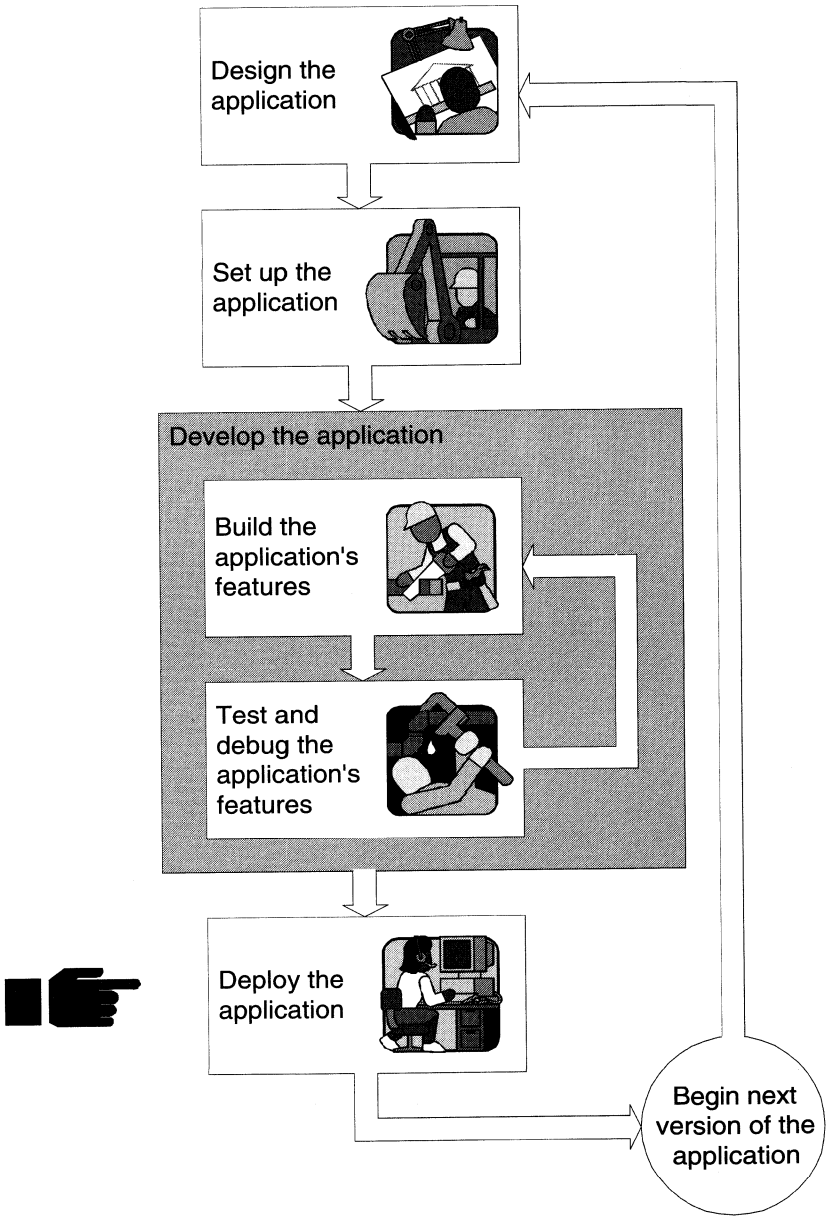
About this chapter

This chapter tells you about the process of deploying an application that you've developed.

Contents

Orientation

Here's where you are now in the lifecycle of your application development project:


Design the application
Set up the application
Develop the application
Build the application's features
Test and debug the application's features
Deploy the application
Begin next version of the application

Tying up any loose ends

Once you've built and tested your application's features, you need to think about packaging it into a form that can ultimately be distributed to users. That means creating an **executable** version of it—a version that people can run on its own, outside of the PowerBuilder development environment.

The first step in this process is to tie up any loose ends that remain from your development work on the application.

Making a checklist for yourself

There are typically a few different cleanup chores that developers need to take care of in preparation for generating an executable. These generally involve removing or changing things that you put into the application just to support your development and testing work.

During the course of a project, it's a good idea to keep a checklist of these chores and add to it as needed. That way, you'll know what it will take to wrap up at the end and how much time it will require. For example, your checklist may remind you to:

- **Remove or comment out any test code** that you inserted in the application's event scripts or functions
- **Adjust the application to access a production database** instead of your test version of that database

  This usually involves changing one or more connection parameters in the application's INI file or in a script to point to the production database. (That way, your application can assign these parameters to its transaction object and use them instead of the old ones when connecting.)
- **Check in all objects** that you've checked out of public libraries

  Make sure that your other development team members check in their work as well.
- **Modify the application's library list** if necessary to point to the completed versions of all the appropriate libraries

  You may need to do this if you work in a multi-tier development environment and juggle different versions of libraries (such as production, QA, and development) during the course of your project.

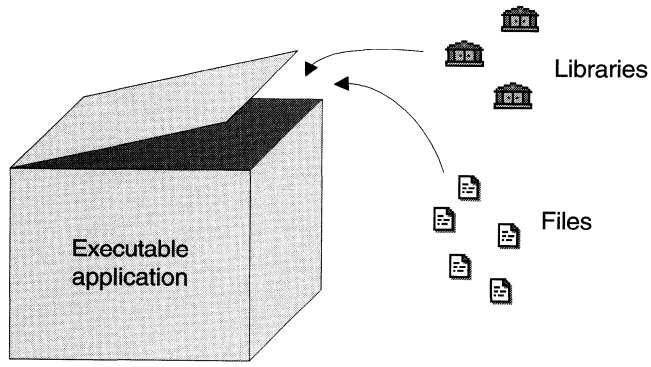
If you just want an executable for testing

Sometimes you'll want to create an executable version of your application for *test* purposes, before the application is actually ready to distribute for production use. In fact, you may want to do this on a periodic basis over the course of your project—maybe at particular time intervals (such as every week or two) or maybe when particular functionality milestones have been achieved.

In these cases, you probably won't need to perform most of the chores on your cleanup checklist. As a result, you'll be ready to create your executable that much quicker.

Creating an executable version of your application

Creating an executable version of an application is a matter of *packaging* the various pieces it needs in an appropriate way.



To get the kind of executable you want, you need to know how to choose the right pieces and the right model for packaging them together. Once you have this knowledge, you can use the packaging tools that PowerBuilder provides to do the job.

For the details

The next few sections tell you more about the packaging process and provide information to help you make choices about the resulting executable. You'll read about:

- 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

Learning what can go in the package

An executable application that you create in PowerBuilder can consist of one or more of the following pieces:

- An executable file
- PowerBuilder 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

You'll *always create exactly one* executable (EXE) file for any PowerBuilder application you want to deploy.

At minimum, this file contains code that enables your application to run as a native application on its target platform. That means when Windows users want to start your application, they can double-click the executable file's icon in Program Manager. Macintosh users can double-click the icon on the desktop.

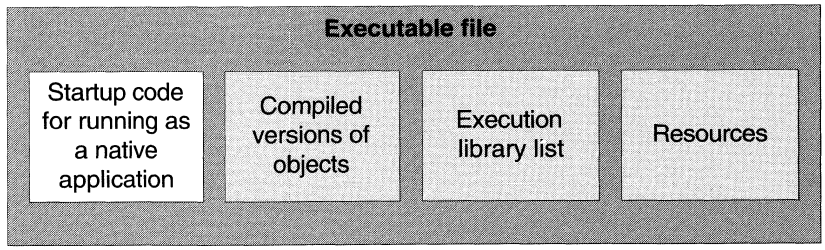
What else can go in it Depending on the packaging model you choose for your application, the executable file will also contain one or more of the following:

- **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 to distribute only this one file) or you can choose to split your application into one executable file and one or more PowerBuilder dynamic libraries (PBD files). More about that in just a moment.

- **An execution library list** that the PowerBuilder execution system uses to find objects and resources in any PowerBuilder dynamic libraries you've packaged for the application

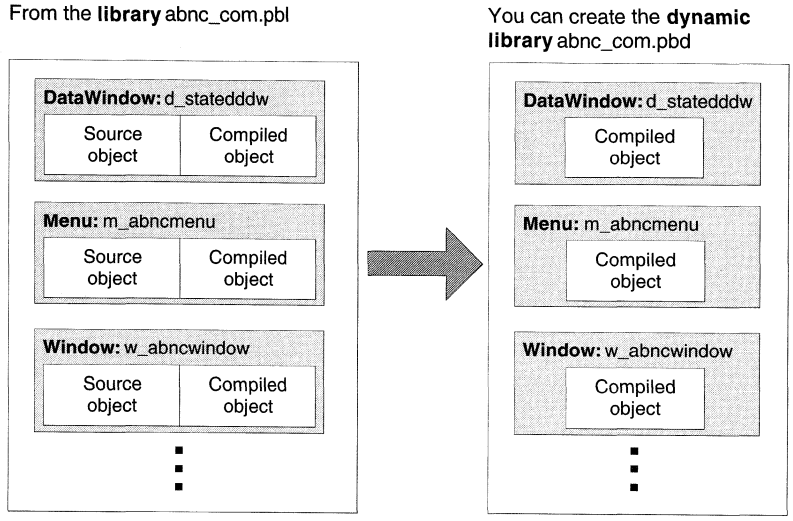
- **Resources** that your application uses (such as bitmaps)



Size guidelines Try to keep an executable file smaller than 1.2 to 1.5 megabytes. If it approaches that size, consider using PowerBuilder dynamic libraries.

About PowerBuilder dynamic libraries

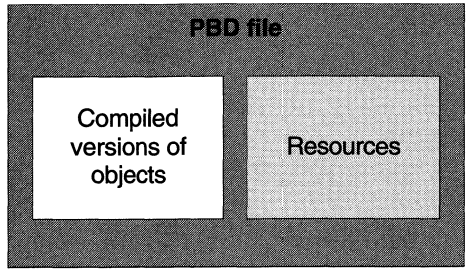
As an alternative to putting your entire application in one large executable file, you can distribute some (or even all) of its objects in one or more PowerBuilder dynamic libraries (PBD files). As with an executable file, only the *compiled* versions of objects go into PBD files (not their source):



PBD files are similar to dynamic link libraries (DLLs) in that they are linked to your application at execution time. But just remember that PBD files are *not* interchangeable with DLLs—they have different internal formats.

What else can go in them Unlike your executable file, PBD files don't include any startup code. That's because they can't be executed independently. Instead, they are accessed when needed as an application executes to provide objects it requires (if it can't find those objects in the executable file).

But PBD files can include resources (such as bitmaps). In fact, you'll often want to put any resources needed by a PBD file's objects in that PBD file—this makes the PBD file a self-contained unit that can easily be reused (although if performance is your main concern, you'll find that resources are loaded faster at execution time when they're in the executable file).



Why use them There are several reasons why you might want to use PBD files:

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 distribute your application components separately. That means when you need to provide users with an upgrade or bug fix, you don't have to redistribute the entire application—you can just give them the particular PBD file that was affected.
Reusability	They make it possible for multiple applications to reuse the same components. That's because PBD files 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 execution time (such as a window object referenced only through a string variable). You can't put such objects in your executable file (unless they are DataWindow objects).
Efficiency	They can help a large application use memory efficiently because: ♦ PowerBuilder doesn't load an entire PBD file into memory at once. Instead, it loads individual objects from the PBD file 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 PBD file, you'll need to tell PowerBuilder which library (PBL file) to create it from. PowerBuilder then copies the compiled versions of *all* objects from that PBL file into the PBD file.

But suppose your application uses only some of those objects. In that case, you may not want the PBD file to include the superfluous ones, since they'll just serve to make the file larger than it has to be. The solution is to:

1 **Create a new PBL file** and copy just the objects you want into it.

2 **Use this new PBL file** as the source of your PBD file.

For further details on these steps, see the *User's Guide*.

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 distribute 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. Here's a list of them, organized by the specific objects in which they might be needed:

These objects	Can use these kinds of resources
Window objects and user objects	Icons (ICO files) Pictures (BMP, RLE, and WMF files) Pointers (CUR files)
DataWindow objects	Pictures (BMP, RLE, and WMF files)
Menu objects (when in an MDI application)	Pictures (BMP, RLE, and WMF files)

Distributing 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.

  But PowerBuilder doesn't 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, 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 PBD files, if you decide to put most or all resources in the executable file for performance reasons.)

- **Include them in PBD files**

  You'll often want to include resources right in one or more PBD files. But PowerBuilder doesn't automatically copy any resources into a PBD file that you ask to create (even if they are explicitly referenced by objects in that file). So you'll need to produce a PBR file that tells PowerBuilder which resources you want in this particular PBD file.

  Use a different PBR file for each PBD file in which you want to include resources. (When appropriate, you can even use this approach to generate a PBD file that contains only resources and no objects—simply start with an empty PBL file as the source.)

- **Distribute them as separate files**

  This means that when you deploy the application, you'll give users various ICO, BMP, RLE, WMF, and CUR files in addition to the application's executable file and any PBD files. As long as you don't mind distributing a lot of files, this can be useful if you expect to revise some of them in the future.

But keep in mind that this is not the fastest approach at execution time because it requires more searching. Here's what happens. Whenever your application needs a resource, it first searches the executable file. If the resource isn't there, it next searches the PBD files. If the resource isn't there, it finally searches for a separate file.

Make sure your application can find where these separate files are stored. Otherwise it won't 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

You may 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're 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 *don't* need to do this when creating a PBD file. PowerBuilder will automatically copy every DataWindow object from the source library (PBL file) to your new PBD file.

For more information on working with PBR files, see the *User's Guide*.

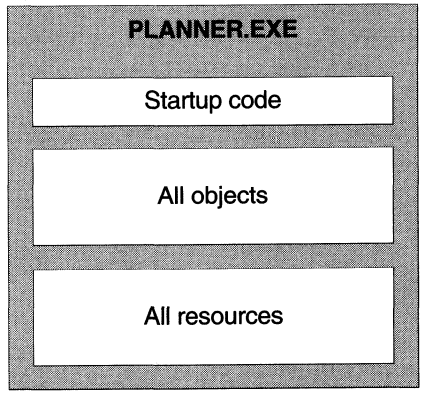
Choosing a packaging model

As you can see from reading the previous section, you've got a lot of options when it comes to packaging an executable version of an application. To help you choose what's best for your project, 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. That means there's just one file to distribute.

Illustration Here's a sample of what this model can look like:

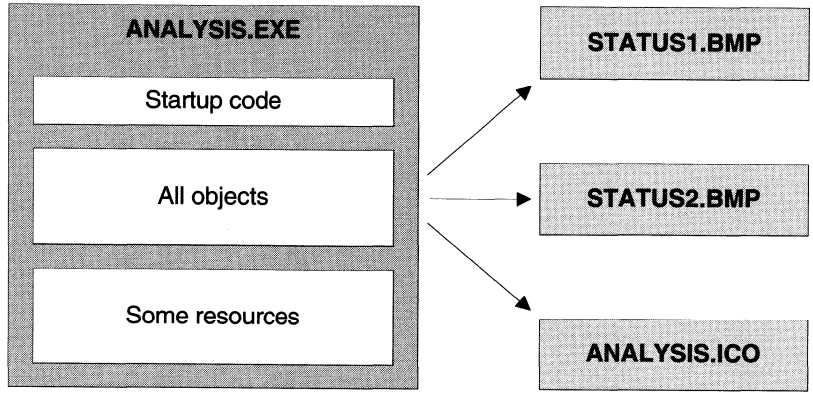


Use This model is good for small, simple applications—especially those you expect won't need a lot of maintenance. For such projects, it ensures the best performance and the easiest distribution.

An executable file and external resources

In this model, you include all objects and most resources in the executable file. But you distribute separate files for particular resources.

Illustration Here's a sample of what this model can look like:



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 distribute a revised copy of the executable file.

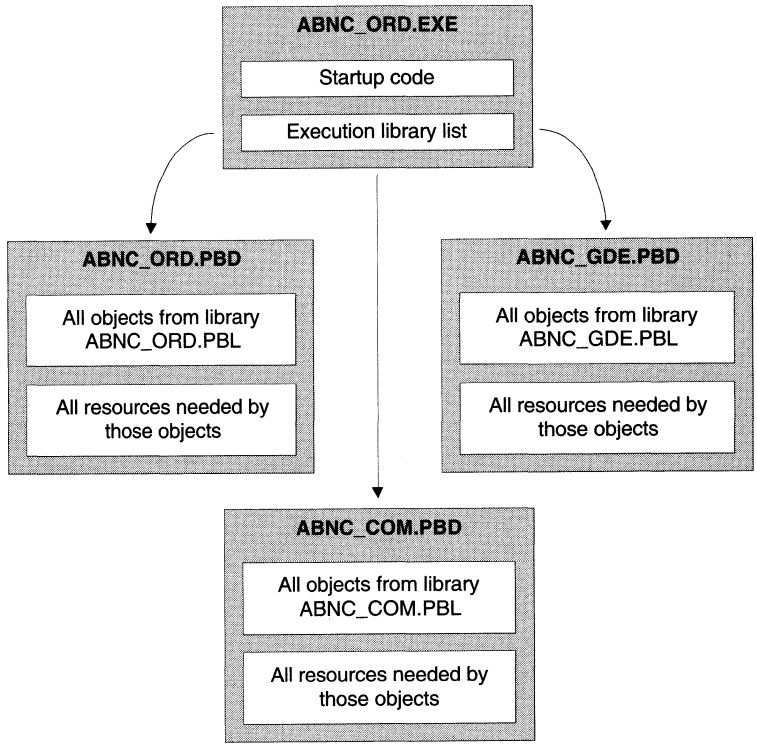
You could also use this model to deal with resources that must be shared by other applications or that are large and infrequently needed.

An executable file and dynamic libraries

In this model, you split up your application into an executable file and one or more PBD files. When doing so, you can organize your objects and resources in various ways, including these:

To organize	You can
Objects	Place them all in PBD files 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 PBD files
Resources	Place most or all of them in PBD files 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 Here's a sample of what this model can look like:



Use This model is good for most substantial projects. That's because of the flexibility it gives you in organizing, distributing, and maintaining your applications.

For instance, it enables you to make revisions to a particular part of an application (in one PBD file) and distribute that revised PBD file to users without disrupting the remainder of the application. It also lets you distribute the same PBD file for use in multiple applications.

Looking at the sample Order Entry application

As you may have noticed, this is the packaging model used for the Anchor Bay Nut Company's Order Entry application. Specifically, it takes the approach of placing all objects in its three PBD files and none in the executable file. It also places all resources in those PBD files along with the objects that need them.

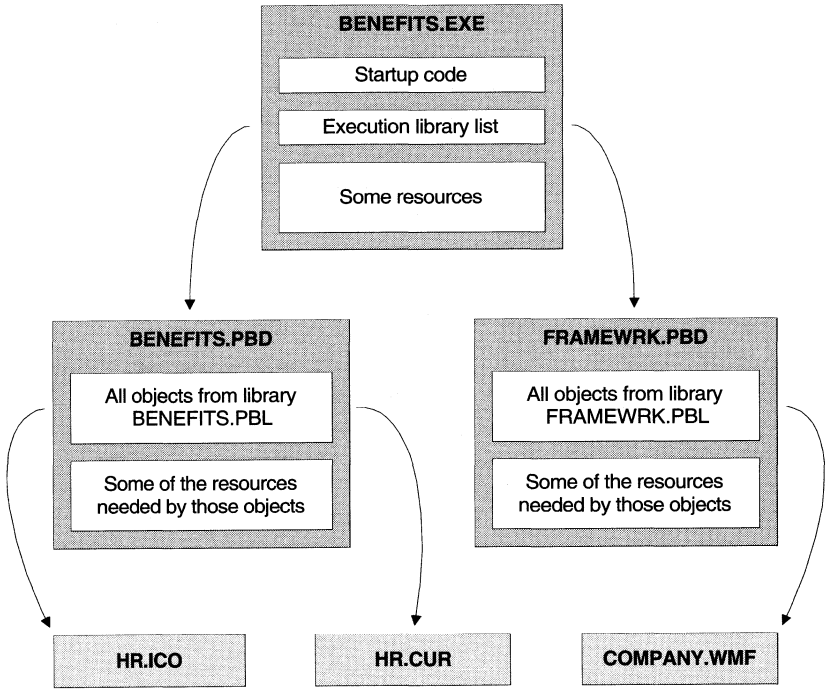
Take a closer look at the application's files to see how this approach works. For example, examine:

- Its *PBR files* (ABNC_ORD.PBR, ABNC_GDE.PBR, and ABNC_COM.PBR) and how they list resources that are to go into particular PBD files
- How objects are organized among its *PBL files* to produce PBD files that make the most sense
- The *library search path* (in the Application painter), because it specifies the execution library list for the application

An executable file, dynamic libraries, and external resources

This model is just like the preceding one except that you distribute separate files for particular resources (instead of including all of them in your PBD and executable files).

Illustration Here's a sample of what this model can look like:



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've figured out the appropriate packaging model for your application, you can use the packaging facilities in PowerBuilder to implement it. Actually, there are a couple of different ways you can go, depending on whether you're deploying:

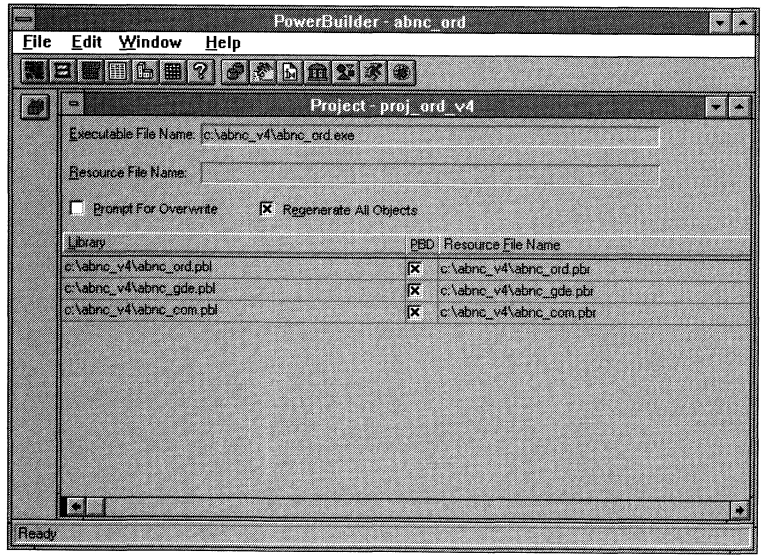
- An application of significant size, *or*
- A small, simple application

For most significant applications

If your application is large enough to involve PBD files, then you'll typically want to use the **Project painter** in PowerBuilder to create your executable version of it. The Project painter orchestrates all aspects of the packaging job by enabling you to:

- Specify the **executable file** to create
- Specify any **PBD files** to create
- Specify the **resources you want included** in the executable file or in each particular PBD file (by using appropriate PBR files that indicate where to get those resources)
- Save all of these specifications as a **project object** that you can use whenever necessary to rebuild the whole package

Example At the Anchor Bay Nut Company, they used the Project painter to define their packaging model for the Order Entry application:



Then they saved this information in a project object named proj_ord_v4. Now, anytime they want a new executable version of the application, they can simply:

1 Open that project object.

2 Tell the Project painter to build the executable application from it.

Recording the version of objects used If you're using a version control system with PowerBuilder, you can also have the project object remember which version of objects it used to create the executable application. That way, if someday you need to recreate all of the application's libraries (PBL files) from that particular point, you can return to the Project painter and use the Restore Libraries facility.

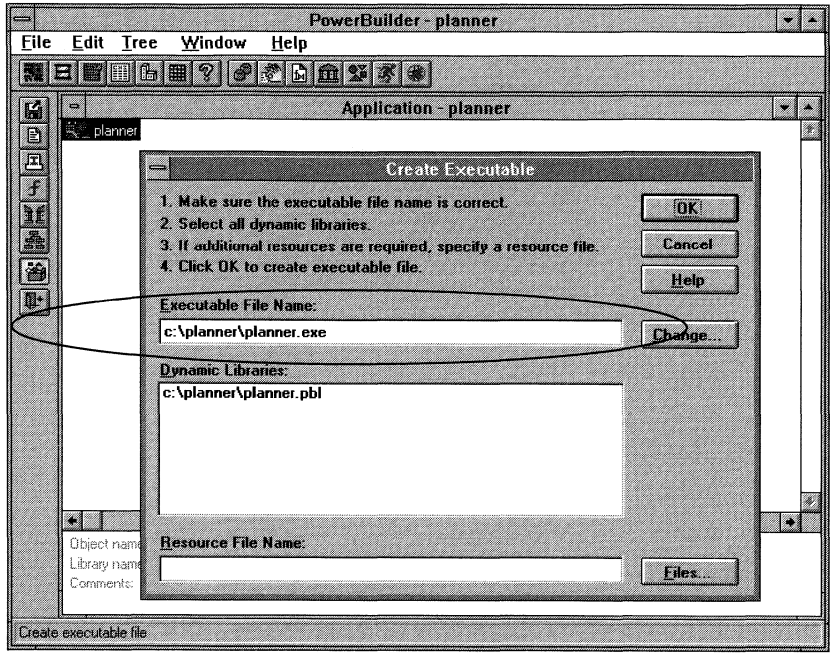
For more information on using the Project painter, see the *User's Guide*. To learn the details about version control in PowerBuilder, see *Version Control Interfaces*.

For small and simple applications

If your application is fairly basic and you decide to put all of its objects in the executable file, then you may not want to be bothered defining a project object for it. This is especially true for one-shot applications that you don't expect to revise.

In these cases, you can go to the Application painter and tell it to create that executable file:

Creating a standalone executable file



Handling more complexity Notice that if necessary you *can* use the Application painter to generate packaging models of greater complexity (although such models are usually easier in the Project painter). This includes the ability to specify:

- **PBD files** that you've built or plan to build for your application. This tells the Application painter not to include the objects from those files in the executable file it creates.

  You can use the Library painter to build any individual PBD files you need from your libraries (PBL files). These PBD files don't have to exist in order for you to create the executable file.

- A **PBR file** that you've written. This tells the Application painter to include in the executable file all of the resources listed in that PBR file.

Repackaging particular pieces of an application
When revising an existing application (of any size), you might want to distribute only those portions that have been changed—such as a certain PBD file or maybe the executable file. In these cases, your project object may do more than what you need. So instead, you can use the Library painter or Application painter to rebuild just that revised PBD or executable file.

For more information on using the Application painter and the Library painter to package the executable version of an application, see the *User's Guide*.

Testing the executable application

Once you create the executable version of your application, you'll want to test how it runs before proceeding with distribution. You may have already executed the application many times within the PowerBuilder development environment, but it's 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 (windowing) environment.

2 Run the application's executable file as you run any native application.

What to look for during a test run

As the application executes, you should monitor its behavior closely to uncover any problems. Here are some things to look for:

- Can the application **access all of its objects**?

  Pay special attention to those that are referenced dynamically (through string variables in scripts).

- Can the application **access all of its resources**?

  Pay special attention to those that are referenced dynamically (through string variables in scripts) and those that are needed by objects in your PBD files.

- Can the application **access the production database(s)** that it needs to?

- Can the application **access any external programs or files** that it needs to?

- Do all of the **features of the application work** as they did when you tested them in the PowerBuilder development environment?

- Is the **performance of the application acceptable**, given your requirements and expectations?

  For example, maybe the application takes longer than you like getting loaded into memory (because you created a really large executable file). Or maybe the application has to do too much searching for particular resources (because you've kept them in separate files instead of copying them into the executable or PBD files).

  In cases such as these, it might help you to repackage the application using a different packaging model.

Tracing the application's execution

To help you track down problems, PowerBuilder provides an **execution trace facility** that you can use when running the executable version of an application. This facility records the application's activities, including:

- The creation and destruction of objects
- The execution of event scripts and functions

It stores this information in a text file that you can read in any editor. 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 *User's Guide*.

Conducting a beta test with users

After the executable application passes your tests, you may want to conduct beta tests of it with actual end users. This typically involves:

- **Distributing** the application on a very limited basis (to a subset of its ultimate audience)
- **Monitoring** usage closely
- **Restricting** usage to nonproduction work
- **Running** in parallel with any existing production application(s) until the decision is made to go final

Why do it Testing on *user* computers can often uncover problems that don't reveal themselves on *developer* computers. In many cases, this is due to hardware and software differences between those two environments:

- **Hardware differences** For instance, developers may have more powerful processors or larger monitors than end users have. Developers may also have different network connections and peripherals.
- **Software differences** For instance, developers already have all of the PowerBuilder deployment DLLs needed to run the executable application independently (because those DLLs get installed along with the PowerBuilder development environment). In contrast, you must install the PowerBuilder deployment DLLs on user computers yourself (as described in the next section) when distributing the application.

  Developer and user computers may also differ in: which version of the operating system they're using, how that operating system is tuned and configured, and which applications, databases, or utilities are installed.

Distributing your application to end users

Distributing the executable version of your application to users is basically a matter of installing all of the right pieces in the right places (such as on their computers or on the network). This section presents a checklist you can follow to make sure you install everything that's needed. For easy reading, the checklist is divided into:

- Installing environmental pieces
- Installing application pieces

Installing environmental pieces

✓	To do	Details
	Install the PowerBuilder deployment 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 of the development environment). For details on installing the deployment DLLs, see the *Installation and Deployment Guide*. **Handling maintenance releases** If you're using a maintenance release of PowerBuilder in your development environment, make sure you provide users with the deployment DLLs from that maintenance release.
	Install the database interface(s)	You should install on each user computer any database interfaces required by the application. Possibilities include: the ODBC interface and the other Powersoft database interfaces. For details on installing any database interfaces you need, 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 those 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*.

✓	To do	Details
	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 may require some special adjustments to the operating or windowing system for performance (or other) reasons. If that's the case with your application, be sure to make those adjustments to each user computer.

Installing application pieces

✓	To do	Details
	Copy the executable application	Make copies of the files that comprise your executable application and install them on each user computer. These files can include: ♦ The executable (EXE) file ♦ Any PBD files ♦ Any files for resources you're distributing separately (such as ICO, BMP, RLE, WMF, or CUR files) **Handling maintenance releases** If you plan to revise these files on a regular basis, you may 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 deployment 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 and possibly various others (such as text or sound files). In some cases, you may want to install particular files on a server instead of locally (depending on their use).

✓	To do	Details
	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 (as described earlier in this section) if you haven't 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.
	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 will be able to search (typically the current one).
	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. **Alternative** Users can also start the application in any other manner provided for native applications under that windowing system.

Concluding the project

When the application is at last deployed and in production use, your project is just about finished. To wrap it up, you might only need to perform a few logistical chores, such as:

- **Completing any internal documentation** that remains to be done for the application and its components
- **Creating archive copies of the files** for this version of the application for safekeeping
- **Conducting a postmortem** for the project to determine what you learned that might help you improve future projects

Of course in many cases, this won't be the end of the story for your application. That's because the conclusion of a *development* project often marks the beginning of a *maintenance* project—either for you or someone else. It's also possible that you'll sooner or later need to begin work on a whole new version of the application. In that case, you'll start the whole cycle over again...

Where to go from here

Now that you know about the process of building applications in PowerBuilder, you'll probably want to start learning the various techniques you'll need to develop particular application features.

For more information

To learn about developing	Read
Basic application features	*Getting Started*
User interface features	Part Two of *Building Applications*
Data access features	Part Three of *Building Applications*
Program access features	Part Four of *Building Applications*
Miscellaneous features	Part Five of *Building Applications*

PART TWO

User Interface Techniques

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

CHAPTER 6

Building an MDI Application

About this chapter

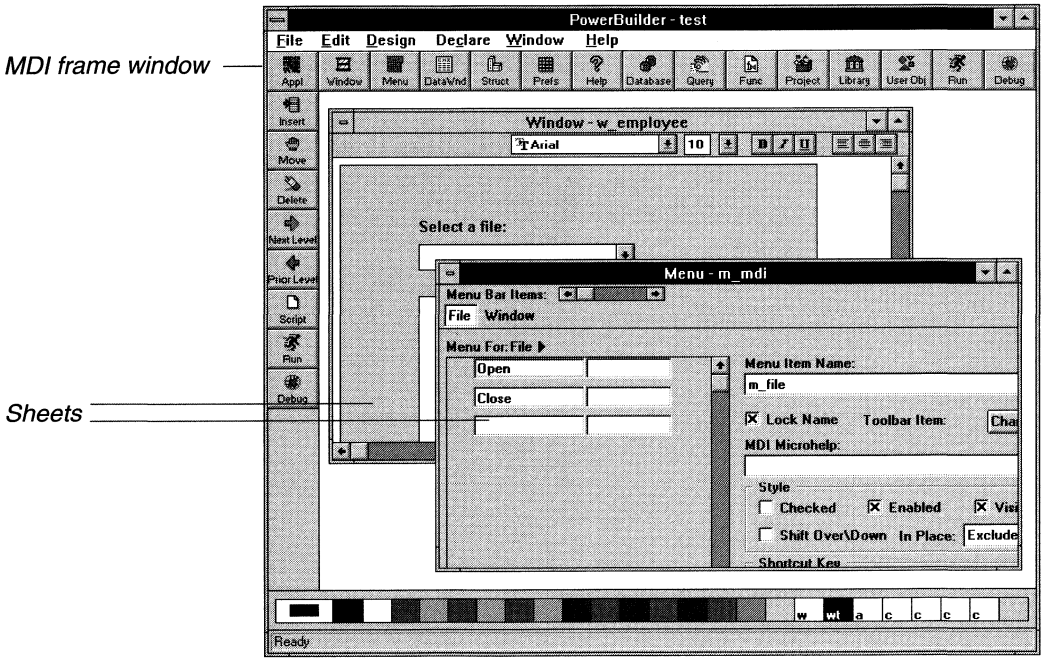
This chapter describes how to build a Multiple-Document Interface (MDI) application in PowerBuilder.

Contents

Overview of 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.

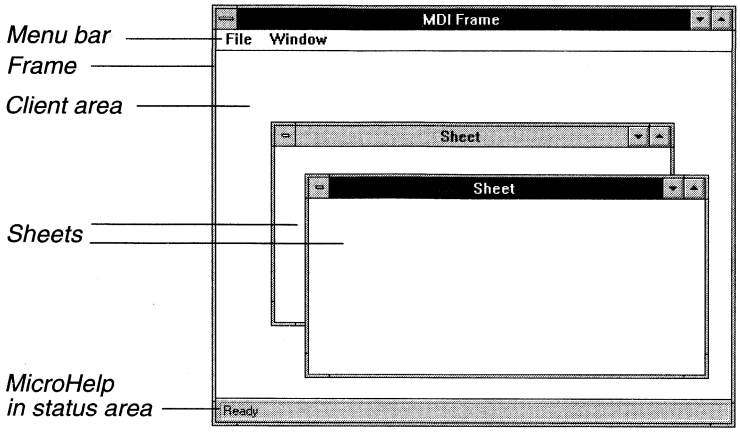


If you expect your users to want to be able to open several windows and easily move from window to window, you should make your application an MDI application.

The rest of this section describes the parts of an MDI application.

About 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).



MDI on the Macintosh
On the Macintosh, the MDI frame window is not visible as it is in Microsoft Windows. In Windows, the menu is at the top of the frame window and the sheets are clipped within the frame. On the Macintosh, the frame window is equivalent to the desktop and is not visible as a separate entity. The menu displays across the top of the desktop and the sheets are clipped within the desktop.

About 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 in an MDI application typically has an item that lists all open sheets and another item that the user can use to 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, that you add to meet the needs of the application.

For example, you might want to add a set of buttons with some explanatory text in the client area.

Don't use custom frames on the Macintosh
On the Macintosh, the frame window is equivalent to the desktop and is not visible as a separate entity. So you should not use custom frames if you want to deploy your application to the Macintosh: The objects you added to the client area will not display.

Managing the 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.

Providing a toolbar
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 more information on providing a toolbar, see "Providing toolbars" on page 255.

For more information on sizing the client area, see "Sizing the client area" on page 262.

About the client area

The client area is the area within the MDI frame in which open sheets display. In a standard MDI frame, PowerBuilder sizes the client area automatically so it fills the space inside the frame. For example, if the frame has a menu bar and MicroHelp, the client area fills the space between the sides of the frame and the space below the menu bar and above the MicroHelp.

In a custom MDI frame window, you determine the size of the client area. For example, if the frame has buttons below the frame menu bar, you size the client area so it begins below the buttons.

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 attributes and related functions for MDI_1 in the Object browser: open the browser from within the PowerScript painter and double-click the MDI frame window. The browser lists MDI_1 as one of the controls. Select MDI_1 and click Attributes or Functions as the Paste Category to see the information.

About 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 the OpenSheet function.

Building an MDI frame window

❖ To build an MDI frame window:

1 Open the Window painter.

The Select Window window displays.

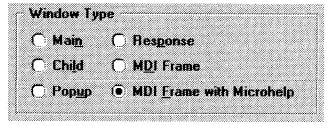
2 Click the New button.

The Window Painter workspace displays.

3 Double-click in the window or select Design➤Window Style from the menu bar.

The Window Style dialog box displays.

4 Select the type of MDI frame you want to create: MDI Frame or MDI Frame with MicroHelp.



5 Select the style options you want for the MDI frame.

6 Click the Menu checkbox and specify the menu you want to associate a menu with the frame.

> **Tip**
> Use the Menu painter to build the menu.

7 Click OK to return to the Window painter workspace.

8 Build scripts to open the sheets in the frame and to perform other processing as appropriate. The scripts can be triggered by events in the frame or in the MenuItems in the menu associated with the frame.

> **Building a custom frame**
> If you are adding objects to the client area (that is, building a custom frame), you must size the client area in the script for the Resize event in the frame window.

9 Save and run the MDI window.

Using menus

When you build an MDI frame window, you must associate a menu with the frame. It is good practice to include items in the menu to open sheets in the frame. Also, a menu provides a nice way to close the frame if the user has closed all the sheets.

About menus and sheets

A sheet can have its own menu, but it 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 the users open to create and send messages, read messages they have received, and reply to these messages. All sheets can be open at the same time and the user can move among the sheets, performing different activities in each sheet.

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 MenuItem, an event in another sheet, or an event in any object in the frame.

OpenSheet has this syntax:

OpenSheet (*sheet*, {, *window* }, *mdiframe* {, *position* {, *arrangeopen* } })

Parameter	Description
sheet	The name of the window you want to open as a sheet in the frame. It can be any window type except an MDI frame.
window	(Optional) A string containing the name of the window you want to open.
mdiframe	The name of the MDI frame in which you are opening the sheet.
position	(Optional) The number of the menu bar item to which you want to append the names of the open sheets. Menu bar items are numbered from the left, beginning with 1. The default is to list the open sheets under the next-to-last menu item (typically this is the Window menu, which is to the left of Help, which is the last item). For more information, see "Listing open sheets" on page 250.

Parameter	Description
arrangeopen	(Optional) An enumerated data type specifying how you want the sheets arranged in the MDI frame when they are opened: Cascade!—Window is displayed with the other open sheets. The sheets are arranged so they overlap and all the title bars are visible. Layered!—Window is displayed the size of the workspace. Original!—Window is displayed the size it was built.

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

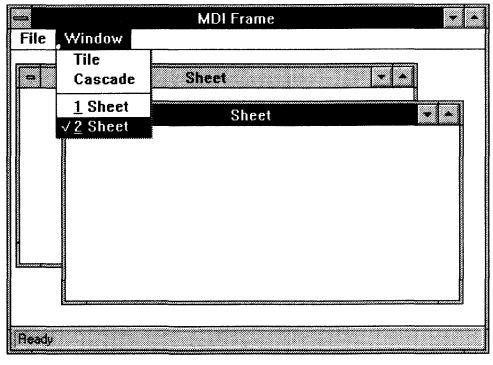
Opening instances of windows

Typically in an MDI application, you will allow your 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).

For information about opening instances of windows, see the *User's Guide*.

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 dropdown menu. For example, this menu lists two open sheets.



❖ **To list open sheets in a dropdown menu:**

- Specify the number of the menu bar item (the leftmost item is 1) in which you want the open sheets listed when you call the OpenSheet function to open each sheet.

If more than nine sheets are open at one time, the first nine sheets are listed in the menu and More Windows displays in the tenth position. When you click More Windows, the Select Window lists the open sheets so users can select the sheet they want to activate.

Arranging sheets

After you open sheets in an MDI frame, you can change the way the sheets are arranged in the frame. To arrange the sheets, use the ArrangeSheets function.

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

For more information about ArrangeSheets, see the *Function Reference*.

Maximizing sheets

If sheets opened in an MDI window have a control menu, users can maximize the sheets. This is what happens 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

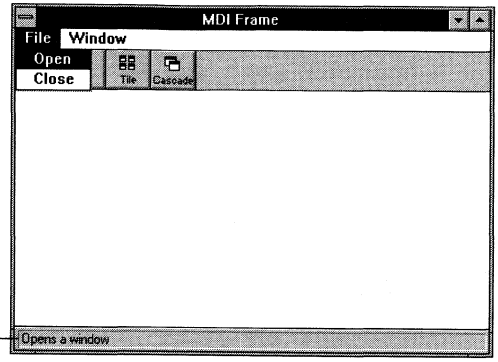
To close the active window (sheet), users can press CTRL+F4. In addition, you can write a script for a MenuItem that closes the parent window of the menu (make sure the menu is associated with the sheet, not the frame). For example:

```
Close(ParentWindow)
```

To close all the sheets, the users can press ALT+F4. In addition, 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.

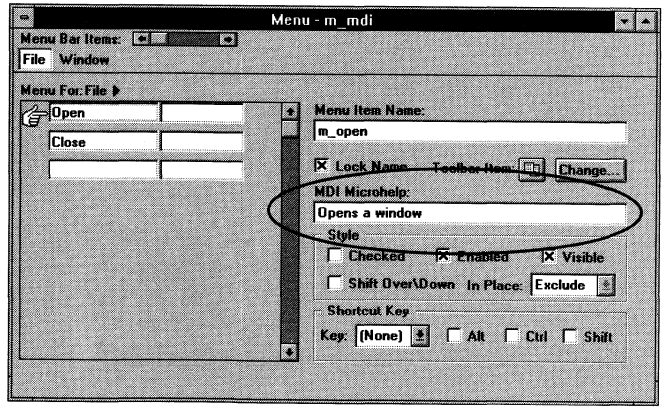


MicroHelp for selected MenuItem

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

Providing MicroHelp for MenuItems

You specify the text for the MicroHelp associated with a MenuItem for an MDI frame or sheet in the Menu painter.



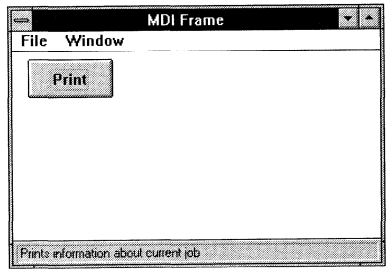
To change the text of the MicroHelp in a script for a MenuItem, use the SetMicroHelp function.

For information about SetMicroHelp, see the *Function Reference*.

Providing MicroHelp for controls

You can associate MicroHelp with a control in a custom frame window by using the control's Tag attribute. 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 attribute to the desired text, then uses SetMicroHelp to display the text. For example:

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



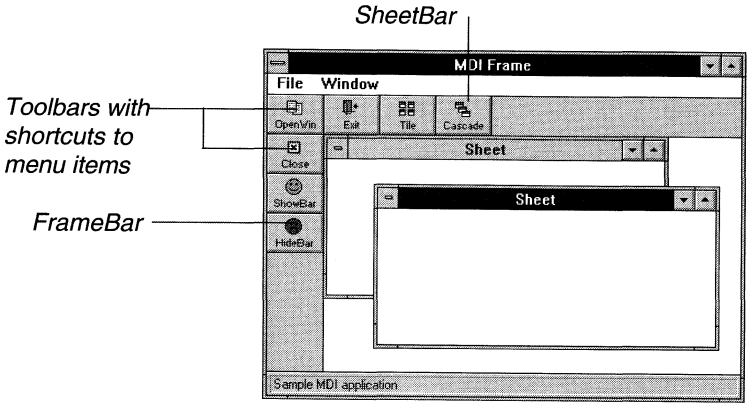
(You can also set a control's Tag attribute in the Tag window.)

In the LoseFocus event, you should restore the MicroHelp, such as:

```
w_frame.SetMicroHelp("Ready")
```

Providing toolbars

To make your MDI application easy to use, you might want to add toolbars with buttons that users can click as a shortcut for choosing an item from a menu. You can associate a toolbar with the MDI frame and a toolbar with the active sheet.



PowerBuilder provides an easy way for you to add toolbars to an MDI application: When you are defining a MenuItem in the Menu painter for a menu that will be associated with either an MDI frame window or an MDI sheet, you simply specify that you want the MenuItem to display in the toolbar with a specific picture. During execution, PowerBuilder automatically generates a toolbar for the MDI frame window or sheet containing the menu. The toolbar associated with the MDI frame is called the FrameBar. The toolbar associated with the active sheet is called the SheetBar.

The toolbars work the same as the PowerBuilder toolbars. Your users can select items from a toolbar, choose to display or not display text in the toolbar, use PowerTips, move the toolbar around the frame, make the toolbar floating, and so on. And all without your writing a line of code.

If you provide toolbars this way, PowerBuilder automatically manages the client area of the MDI window, so you don't need to resize the client area based on the toolbar.

❖ To add a toolbar to an MDI window:

1 In the Menu painter, associate pictures with the MenuItems you want to display in the toolbar.

2 In the Window painter, associate the menu with the window and turn on the display of the toolbar.

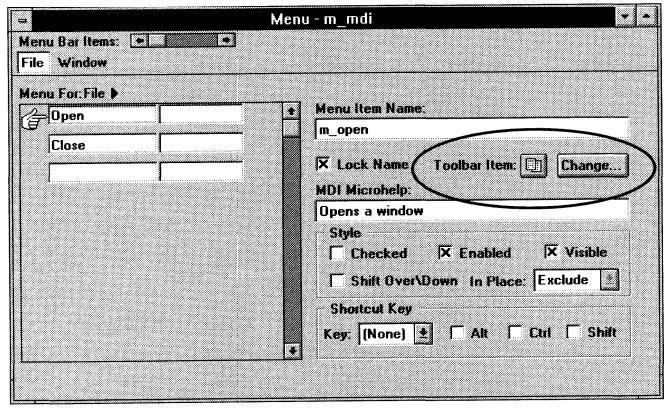
3 Specify toolbar-related attributes of the application object as desired.

Working in the Menu painter

In the Menu painter, you specify which MenuItems you want to display in the toolbar and which pictures to use to represent the MenuItems.

❖ To associate a toolbar picture with a MenuItem:

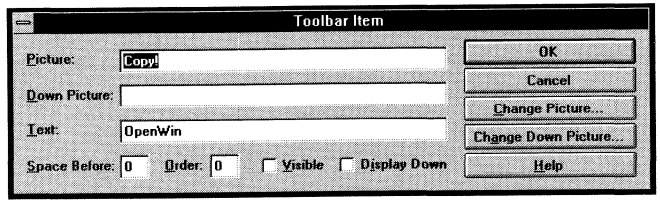
1 Open the menu in the Menu painter.



2 Select the MenuItem you want to display in the toolbar.

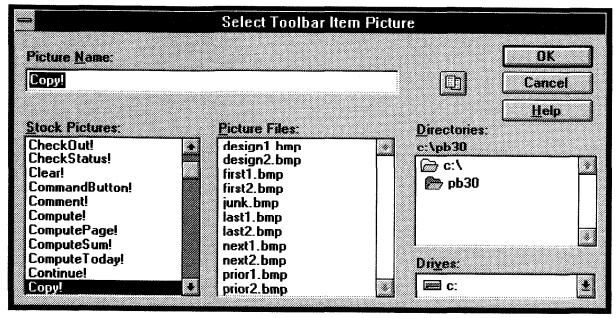
3 Click the Change button.

The Toolbar Item dialog box displays.



4 To specify the picture displayed in the toolbar, click Change Picture.

The Select Toolbar Item Picture dialog box displays.



5 Specify the picture to display in the toolbar. You can choose a stock picture or a BMP, RLE, or WMF file. If you choose a stock picture, PowerBuilder uses the up version when the item is not clicked and the down version when the item is clicked. If you are specifying a file, the picture should be 16 pixels wide and 15 pixels high.

6 If you chose a file as the picture and want a different picture to display when the item is clicked, click Change Down Picture and specify the picture.

7 In the Text box in the Toolbar Item dialog box, specify the text to display for the toolbar item (see "Specifying text" below).

8 If you want to leave space before the picture in the toolbar, specify a value in the Space Before box. You can specify any integer in this box. Experiment with values to get the spacing the way you want it. If you leave the value 0, there will be no spacing before the picture. (Note that spacing is used only when the toolbar is not displaying text.)

9 In the Order box, specify the order in which the MenuItem displays in the toolbar. If you leave the value 0, PowerBuilder places the MenuItem in the order in which it goes through the menu to build the toolbar.

10 If you want the item to stay down until the user performs an action, click the Display Down checkbox. The item stays down after it is selected until you set it up in a script (set the MenuItem's ToolbarItemDown attribute to FALSE).

11 (Optional) Preview the menu.

The currently defined toolbar displays below the menu.

12 Save the menu.

Specifying text

In the Text box in the Toolbar Item dialog box, you can specify two pieces of text:

- Text that displays when text is displayed in the toolbar
- Text that displays for PowerTips when text is not displayed in the toolbar

You specify the text with a comma-delimited string in the Text box, as follows:

Text in button, PowerTip

For example, the following string specifies that *New* displays on the toolbar button and *New Customer* displays as the PowerTip.

```
New, New Customer
```

Working in the Window painter

In the Window painter, you associate the menu with the MDI frame window and specify that the window should display the toolbar.

❖ To use the toolbar in a window:

1. Open the MDI frame window to contain the toolbar.
2. Associate the menu with the window.
3. Specify that the window should display the toolbar by coding the following line (typically in the window's Open script):

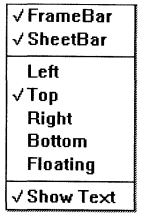
```
ToolbarVisible = TRUE
```

Setting application attributes

You can specify properties for the toolbars by setting the following attributes of the application object (typically you set these in the application's Open event, but you can set them anywhere).

Attribute	Meaning
ToolbarText	(Boolean) If TRUE, text displays in the buttons. If FALSE, text does not display.
ToolbarTips	(Boolean) If TRUE, PowerTips display when text is not displayed in the buttons. If FALSE, PowerTips do not display.
ToolbarUserControl	(Boolean) If TRUE, users can use the toolbar popup menu to hide or show the toolbars, move toolbars, or show text. If FALSE, users cannot manipulate the toolbar.
ToolbarFrameTitle	(String) The text that displays as the title for the FrameBar when it is floating
ToolbarSheetTitle	(String) The text that displays as the title for the SheetBar when it is floating
ToolbarPopMenuText	(String) Text to display on the popup menu for toolbars (see below)

Specifying the text in the toolbar's popup menu



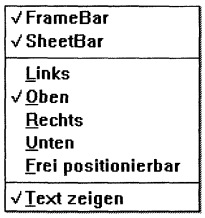
By default, PowerBuilder provides a popup menu for the toolbar, which users can use to manipulate the toolbar (it is similar to the popup menu you use to manipulate the PowerBar and PainterBar).

You can change the text that displays in this menu (but you cannot change the functionality of the MenuItems in the menu). Typically, you will do this when you are building an application in a language other than English. You change the text as follows:

The first two items in the popup menu display the titles set in ToolbarFrameTitle and ToolbarSheetTitle (defaults: FrameBar and SheetBar).

The remaining text items are specified by the attribute ToolbarPopMenuText: To specify values for this attribute, use a comma-delimited list of values to replace the text "Left," "Top," "Right," "Bottom," "Floating," and "Show Text."

```
toolbarPopMenuText = "left, top, right, bottom, floating, showText"
```



For example, to change the text for the toolbar popup menu to German and have hot keys underlined for each, you would specify the following:

```
toolbarPopMenuText = "&Links,&Oben,&Rechts,"  &
+"&Unten,&Frei positionierbar,&Text zeigen"
```

What happens during execution

When the user runs your application, the toolbars display as you defined them. Users can use them the same way you use the PowerBar and PainterBars. Clicking a picture is the same as choosing the corresponding MenuItem.

You can manipulate the toolbars in scripts during execution by setting values for the toolbar-related application attributes, as described above.

Testing for whether a toolbar is moved

When the user moves the FrameBar or SheetBar, the ToolbarMoved event is triggered for the MDI frame window and the Message.WordParm and Message.LongParm attributes are populated with values, as shown below.

Message.WordParm value	Meaning
0	FrameBar moved
1	SheetBar moved

Message.LongParm value	Meaning
0	Moved to left
1	Moved to top
2	Moved to right
3	Moved to bottom
4	Set to floating

Using toolbars and menus in MDI applications

This section describes how toolbars and menus behave in an MDI application.

The basics

- Toolbar buttons map directly to menu items. Clicking a toolbar button is the same as clicking its corresponding menu item (or pressing the accelerator key for that item).
- Toolbars work only in MDI frame and MDI sheet windows. If you open a non-MDI window with a menu that has a toolbar, the toolbar will not show.
- If both the sheet and the frame have toolbars, and the sheet is open, then the menu that is displayed will be the menu for the sheet, but both toolbars will appear and be operative.
- If the currently active sheet does not have a menu, then the menu and toolbar for the frame remains in place and operative. This can be confusing to your user, because the displayed menu will not be for the active sheet.

> **Tip**
> If any sheet has a menu, then all sheets should have a menu.

Disabling and hiding menu items

- Disabling a menu item will disable its toolbar button as well, but will not change the appearance of the button. You can programmatically make the button look disabled.
- Hiding a menu item does not cause dropdown or cascading menu item toolbar buttons to disappear; however, it does disable them. You can programmatically make the button look disabled.
- Hiding an item on a dropdown or cascading menu does not cause its toolbar button to disappear or to be disabled. You can programmatically make the button look disabled.

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 automatically sizes the client area if you have defined a toolbar based on MenuItems, 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, so you must size it. If you do not size the client area, the sheets will open, but may not be visible and will be clipped if they exceed the size of the client area.

> **Tip**
> If you selected HScrollBar and VScrollBar when defining the window, scroll bars display when a sheet is clipped to indicate that not all the sheet is displayed and to allow the user to scroll to display all the information in the sheet.

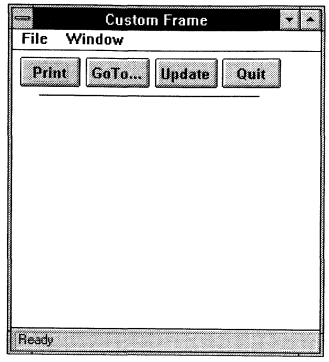
When you create a custom MDI frame window, PowerBuilder creates a control named MDI_1. PowerBuilder uses MDI_1 to identify the client area of the frame. MDI_1 displays in the list of objects in the PowerScript painter Paste Objects dropdown listbox when you create a script for the frame.

❖ 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

Example

For example, the following script sizes the client area for the frame w_mdi_custom2. The frame has a series of buttons running across the frame just below the menu, and it has MicroHelp at the bottom.




```
int     nWidth, nHeight
// Obtain the width and height of the workspace.
nWidth  = w_mdi_custom2.WorkSpaceWidth( )
nHeight = w_mdi_custom2.WorkSpaceHeight( )
// Calculate workspace between the
// CommandButtons and the MicroHelp.
// cb_print is the CommandButton nearest the
// top left-corner of the frame.
nHeight = nHeight - (cb_print.y + cb_print.height)
nHeight = nHeight - MDI_1.MicroHelpHeight
// Move the upper-left corner of the client area
// (MDI_1) to the bottom-left corner of the
// CommandButton (cb_print).
MDI_1.Move (0, cb_print.y + cb_print.height)
// Resize the client area. The frame window has
// a line under the buttons at the top of the
// window so we added 4 to nHeight to accommodate
// the line.
MDI_1.Resize (nWidth, nHeight + 4)
```

About MicroHelpHeight

MicroHelpHeight is an attribute 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.

For more information about the PowerScript functions used in the preceding script, see the *Function Reference*.

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:

Arrow 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 dropdown or cascading menu	The menu item below the current item
	The last menu item in the dropdown or cascading menu	The first item in the menu
Up	A dropdown or cascading menu	The menu item above the current item
	The first menu item in a dropdown or cascading menu	The last item in the menu

Shortcut keys

PowerBuilder automatically assigns every MDI frame two shortcut keys:

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

CHAPTER 7

Using Drag and Drop in a Window

About this chapter

This chapter describes how to make applications graphical by dragging and dropping controls.

Contents

Overview

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 (that is, 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 attributes

Each PowerBuilder control has two drag and drop attributes:

- DragAuto
- DragIcon

The DragAuto attribute

DragAuto is a boolean attribute:

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.

For information about the Drag function, see the *Function Reference*.

❖ To specify automatic drag mode for a control in the Window painter:

1 Click the right mouse button on the control.

2 Select Drag Auto from the Drag and Drop item on the popup menu.

The DragIcon attribute

Use the DragIcon attribute to specify the icon you want displayed when the control is in drag mode. The DragIcon attribute 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 Click the right mouse button on the control.

2 Select Drag Icon from the Drag and Drop item on the popup menu.

The Select Drag Icon window displays.

3 Choose the icon you want to use from the list of stock icons or from an ICO file.

4 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 four drag and drop events:

Event	Occurs
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:

Function	Action
Drag	Starts or ends the dragging of a control
DraggedObject	Returns the control being dragged

For more information about these functions, see the *Function Reference*.

Identifying the dragged control

To identify the type of control that was dropped, use the DraggedObject function and a variable whose data type is DragObject.

Example

The following script for the DragDrop event in a picture declares three variables, then determines the type of object that was dropped:

```
DragObject     Source
CommandButton  Button
StaticText     Info

Source = DraggedObject( )
if TypeOf(Source) = CommandButton! then
   Button = Source
   Button.Text = "You dropped a Button!"
elseif TypeOf(Source) = StaticText! then
   Info = Source
   Info.Text = "You dropped the text!"
End if
```

Tip
If your window has a large number of controls that can be dropped, use a CHOOSE CASE statement.

CHAPTER 8

Providing Online Help for an Application

About this chapter

You will probably want to document your work. This chapter describes how you can:

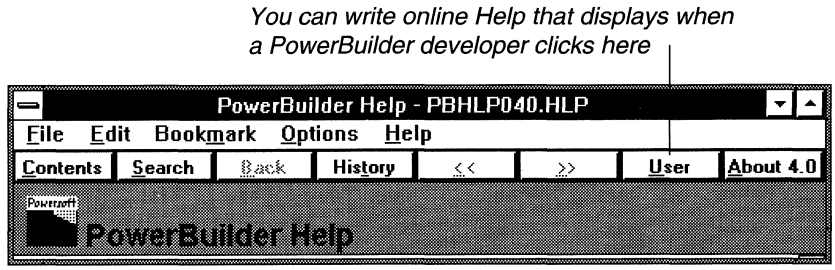
- Provide online Help for other PowerBuilder developers
- Provide online Help for your end users

Contents

Providing online Help for developers

It is a good idea to document the application components you build that other PowerBuilder developers will use. For example, you might create user-defined functions, user events, and user objects for use in several applications. By creating online Help for these components, you can ensure that other developers understand the components and use them correctly.

In the PowerBuilder online Help window, there is a button labeled User. You can create your own online Help that displays when you or another PowerBuilder developer click this button.



❖ To create online Help for PowerBuilder developers:

1 Create the Help text following the directions in the Microsoft Windows Software Developer's Kit (SDK).

At lease one topic in your Help file must have the context ID **index_user_help**. Normally, you assign this context ID to the Contents topic in your Help file. This is the Help topic ID associated with the User button in the Help window. When you click the User button, WINHELP.EXE looks for the Help topic with the index_user_help context ID.

> **Creating Help files**
> There are several products on the market that you can use to create Windows-based Help. If you are using one of these products, follow their directions for creating and compiling the Help file, and then save the compiled Help file in your PowerBuilder directory.

2 Save the Help file in Rich Text Format (RTF) using any number of authoring tools that have this capability.

3 Create a project file (an HPJ file) and save it in text format as PBUSR040.HPJ.

   A sample project file is provided in PowerBuilder online Help. To access it:

   - Open PowerBuilder Help and click the User button.
   - Search on the phrase "project file."
   - Copy the Help topic to the Windows clipboard.

4 Compile the Help using the Windows 3.x Help compiler, which is available in the Windows SDK or as part of a number of software development toolkits, compilers, and environments.

5 Rename the PBUSR040.HLP file that was installed with PowerBuilder.

6 Save the compiled Help file in your PowerBuilder directory. Make sure your Help file is named PBUSR040.HLP.

   Your Help text 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, prefix the name of the function with uf_ (for example, uf_calculate).

2 For each user-defined function Help topic, assign a search keyword (a K footnote entry) that corresponds 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 directory.

What happens

When you select the name of a function or merely place the cursor in the function name in the PowerScript painter and press SHIFT+F1:

1 PowerBuilder looks for the prefix uf_ in the function name.

2 If the prefix uf_ is found, PowerBuilder reads the UserHelpFile variable in PB.INI.

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 PBUSR040.HLP file in the PowerBuilder directory.

The UserHelpFile variable in PB.INI

You can specify a different filename for context-sensitive Help by naming the file with the UserHelpFile variable in your PB.INI file. To use the variable, your Help file must be in the PowerBuilder directory. The format of the variable is as follows:

UserHelpFile=*helpfile.hlp*

Specify only the filename. A full pathname designation will not be recognized.

The UserHelpPrefix variable in PB.INI

You can prefix your user-defined functions with something other than the default uf_ prefix by defining the prefix you want to use in the PB.INI file with the UserHelpPrefix variable. To do so, you enter the UserHelpPrefix variable in the [PB] section of PB.INI in the following format:

UserHelpPrefix=*yourprefix_*

The only requirement is that the prefix you provide must end with the underscore character, as shown above. Once you have specified a user-defined function prefix, when you place the cursor in your function name with the same prefix and press SHIFT+F1, the function Help topic corresponding to the function name you selected is displayed.

Providing online Help for your users

You can create a Microsoft Windows 3.x-based Help system to provide the users of your application with online information about your application.

❖ To provide online Help for your application:

1. Create the Help text following the directions in the Microsoft Windows Software Developer's Kit (SDK). If available, use a Winhelp authoring tool to help streamline the Help file development process.
2. Create a project file (an HPJ file) that contains the information the Windows 3.x Help compiler requires.

   A sample project file is provided in PowerBuilder online Help. To access it:

   - Open PowerBuilder Help and click the User button.
   - Search on the phrase "project file."
   - Copy the Help topic to the Windows clipboard.
3. Compile the project file using the Microsoft Help Compiler.

   The compiler saves the compiled Help file as an HLP file.
4. Use the PowerScript ShowHelp function to display the Help text in your application. Specify the compiled Help file (the HLP file) as the HelpFile parameter in the function.

The ShowHelp function

You can use the ShowHelp function to search for Help topics by Help context ID, by keyword, and by accessing the Help file Contents topic (the topic defined as the Help Contents page).

For more information

For complete information on building an online Help system, see the Microsoft SDK documentation as well as a number of commercially available products. For information about the ShowHelp function, see the *Function Reference* or PowerBuilder online Help.

PART THREE

Data Access Techniques

A collection of techniques you can use to implement data access features in the applications you develop with PowerBuilder. Includes: using transaction objects, using DataWindow objects, using dynamic DataWindow objects, piping data between data sources, and reading and writing text or binary files.

CHAPTER 9

Using Transaction Objects

About this chapter

This chapter describes transaction objects and how to use them in PowerBuilder applications.

Contents

About transaction objects

PowerBuilder transaction objects are the communication area between PowerScript and the database. Each transaction object has 15 attributes:

- Ten are used to connect to the database.
- Five are used to receive information from the database about the success or failure of each command that is executed. These error-checking attributes all begin with *SQL*.

Description of transaction object attributes

The following table describes each transaction object attribute.

Transaction object attributes for your DBMS

For the transaction object attributes that apply to your DBMS, see the table on pages 284 and 285.

For information about the values you should supply for each attribute, see the section for your DBMS in *Connecting to Your Database*.

Attribute	Data type	Description
DBMS	String	The database vendor identifier.
Database	String	The name of the database to which you are connecting.
UserID	String	The name or ID of the user who connects to the database.
DBPass	String	The password used to connect to the database.
Lock	String	The isolation level to use when you connect to the database. For information about the Lock values you can set for each DBMS, see *Connecting to Your Database*.
LogID	String	The name or ID of the user who logs on to the database server.

Attribute	Data type	Description
LogPass	String	The password used to log on to the database server.
ServerName	String	The name of the server on which the database resides.
AutoCommit	Boolean	Specifies whether you want to turn on or turn off normal recoverable transaction processing for those DBMSs that allow it. Values are: ♦ **True** Turns off normal recoverable transaction processing. PowerBuilder automatically commits each successful SQL statement as it occurs, and not as part of a transaction. When AutoCommit is set to True, you *cannot* issue a ROLLBACK to undo your changes. ♦ **False** (Default) Turns on normal recoverable transaction processing. For a description of the AutoCommit attribute, see *Connecting to Your Database*.
DBParm	String	Contains DBMS-specific parameters. For a description of each DBParm parameter that PowerBuilder supports, see *Connecting to Your Database*.
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 row inserted into the table.
SQLCode	Long	The success or failure code of the most recent SQL operation. Return codes are: ♦ **0** Success ♦ **100** Not Found ♦ **-1** Error (use SQLDBCode or SQLErrText to obtain the details)

Attribute	Data type	Description
SQLNRows	Long	The number of rows affected. The database vendor supplies this number, so the meaning may not be the same in every DBMS.
SQLDBCode	Long	The database vendor's error code.
SQLErrText	String	The database vendor's error message.

Transaction object attributes and supported DBMSs

The transaction object attributes required to connect to the database are different for each DBMS. The five attributes that return information about the success or failure of a SQL statement apply to all DBMSs.

The table on this page and the next lists the transaction object attributes and the DBMSs to which they apply. A checkmark indicates that an attribute applies to this DBMS.

Transaction object attributes	ALLBASE/ SQL	IBM DRDA	INFOR-MIX 4	INFOR-MIX 5	MDI Gateway	ODBC
DBMS	✓	✓	✓	✓	✓	✓
Database	✓	✓	✓	✓	✓ *	
UserID	✓	✓		✓		✓ #
DBPass	✓	✓		✓		
Lock	✓		✓	✓		✓
LogID					✓	✓ +
LogPass					✓	✓ +
ServerName				✓	✓	
AutoCommit					✓	✓
DBParm	✓	✓	✓	✓	✓	✓
SQLReturnData			✓	✓		✓
SQLCode	✓	✓	✓	✓	✓	✓
SQLNRows	✓	✓	✓	✓	✓	✓
SQLDBCode	✓	✓	✓	✓	✓	✓
SQLErrText	✓	✓	✓	✓	✓	✓

* Specify the Database attribute only to create a table in a database that is *not* the default database.

\# Optional for ODBC. Specify the UserID attribute with care, because it overrides the connection's UserName attribute returned by the ODBC SQLGetInfo call.

\+ PowerBuilder uses the LogID and LogPass attributes only if your ODBC driver does *not* support the ODBC SQL driver CONNECT call.

Transaction object attributes	Oracle	SQL Server	SQL Base	Sybase Net-Gateway	Sybase SQL Server System 10	XDB
DBMS	✓	✓	✓	✓	✓	✓
Database		✓	✓	✓ *	✓	✓
UserID			✓			✓
DBPass			✓			
Lock			✓			✓ #
LogID	✓	✓		✓	✓	✓
LogPass	✓	✓		✓	✓	✓
ServerName	✓	✓		✓	✓	
AutoCommit		✓			✓	
DBParm	✓	✓	✓	✓	✓	✓
SQLReturnData	✓					
SQLCode	✓	✓	✓	✓	✓	✓
SQLNRows	✓	✓	✓	✓	✓	✓
SQLDBCode	✓	✓	✓	✓	✓	✓
SQLErrText	✓	✓	✓	✓	✓	✓

* Specify the Database attribute only to create a table in a database that is *not* the default database.

\# The Lock attribute is supported in XDB multiuser databases only.

Using transaction objects

PowerBuilder uses a basic concept of database transaction processing called **logical unit of work** (LUW). LUW is synonymous with transaction. When an application executes a database statement, it always executes the statement within the boundaries of a transaction. There are four PowerScript transaction management statements:

- COMMIT
- CONNECT
- DISCONNECT
- ROLLBACK

Transaction basics

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.

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.

You can issue a COMMIT or ROLLBACK only if the AutoCommit attribute of the transaction object is FALSE. (For more about the AutoCommit attribute, see the table on page 283.)

Automatic commit when disconnected
By default, when a transaction is disconnected, a COMMIT is issued.

Before you issue a CONNECT statement, the transaction object must exist and you must assign values to all transaction object attributes required to connect to your DBMS.

The default transaction object

Since most applications communicate with only one database, PowerBuilder provides a global default transaction object called SQLCA, which stands for SQL Communications Area.

PowerBuilder creates this object before the application's Open event script executes. You can use PowerScript dot notation to reference it in any script in the application.

You can create additional transaction objects as you need them (for example, when you are using multiple database connections at the same time). However, in most cases SQLCA is the only transaction object you need.

Example

Here is a simple example that uses the default transaction object to connect to and disconnect from a database named Sample:

```
// Set the default transaction object attributes.
SQLCA.DBMS     = "ODBC"
SQLCA.dbParm   = "ConnectString='DSN=Sample'"
// Connect to the database.
CONNECT ;
if SQLCA.SQLCode < 0 then  &
   MessageBox("Connect Error", SQLCA.SQLErrText,&
   Exclamation!)
// Disconnect from the database.
DISCONNECT ;
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 are terminated with a semicolon (;). You do *not* use a continuation character for multiline SQL statements.

Specifying a transaction object

When a database statement requires a transaction object, PowerBuilder assumes the transaction object is SQLCA unless you specify otherwise.

Therefore, these CONNECT statements are equivalent:

```
CONNECT ;
CONNECT USING SQLCA ;
```

If you are using a transaction object *other* than SQLCA, you need to specify it in the following SQL statements:

- COMMIT
- CONNECT
- DELETE
- DECLARE *cursor*
- DECLARE *procedure*
- DISCONNECT
- EXECUTE (dynamic SQL)
- INSERT
- PREPARE (dynamic SQL)
- ROLLBACK
- SELECT
- SELECTBLOB
- UPDATEBLOB
- UPDATE

To include a transaction object in the above statements, add this clause at the end of the statements:

USING *TransactionObject*

For example, this statement uses a transaction object named WatcomTrans to connect to the database:

```
CONNECT USING WatcomTrans ;
```

Using multiple databases

To carry out operations in multiple databases at the same time, use multiple transaction objects, one for each database connection. You must declare and create the additional transaction objects before you reference 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.*

Example

The following statements use the default transaction object (SQLCA) to communicate with a Watcom SQL database and a transaction object named SQLServerTrans to communicate with a SQL Server database.

Assigning attribute values to your transaction object

When you assign values to attributes of a transaction object that you declare and create in a PowerBuilder script, you *must* assign the values *one attribute at a time*, like this:

```
// This code produces correct results.
transaction    SQLServerTrans
SQLServerTrans = CREATE TRANSACTION
SQLServerTrans.DBMS = "Sybase"
SQLServerTrans.database = "Personnel"
```

You *cannot* assign values by setting the nondefault transaction object equal to SQLCA, like this:

```
// This code produces incorrect results.
transaction    SQLServerTrans
SQLServerTrans = CREATE TRANSACTION
SQLServerTrans = SQLCA
```

```
// Declare the SQL Server transaction object.
transaction    SQLServerTrans

// Set the default transaction object attributes.
SQLCA.DBMS     = "ODBC"
SQLCA.dbParm   = "ConnectString='DSN=Sample'"
// Connect to the WATCOM database.
CONNECT ;

// Create the SQL Server transaction object.
SQLServerTrans = CREATE TRANSACTION
// Set the SQL Server transaction object attributes.
SQLServerTrans.DBMS =  "Sybase"
SQLServerTrans.database =  "Personnel"
SQLServerTrans.logid =  "JPL"
SQLServerTrans.logpass =  "JPLPASS"
SQLServerTrans.servername =  "SERVER2"
// Connect to the SQL Server database.
CONNECT USING SQLServerTrans ;

// Insert a row into the WATCOM database.
INSERT INTO CUSTOMER
   VALUES ( 'CUST789', 'BOSTON' ) ;
// Insert a row into the SQL Server database.
INSERT INTO EMPLOYEE
   VALUES ( "Peter Smith", "New York" )
   USING SQLServerTrans ;

// Disconnect from the WATCOM database.
DISCONNECT ;
// Disconnect from the SQL Server database.
DISCONNECT USING SQLServerTrans ;
```

```
// Destroy the SQL Server transaction object.
DESTROY SQLServerTrans
```

Using error checking

An actual script would include error checking after the CONNECT, INSERT, and DISCONNECT statements.

Using custom transaction objects in your application

SQLCA is a built-in global variable of type transaction that is used in all PowerBuilder applications. In your application, you may want to use a specialized version of SQLCA that performs certain processing or calculations on your data. If your database supports stored procedures, you may already have defined remote stored procedures to perform these operations.

Overview of the procedure

To call these database stored procedures from within your PowerBuilder application, you can define a customized version of the transaction object that performs the necessary processing. This technique involves the following general steps:

1 In the User Object painter, define a standard class user object inherited from the built-in transaction object.

2 In the User Object painter, declare the appropriate stored procedures as external functions 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 Use the user object in your PowerBuilder application.

For more information

For instructions on using the User Object, Application, and PowerScript painters in PowerBuilder, see the *User's Guide*.

Understanding the example

The following sections describe how to perform each of these steps by showing how to define and use a standard class user object named u_trans_database.

The u_trans_database user object is inherited from the built-in transaction object. It uses an Oracle 7 database stored procedure named GIVE_RAISE to calculate a five percent raise on the current salary. Here is the Oracle syntax to create the GIVE_RAISE database stored procedure:

TerminatorCharacter
The syntax shown here for creating an Oracle 7 stored procedure assumes that the SQL statement terminator character is ` (back quote).

```
// Create GIVE_RAISE function for Oracle7.
// Assumes TerminatorCharacter is ` (back quote).

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.

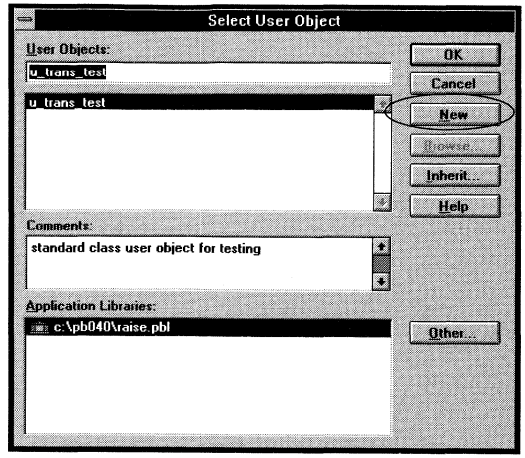
2 Connect to a database that supports stored procedures.

The rest of this procedure assumes you are connected to an Oracle 7 database that contains remote stored procedures on the database server.

For instructions on connecting to an Oracle 7 database in PowerBuilder and using Oracle 7 stored procedures, see *Connecting to Your Database*.

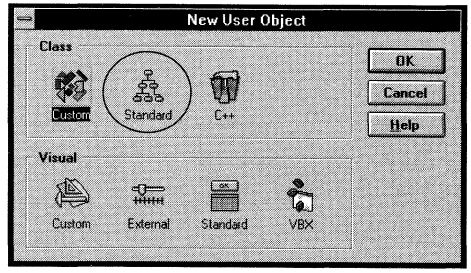
3 Open the User Object painter.

The Select User Object dialog box appears, listing the user objects in the current library.



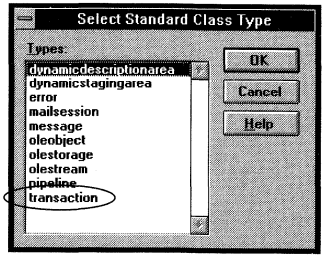
4 Click the New button.

The New User Object dialog box appears.



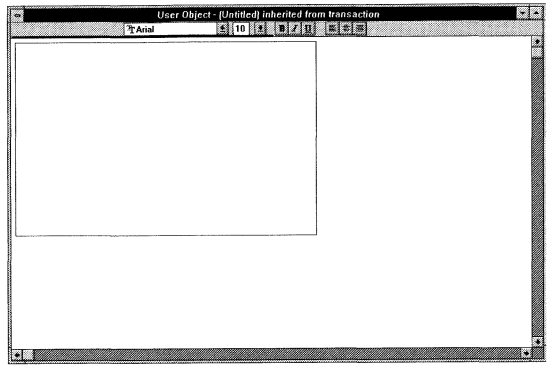
5 Select the Standard icon in the Class group box and click OK to define a new standard class user object.

The Select Standard Class Type dialog box appears.



6 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 you can assign attributes (instance variables) and functions to your user object.

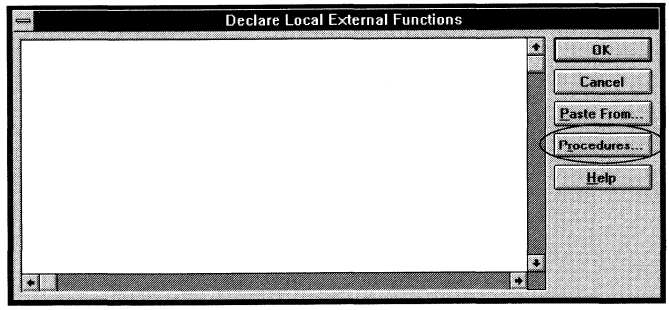


Step 2: declare stored procedures as external functions

❖ **To declare stored procedures as external functions for the user object:**

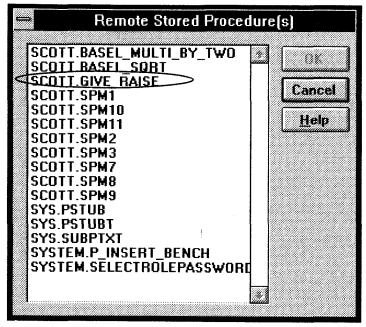
1 In the User Object painter, select Declare➢Local External Functions from the menu bar.

The Declare Local External Functions dialog box appears. If the database to which you are connected supports stored procedures, the dialog box contains a Procedures button so you can access the stored procedures.



2 Click the Procedures button to access the remote stored procedures for your database.

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.



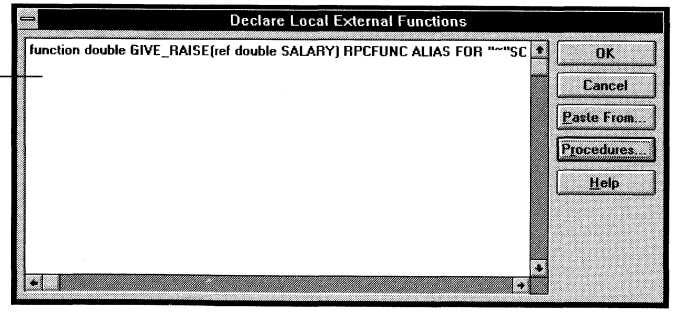
3 Select the names of one or more stored procedures that you want to declare as functions for the user object, and click OK.

PowerBuilder retrieves the stored procedure declarations from the database and pastes each declaration into the Declare Local External Functions dialog box.

For example, here is the declaration that appears on one line when you select SCOTT.GIVE_RAISE:

```
function double GIVE_RAISE(ref double SALARY)
RPCFUNC ALIAS FOR "~"SCOTT~".~"GIVE_RAISE~""
```

You can edit the syntax here



4 Edit the stored procedure declaration as needed for your application.

You can declare a database remote procedure call (RPC) as an external function or an external subroutine. The syntax for this is similar to the syntax for declaring external functions described in *PowerScript Language*.

You must use the keyword RPCFUNC to declare a database remote procedure call as an external function or subroutine. Optionally, you can use the expression ALIAS FOR *AliasName* to supply the name of the database remote procedure call.

The following syntax declares a database remote procedure call (RPC) as an external function:

```
FUNCTION ReturnDataType FunctionName
( {REF} {DataType1 Arg1, ..., DataTypeN ArgN} )
RPCFUNC {ALIAS FOR AliasName}
```

The following syntax declares a database remote procedure call as an external subroutine. Subroutines are the same as external functions, except that they do not return a value.

```
SUBROUTINE SubroutineName
( {REF} {DataType1 Arg1, ..., DataTypeN ArgN} )
RPCFUNC {ALIAS FOR AliasName}
```

For example, here is the edited RPC declaration for GIVE_RAISE:

```
function double GIVE_RAISE(ref double SALARY)
RPCFUNC ALIAS FOR "GIVE_RAISE"
```

5 When you finish editing the stored procedure declaration, click OK.

PowerBuilder returns you to the User Object painter workspace.

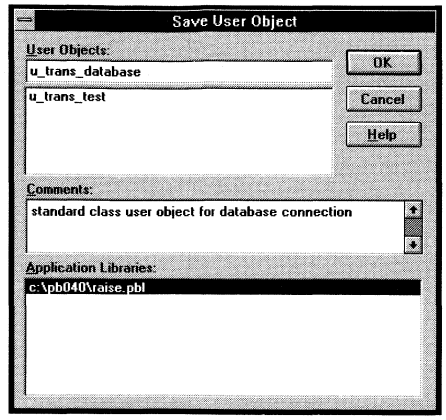
Step 3: save the user object

❖ **To save the user object:**

1 In the User Object painter, select File➤Save from the menu bar.

The Save User Object dialog box appears.

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 run 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. However, you can also declare and create an instance of your own transaction object in your application, and then write code that calls the user object as an attribute of your transaction object.
>
> For instructions, see the chapter on working with user objects in the *User's Guide*.

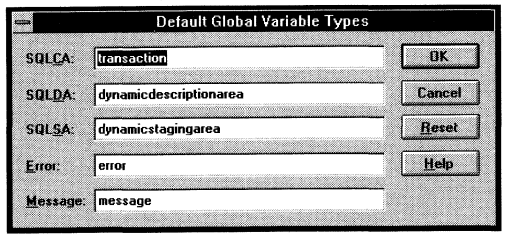
❖ To specify the default global variable type for SQLCA:

1 Open the Application painter.

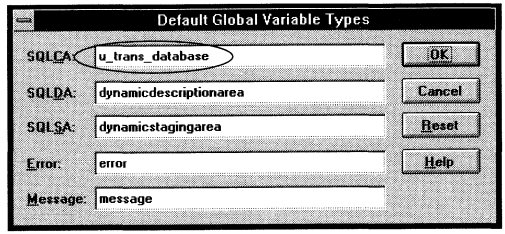
The Application painter workspace displays.

2 Select Edit➤Default Global Variables from the menu bar.

The Default Global Variable Types dialog box appears.



3 In the SQLCA field, specify the name of the standard class user object you defined in the previous steps.



4 Click OK.

When you run your application, PowerBuilder will use the specified standard class user object instead of the built-in system object type it inherits from.

Step 5: use the user object in your application

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 attributes and functions encapsulated in the user object. You now need to write code that uses the user object to perform the processing you want.

In your application script, you call the stored procedure functions you defined for the user object by using PowerScript dot notation, 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 use the user object in your application:

1 Select the object or control for which you want to write a script.

2 Open the PowerScript painter, and select the event for which you want to write the script.

For instructions on using the PowerScript painter, see the *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 7 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 attributes.
SQLCA.DBMS        =  "OR7"
SQLCA.LogID       =  "scott"
SQLCA.LogPass     =  "xxyyzz"
SQLCA.ServerName  =  "@t:oracle:testdb"
SQLCA.DBParm      =  "sqlcache=24,pbdbms=1"

// Connect to the Oracle 7 database.
CONNECT ;

// 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.
rv = SQLCA.give_raise(val)

// Display a message box with the new salary.
MessageBox("The new salary is",string(rv))

// Disconnect from the Oracle 7 database.
DISCONNECT ;
```

Uses dot notation to call GIVE_RAISE

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.

Supported DBMS features when using custom transaction objects

When you define and use a custom transaction object in your PowerBuilder 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 from PowerBuilder. Read the section for your DBMS to determine what you can and cannot do in a PowerBuilder application.

For all DBMSs
For all DBMSs, custom transaction objects support remote stored procedures that do *not* return result sets. If the stored procedure you want to use returns a result set, use the embedded SQL statement DECLARE Procedure.

For information about the DECLARE Procedure statement, see *PowerScript Language*.

INFORMIX

If your application connects to INFORMIX, you can use simple non-array data types. You *cannot* use Binary Large Objects (BLOBs).

ODBC

If your application connects to ODBC, you can use the following ODBC features if the back-end ODBC driver supports them. (For information, see the documentation for your ODBC driver.)

- IN, OUT, and IN OUT parameters, as follows:

  IN parameter An IN variable is passed by value and indicates a value being passed to the procedure.

OUT parameter An OUT variable is passed by reference and indicates that the procedure can modify the PowerScript variable that was passed. Use the PowerScript REFERENCE keyword for this parameter type.

IN OUT parameter 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 REFERENCE keyword for this parameter type.

- BLOBs as parameters

  You can use BLOBs that are up to 32,512 bytes long.

- Integer return codes

Oracle Version 7

If your application connects to Oracle 7, you can use the following Oracle PL/SQL features:

- IN, OUT, and IN OUT parameters, as follows:

  IN parameter An IN variable is passed by value and indicates a value being passed to the procedure.

  OUT parameter An OUT variable is passed by reference and indicates that the procedure can modify the PowerScript variable that was passed. Use the PowerScript REFERENCE keyword for this parameter type.

  IN OUT parameter 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 REFERENCE 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

SQL Server and Sybase SQL Server System 10

If your application connects to SQL Server or to Sybase SQL Server System 10, you can use the following Transact SQL features:

- IN, OUT, and IN OUT parameters, as follows:

  IN parameter An IN variable is passed by value and indicates a value being passed to the procedure.

  OUT parameter An OUT variable is passed by reference and indicates that the procedure can modify the PowerScript variable that was passed. Use the PowerScript REFERENCE keyword for this parameter type.

  IN OUT parameter 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 REFERENCE keyword for this parameter type.

- BLOBs as parameters

  You can use BLOBs that are up to 32,512 bytes long.

- Integer return codes

Watcom SQL

If your application connects to Watcom SQL, you can use the following Watcom SQL features:

- IN, OUT, and IN OUT parameters, as follows:

  IN parameter An IN variable is passed by value and indicates a value being passed to the procedure.

  OUT parameter An OUT variable is passed by reference and indicates that the procedure can modify the PowerScript variable that was passed. Use the PowerScript REFERENCE keyword for this parameter type.

  IN OUT parameter 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 REFERENCE keyword for this parameter type.

- BLOBs as parameters

  You can use BLOBs that are up to 32,512 bytes long.

CHAPTER 10

Using DataWindow Objects

About this chapter

This chapter describes how to use DataWindow objects in an application.

Contents

Before you begin

This chapter assumes you know how to build DataWindow objects in the DataWindow painter, as described in Part Four, "Working with Databases," in the *User's Guide*.

Overview

After you build a DataWindow object in the DataWindow painter, you can use it in a window to display and process information from the data source.

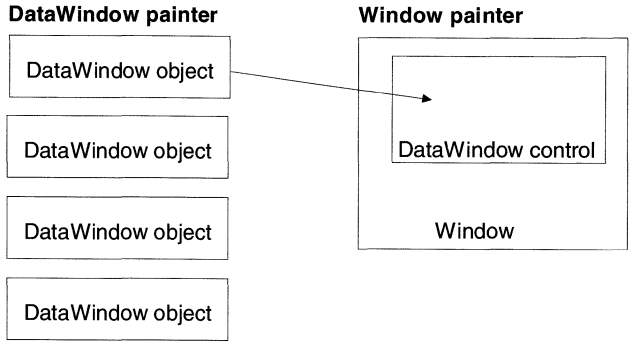
To use the DataWindow object, you place it in a DataWindow control in a window. You then write scripts to control the processing that is initiated when events occur in the window and its controls, including in the DataWindow control.

This chapter describes this process in detail.

Associating a DataWindow object with its control

In the DataWindow painter, you define a DataWindow object. You specify its data source and presentation style, then enhance the object by specifying display formats, edit styles, and so on.

To use the DataWindow object in an application, you place a **DataWindow control** in a window, then associate that control with a DataWindow object you created in the DataWindow painter.



Reusing DataWindow controls

You might want all the DataWindow controls in your application to have similar appearance and behavior. For example, you might want all of them to do the same error handling, which is defined in scripts for the DataWindow control.

To be able to define these properties once and reuse them in each window, you should create a standard user object based on the DataWindow control: define the user object's attributes and write scripts that perform the generic processing you want, such as error handling. Then place the user object instead of a new DataWindow control in the window. The DataWindow user object has all the desired functionality predefined. You don't need to respecify it.

For more information about user objects, see the *User's Guide*.

❖ To place a DataWindow control in a window:

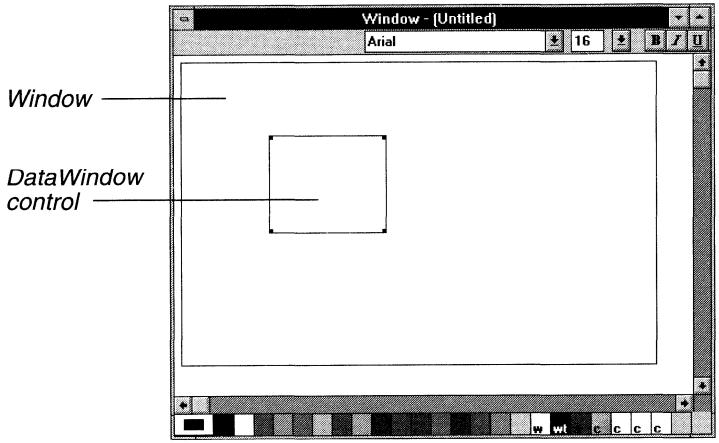
1 Open the Window painter.

2 Open the window that will contain a DataWindow control.

3 Click the DataWindow button in the PainterBar or select Controls➤DataWindow from the menu bar.

4 Click where you want the control to display.

PowerBuilder places an empty DataWindow control in the window.



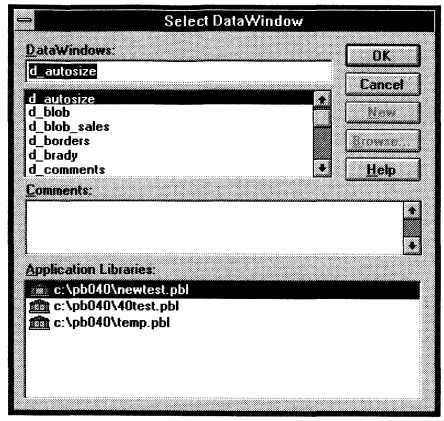
After placing the DataWindow control, you associate a DataWindow object with the control.

❖ To associate a DataWindow object with the control:

1 Move the pointer to the DataWindow control and display its popup menu.

2 Select Change DataWindow from the popup menu.

The Select DataWindow dialog box displays.



3 Select the DataWindow object that you want to place in the control and click OK.

The DataWindow object displays in the control (at this point, you see everything but the data).

Tip

You don't have to associate a DataWindow object with a DataWindow control in the Window painter. You can make (or change) the association in a script during execution.

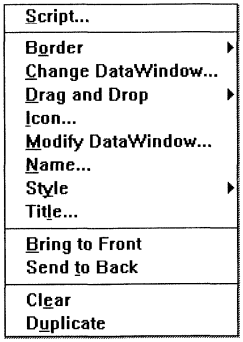
For more information, see "Changing the associated DataWindow object during execution" on page 312.

Resizing the control

You will probably want to resize the DataWindow control so that everything fits. You can resize the control by selecting it and dragging one of the handles.

Changing the attributes of the control

Once you have associated a DataWindow object with its control, you might want to change the attributes of the control.

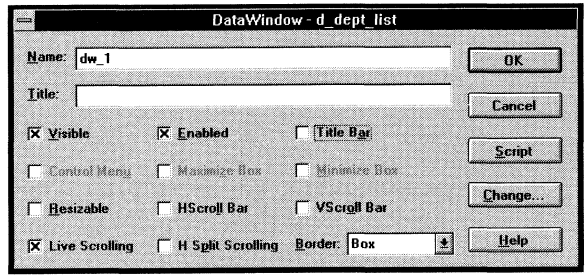


Like other controls, a DataWindow control has a popup menu that you can use to modify its appearance and behavior. You can change attributes directly from the menu or use the control's Style dialog box to change several attributes at once.

❖ To change the control's attributes using the Style dialog box:

1 Select Name from the DataWindow control's popup menu.

The control's Style dialog box displays (if you have not associated the control with a DataWindow object, the Select DataWindow dialog box displays instead).



2 Change the attributes as needed.

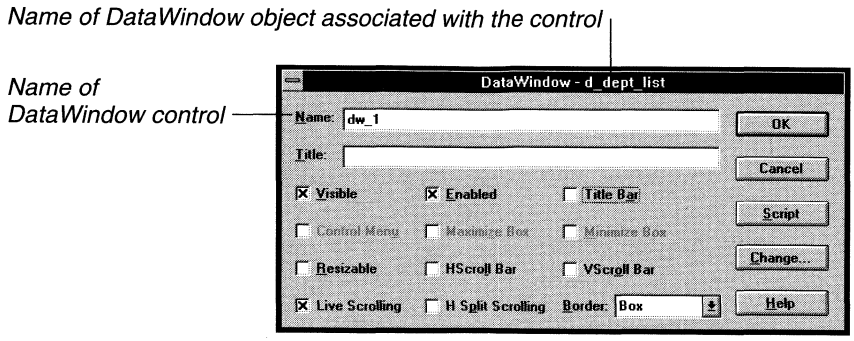
3 Click OK.

Allowing users to move DataWindow controls
If you want users to be able to move a DataWindow control during execution, give it a title. Then users can move the control by dragging it by the title bar.

About the names

There are two names to be aware of when you are working with a DataWindow:

- The name of the DataWindow control
- The name of the DataWindow object associated with the control



The DataWindow control name

When you place a DataWindow control in a window, PowerBuilder assigns it a default name, which is the concatenation of the default prefix for DataWindow controls (initially, dw) and the smallest integer that makes the name unique. You should change the suffix to a suffix that has meaning for the control in your application. For example, if the DataWindow control lists departments, you might want to name it dw_department.

> **Using the name**
> In scripts, you always reference a DataWindow by the *control's* name (such as dw_department). You don't refer to the DataWindow object that is in the control (such as d_dept_list).

The associated DataWindow object name

PowerBuilder displays the name of the DataWindow object associated with the control in the title bar of the control's Style window. In the preceding window, the DataWindow object d_dept_list was associated with the control dw_1.

Modifying the DataWindow object

Once you have associated a DataWindow object with a DataWindow control in a window, you can go from the Window painter directly to the DataWindow painter to modify the associated DataWindow object.

❖ To modify an associated DataWindow object:

- Select Modify DataWindow from the DataWindow control's popup menu.

  PowerBuilder opens the associated DataWindow object in the DataWindow painter.

Changing the associated DataWindow object during execution

When you associate a DataWindow object with a control in the Window painter, you are setting the initial value of the DataWindow control's DataObject attribute. During execution, PowerBuilder creates an instance of the DataWindow object specified in the control's DataObject attribute and uses it in the control. During execution, you can change the DataWindow object that displays in the window by changing the value of the DataObject attribute to the name of another DataWindow object.

> **Dynamically creating a DataWindow object**
> You can also create a new DataWindow object during execution and associate it with a control.
>
> For more information, see the next chapter.

Example

To display the DataWindow object d_emp_hist in the DataWindow control dw_emp, you can enter the following statement in a script:

```
dw_Emp.DataObject = "d_emp_hist"
```

The DataWindow object d_emp_hist was created in the DataWindow painter and stored in a library on the application search path. The control dw_emp is contained in the window and is saved with the window as part of the window definition.

Using PSR files

You can also display a PSR (Powersoft report) file in a DataWindow control by specifying its filename as the DataObject attribute.

For more information about PSR files, see the *User's Guide*.

Displaying data

Before you can display data in the DataWindow control, you must move the data specified in the data source into the DataWindow control.

If the data source is not a database

If the data for the DataWindow is not coming from a database (that is, the data source was defined as External in the DataWindow painter), you can use the following PowerScript functions to import data into the DataWindow control:

- ImportClipboard
- ImportFile
- ImportString

You can also get data into the DataWindow by using the SetItem DataWindow function to assign a value to a specific row and column in a DataWindow.

For more information about these functions, see the *Function Reference*.

If the data source is a database

If the data for the DataWindow is coming from a database (that is, the data source was defined as anything but External in the DataWindow painter), you need to communicate with the database to get the data, as described next.

Communicating with the database

An application goes through the following steps in communicating with a database.

❖ **To communicate with the database:**

1 Set the appropriate values for the transaction object.

2 Connect to the database.

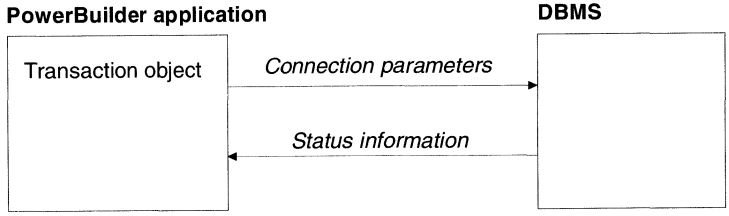
3 Assign the transaction object to the DataWindow control.

4 Do the database processing.

5 Disconnect from the database.

Using transaction objects

Database processing in PowerBuilder uses a special, nongraphic object called a **transaction object**. The transaction object specifies the parameters that PowerBuilder uses to connect to a database. The transaction object must be established before you can access a database through a DataWindow.



If displaying a PSR file in the control
You do not need to use a transaction object if you are displaying a PSR (Powersoft report) file in the DataWindow control.

When you start executing an application, PowerBuilder creates a global default transaction object named SQLCA. You can use this transaction object or create another transaction object for the DataWindow control in a script.

The default transaction object has 15 fields: 10 fields identify the database and server and five fields return status information from the database that indicates the success or failure of database processing.

For more information

Chapter 9, "Using Transaction Objects," contains complete information about transaction objects, including a description of what constitutes a transaction, a description of each field in a transaction object, and how you can customize your own transaction object to, among other things, support database remote procedure calls.

Setting values for the transaction object

Before you can use any transaction object, including the default transaction object, you must assign values to the transaction object database information fields. To assign the values, use dot notation.

Information required depends on your DBMS
The database information fields that are required depend on your DBMS.

For more information, see *Connecting to Your Database*.

Example

These statements assign values to the database information fields required to connect to a SQL Server database in the default transaction object (SQLCA):

```
sqlca.DBMS  = "Sybase"
sqlca.database = "testdb"
sqlca.LogId = "CKent"
sqlca.LogPass  = "superman"
sqlca.ServerName  = "Dill"
sqlca.AutoCommit  = FALSE
```

Reading values from an external file

Often you want to set the transaction object values from an external file, such as your PB.INI file when you are developing the application or an application-specific INI file when you distribute the application.

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. PB.INI 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, variable, defaultValue*)

For example, the following script reads values from the PB.INI file to set the transaction object to connect to a database:

```
SQLCA.DBMS     = ProfileString("PB.INI","Database","DBMS","")
SQLCA.Database= ProfileString("PB.INI","Database","DataBase","")
SQLCA.LogID    = ProfileString("PB.INI","Database","LogID","")
SQLCA.LogPass = ProfileString("PB.INI","Database","LogPassword","")
SQLCA.ServerName = ProfileString("PB.INI","Database","ServerName","")
SQLCA.UserID   = ProfileString("PB.INI","Database","UserID","")
SQLCA.DBPass   = ProfileString("PB.INI","Database","DatabasePassword","")
SQLCA.Lock     = ProfileString("PB.INI","Database","Lock","")
SQLCA.dbParm   = ProfileString("PB.INI","Database","dbParm","")
```

For more information about ProfileString, see the *Function Reference.*

Using multiple databases at one time

If you are using multiple databases at the same time, you can use SQLCA for the first database connection, but you must create a transaction object for each additional database connection.

To create a transaction object, you first declare a variable of type transaction, then you create the object:

transaction *TransactionObjectName*
TransactionObjectName = CREATE transaction

For example, to create a transaction object named DBTrans, code:

```
transaction DBTrans
DBTrans = CREATE transaction

// Now can assign values to DBTrans fields
DBTrans.DBMS = "ODBC"
```

Destroying the created transaction object

When you have finished using the transaction object you created, be sure to destroy it to reclaim memory:

```
DESTROY DBTrans
```

(You should not, however, destroy the default global transaction object SQLCA; PowerBuilder maintains that object automatically.)

Connecting to the database

Once you have established the connection parameters by assigning values to the transaction object, you can connect to the database. You use the SQL CONNECT statement to connect:

```
// 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 use the following syntax:

CONNECT USING *TransactionObject* ;

Disconnecting from the database

When your database processing has been completed, you disconnect from the database using the SQL DISCONNECT statement:

```
DISCONNECT ;
```

If you are using a transaction object other than SQLCA, you use the following syntax:

DISCONNECT USING *TransactionObject* ;

Error handling after a SQL statement

You should always test the success/failure code (the SQLCode attribute in 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

The SQLCodes are:

Value	Meaning
0	Success.
100	The command succeeded but did not retrieve or modify any rows (which may or may not be acceptable).
-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

For example, to display a message box containing the DBMS error number and message if the connection fails, code:

```
CONNECT;
if SQLCA.SQLCode = -1 then
   MessageBox("SQL error " + String(SQLCA.SQLDBCode), &
      SQLCA.SQLErrText )
end if
```

Note

Do *not* use this type of error checking following a retrieval or update made in a DataWindow.

For more information, see "Handling errors following retrieval or update in a DataWindow" on page 322.

Assigning the transaction object to the DataWindow control

To associate a transaction object with a DataWindow control, you use the SetTrans or SetTransObject PowerScript function in a script.

When changing the DataWindow object
If you change the DataWindow object associated with a DataWindow control during execution (by assigning a new value to the control's DataObject attribute), you need to reissue the SetTrans or SetTransObject function.

Using the SetTrans function

The SetTrans function copies the values from a specified transaction object to the DataWindow control's internal transaction object. When you use SetTrans in a script, the DataWindow control uses its own transaction object and *automatically* performs connects and disconnects as needed; any errors that occur cause an *automatic* rollback.

Whenever the DataWindow needs to access the database (such as when a Retrieve or Update function is executed), the DataWindow issues an internal CONNECT statement, then does the appropriate data access, then issues an internal DISCONNECT.

Connecting to the database
When you use SetTrans, you do not need to explicitly code a CONNECT or DISCONNECT statement in a script. CONNECT and DISCONNECT statements are automatically issued when needed.

SetTrans connects and disconnects from the database each time the database is accessed. In general, you should avoid using SetTrans because CONNECTs tend to be expensive and because you have no control over transaction management. However, if at your site you are limited by the number of available connections, you might want to use SetTrans for database activity. When the activity has completed, a DISCONNECT is automatically issued.

Use SetTrans in simple situations when you are doing pure retrievals (such as in reporting) and don't need to hold database locks—basically when transaction management is not an issue.

Using the SetTransObject function

The SetTransObject function tells the DataWindow control to share a transaction object in scripts. When you use SetTransObject, you have more control of the database processing and are responsible for managing the database transaction. When you use SetTransObject in your scripts, you should code the following statements:

- CONNECT
- SetTransObject
- Retrieve or Update
- COMMIT or ROLLBACK
- DISCONNECT

> **Tip**
> Application performance is usually better when you use SetTransObject rather than SetTrans.

For more information about transactions, see Chapter 9, "Using Transaction Objects."

For more information about SetTrans and SetTransObject, see the *Function Reference*.

Examples

The following statement uses SetTransObject to associate the DataWindow control dw_emp with the default transaction object, SQLCA:

```
dw_emp.SetTransObject(SQLCA)
```

The following statement uses SetTransObject to associate dw_emp with a programmer-created transaction object, Emp_TransObj:

```
dw_emp.SetTransObject(Emp_TransObj)
```

The following statement uses SetTrans to associate dw_emp with SQLCA:

```
dw_emp.SetTrans(SQLCA)
```

Retrieving and updating data

You call the following two PowerScript functions to access a database through a DataWindow:

- Retrieve
- Update

Retrieving data

After you have used SetTrans or SetTransObject to associate a DataWindow control with a transaction object, you can use the PowerScript Retrieve function to retrieve data from the database into the DataWindow control:

DataWindowControl.**Retrieve**( )

If the data source is a SQL SELECT statement with retrieval arguments, specify the values of the arguments in the Retrieve function:

DataWindowControl.**Retrieve**(*arg1, arg2,...*)

Updating data

After users have made changes to data in a DataWindow, you use the Update function to save the changes in the database. Update sends to the database all inserts, changes, and deletions made in the DataWindow since the last Update function:

DataWindowControl.**Update**( )

This section provides a simple example of retrieving data and updating the database and describes how to do error checking following a retrieval or update.

For more information about exactly how PowerBuilder updates the database (that is, which SQL statements are sent in which situations), see "How PowerBuilder updates the database" on page 329.

Example

The following statements retrieve and update data using the DataWindow control dw_Emp:

```
// Connect to the database specified in the
// transaction object EmpSQL.
CONNECT Using EmpSQL ;
// Set EmpSQL as the transaction object for dw_Emp.
dw_Emp.SetTransObject(EmpSQL)
// Retrieve data from the database specified in
// EmpSQL into dw_Emp.
dw_Emp.Retrieve( )
// Make changes to the data...
.
.
.
// Update the database.
If dw_Emp.Update( ) > 0 Then
   COMMIT Using EmpSQL ;
Else
   ROLLBACK Using EmpSQL ;
End If
DISCONNECT Using EmpSQL ;
```

Error handling

A production script would include error tests after each CONNECT, DISCONNECT, Retrieve, and Update:

For information about error handling after a SQL statement, see "Error handling after a SQL statement" on page 318.

For information about error handling after a DataWindow Retrieve or Update, see the next section.

Handling errors following retrieval or update in a DataWindow

When using Retrieve or Update in a DataWindow control, you should test the function's return code to see whether the activity succeeded.

For the Retrieve function

Return code	Meaning
≥1	Retrieval succeeded; returns the number of rows retrieved
-1	Retrieval failed; DBError event triggered
0	No data retrieved

For the Update function

Return code	Meaning
1	Update succeeded
-1	Update failed; DBError event triggered

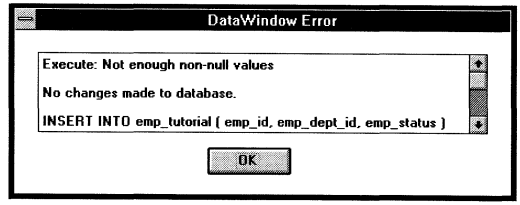
So if you want to commit changes to the database only if an update succeeded, code:

```
If dw_Emp.Update( ) > 0 Then
   COMMIT Using EmpSQL ;
Else
   ROLLBACK Using EmpSQL ;
End if
```

Using the DBError event

PowerBuilder triggers a DataWindow control's DBError event whenever there is an error following a retrieval or update in the DataWindow. For example, if you try to insert a row that doesn't have values for all columns that have been defined as not allowing NULL, the DBMS rejects the row and the DBError event is triggered.

By default, PowerBuilder displays a message box describing the error message from the DBMS.



If you don't want the default message box displayed (perhaps you want to process the information and present it in a way that is more meaningful to your user), you can do your own processing and set the action code to 1 in the DBError event. An action code of 1 in the DBError event tells PowerBuilder not to display the default message box.

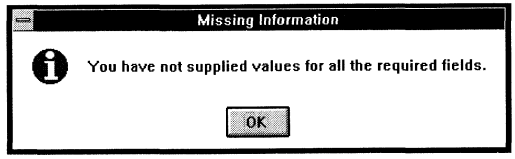
You can call the DataWindow-related functions DBErrorCode and DBErrorMessage to get the DBMS's error code and message text.

Warning
Don't use the transaction object attributes SQLCode, SQLDBCode, or SQLErrText in DataWindow control scripts. Those attributes are used only after SQL statements.

Here is a sample script for the DBError event:

```
// script for DataWindow control's DBError event

// Database errror 331 means not enough non-nulls.
if dw_emp.DBErrorCode( ) = 331 then
   MessageBox("Missing Information",  &
      "You have not supplied values for all "&
      +" the required fields.")
end if
// Do not display default message box.
dw_emp.SetActionCode(1)
```

During execution, your user would see the following message box after the error:



About action codes

As illustrated above, some events for DataWindow controls have action codes that you can set to override the default action that occurs after the event is triggered. The action code you set depends on the event. You use the SetActionCode function in a script to set the event's action code.

Manipulating data in a DataWindow control

Users can add, modify, and delete data in the DataWindow. You can write scripts to process the data. In order to do that, you need to understand how DataWindow controls store and process data and how to access the data.

How a DataWindow control stores data

As users add or change data, the data is first handled as text in an edit control. If the data is accepted, it is then stored as an item in a buffer.

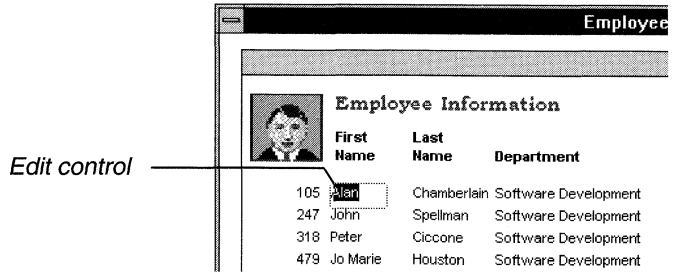
About the DataWindow buffers

A DataWindow uses three buffers to store data:

Buffer	Contents
Primary	Data that has not been deleted or filtered out (that is, the rows that are viewable)
Filter	Data that was filtered out
Delete	Data that was deleted by the user or in a script

About the edit control

As the user moves around the DataWindow, PowerBuilder places an **edit control** over the current cell (row and column).



About text

The contents of the edit control is called **text**. Text is data that has not yet been accepted by the DataWindow control. Data entered in the edit control is not in a DataWindow buffer yet; it is simply text in the edit control.

About items

When the user changes the contents of the edit control and presses ENTER or leaves the cell (by tabbing, using the mouse, or pressing UP ARROW or DOWN ARROW), the text is moved to the current row and column in the DataWindow Primary buffer and is referred to as an item. (This assumes that the data was accepted. For more information, see "How PowerBuilder processes entries" on page 327.)

Accessing the current text or a specified item

You can use the following functions to manipulate data in a DataWindow control:

- GetText—Obtains the text in the edit control
- SetText—Sets the text in the edit control
- GetItemDate, GetItemDateTime, GetItemDecimal, GetItemNumber, GetItemString, GetItemTime—Obtain the data in a specified row and column in a specified buffer in the DataWindow
- SetItem—Sets the value of a specified row and column

You call GetText in the script for the ItemChanged or ItemError event when the value in the edit control has not yet been accepted into the column.

For an example, see "Using the ItemChanged event" on page 328.

You call GetItemDate, GetItemDateTime, GetItemDecimal, GetItemNumber, GetItemString, or GetItemTime to obtain the data that has been accepted into a specific row and column. You can also use these functions to check the data in a specific buffer before you update the database.

Manipulating the contents of the edit control

The edit control in a DataWindow control is similar to a MultiLineEdit control in a window. Accordingly, you can use the following functions, which are also available for MultiLineEdit controls, to manipulate the contents of the edit control:

CanUndo	Scroll
Clear	SelectedLength
Copy	SelectedLine
Cut	SelectedStart
LineCount	SelectedText
Paste	SelectText
Position	TextLine
ReplaceText	Undo

For more information about these functions, see the *Function Reference*.

How PowerBuilder processes entries

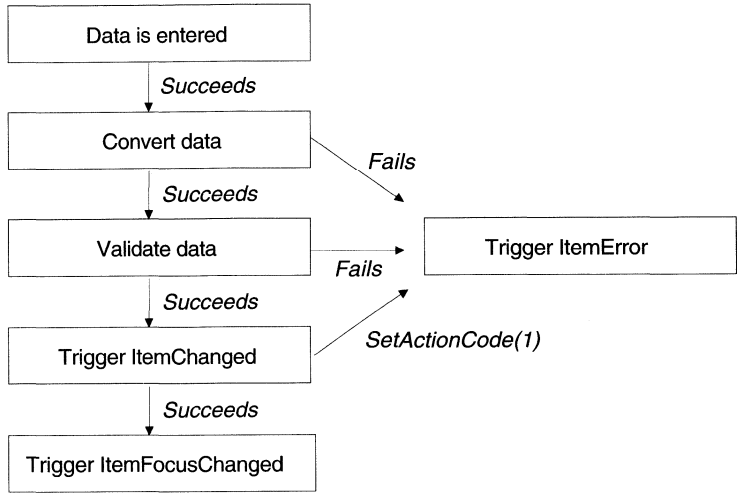
When the data in a column in a DataWindow has been changed and the column loses focus (for example, the user tabs to the next column), the following sequence of events occurs:

1 PowerBuilder converts the entry into the correct data type for the column. For example, if the user is in a numeric column, PowerBuilder converts the string that was entered into a number. If the data cannot be converted, the ItemError event is triggered.

2 If the data converts successfully to the correct type, PowerBuilder applies any validation rule used by the column. If the data fails validation, the ItemError event is triggered.

3 If the data passes validation, then the ItemChanged event is triggered.

4 The ItemFocusChanged event is triggered after the ItemChanged event.



Using the ItemChanged event

If data passes conversion and validation, the ItemChanged event is triggered. By default, the ItemChanged event accepts the data value and allows focus to change. You can write a script for the ItemChanged event to do some additional processing. For example, you could have some tests in the script and set the action code in the ItemChanged event to reject the data, have the column regain focus, and trigger the ItemError event.

For example, the following script for the ItemChanged event for dw_Employee sets the action code in dw_Employee to reject data that is less than the employee's age, which is in the sle_age control in the window:

```
int   a, age
age = Integer(sle_age.text)
a = Integer(dw_Employee.GetText( ))
// Setting the action code to 1 in the ItemChanged
// event tells PowerBuilder to reject the data
// and not change the focus.
If a < age then dw_Employee.SetActionCode(1)
```

Using the ItemError event

The ItemError event is triggered if there is some problem with the data. By default, the ItemError event rejects the data value and displays a message box. You can write a script for the ItemError event to do some other processing. For example, you can set an action code to accept the data value or reject the data value but allow focus to change.

For more information

For more information about these events and the action codes you can set for them, see *Objects and Controls*. For more information about SetActionCode, see the *Function Reference*.

How PowerBuilder updates the database

When updating the database, PowerBuilder determines the type of SQL statements to generate by looking at the **status** of each of the rows in the DataWindow buffers.

There are four DataWindow item statuses, two of which apply only to rows:

Status	Applies to
New!	Rows
NewModified!	Rows
NotModified!	Rows and columns
DataModified!	Rows and columns

How statuses are set

When data is retrieved

When data is retrieved into a DataWindow, all rows initially have a row status of NotModified!, and all columns initially have a column status of NotModified!.

After data has changed in a column in a particular row, either because the user changed the data or the data was changed programmatically, such as through the SetItem function, the column status for that column changes to DataModified!. Once the status for any column in a retrieved row changes to DataModifed!, the row status also changes to DataModified!.

When rows are inserted

When a row is inserted into a DataWindow, it initially has a row status of New!, and all columns in that row initially have a column status of NotModified!. After data has changed in a column in the row, either because the user changed the data or the data was changed programmatically, such as through the SetItem function, the column status changes to DataModified!. Once the status for any column in the inserted row changes to DataModified!, the row status changes to NewModified!.

Any columns in the newly inserted row that have default values defined are given an initial status of DataModified!, which changes the row status to DataModified! as well.

When Update is called

For rows in the primary and filtered buffers

When the DataWindow Update function is called, PowerBuilder generates SQL INSERT and UPDATE statements for rows in the primary and/or filtered buffers of the DataWindow based upon the following row statuses:

Row status	SQL statement generated
NewModified!	INSERT
DataModified!	UPDATE

A column is included in an UPDATE statement only if the following two conditions are met:

- The column is on the updateable column list maintained by the DataWindow.

  For more information about setting the update characteristics of the DataWindow, see the *User's Guide*.

- The column has a column status of DataModifed!

PowerBuilder includes all columns in INSERT statements it generates. If a column has no value, the DataWindow attempts to insert a NULL. This causes a database error if the database does not allow NULLs in that column.

For rows in the deleted buffer

PowerBuilder generates SQL DELETE statements for any rows that were moved into the deleted buffer of the DataWindow using the DeleteRow function. (However, if a row has a row status of New! or NewModified! before DeleteRow is called, then no DELETE statement is issued for that row.)

Changing row or column status programmatically

You might need to change the status of a row or column programmatically. Typically, you do this to prevent the default occurrence from taking place. For example, you might copy a row from one DataWindow to another. After the user modifies the row, you want to issue an UPDATE statement, instead of an INSERT statement.

You use the SetItemStatus function to programmatically change a DataWindow's row/column status information. Use the GetItemStatus function to determine the status of a specific row or column.

Changing column status

You simply use SetItemStatus to change the column status from DataModified! to NotModified!, or vice versa.

Changing row status

Changing row status is a little more complicated. The following table illustrates the effect of changing from one row status to another:

	Desired status			
Initial status	**New!**	**NewModified!**	**DataModified!**	**NotModified!**
New!	✓	✓	✓	*No change*
NewModified!	*No change*	✓	✓	New!
DataModified!	NewModified!	✓	✓	✓
NotModified!	✓	✓	✓	✓

In the preceding table, ✓ means the change is valid. For example, issuing SetItemStatus on a row that has the status NotModified! to change the status to New! does change the status to New!.

No change means that the change is not valid and the status is not changed.

Issuing SetItemStatus to change a row status from NewModified! to NotModified! actually changes the status to New!. Issuing SetItemStatus to change a row status from DataModified! to New! actually changes the status to NewModified!.

Changing a row's status to NotModified! or New! causes all columns in that row to be assigned a column status of NotModified!.

Changing status indirectly
When you cannot change to the desired status directly, you can usually do it indirectly. For example, change New! to DataModified! to NotModified!.

Using DataWindow functions

PowerScript provides many functions that you can use to perform activities in DataWindow controls. Here are some of the more commonly used:

Function	Purpose
AcceptText	Applies the contents of the edit control to the current item in the DataWindow
GetRow	Returns the current row number
DeleteRow	Removes the specified row from the DataWindow (places it in the Delete buffer; does not delete the row from the database)
Filter	Displays rows in the DataWindow based on the current filter
InsertRow	Inserts a new row
Reset	Clears all rows in the DataWindow
Retrieve	Retrieves rows from the database
RowsCopy and RowsMove	Copies or moves rows from one DataWindow control to another
ScrollToRow	Scrolls to the specified row
SelectRow	Highlights a specified row
ShareData	Shares data among different DataWindow controls
Update	Sends to the database all inserts, changes, and deletions that have been made since the last Update

For a complete list of the DataWindow functions, see *Objects and Controls*.

For complete information about each function, see the *Function Reference*.

About dynamic DataWindow objects

The functions listed above manipulate data in the DataWindow control but don't change the definition of the underlying DataWindow object. PowerScript also provides two functions that allow you to access and manipulate *the attributes of a DataWindow object*. Using these functions, you can provide an interface for your user to actually change the DataWindow object during execution—create a dynamic DataWindow object. For example, you can change the appearance of a DataWindow or allow your user to create ad hoc reports.

For more information, see the next chapter.

Using DataWindow attributes and events

This chapter mentions only a few of the DataWindow attributes and events that you can use to manipulate DataWindow controls.

For more information about DataWindow attributes and events (including action codes you can set for an event to change the action that occurs after the event is triggered), see *Objects and Controls*.

Creating reports

You can use DataWindow objects to create standard business reports such as financial statements, sales order reports, employee lists, or inventory reports.

To create production reports, you:

- Determine the type of report you want to produce (for example, a report containing sales figures or employee addresses)
- Build a DataWindow object to display data for the report
- Place the DataWindow object in a DataWindow control in a window
- Write scripts to perform the processing required to populate the DataWindow control and print the contents as a report

Calling InfoMaker from within an application
If your users have installed InfoMaker, the Powersoft end-user reporting product, you can invoke InfoMaker from within a PowerBuilder application by using the Run PowerScript function. By doing this, you let your users create and save their own reports.

For more information about Run, see the *Function Reference.*

For information about invoking InfoMaker from the command line, see the InfoMaker *Getting Started* manual.

Planning the DataWindow object

To design the report, you create a DataWindow object. You select the data source and presentation style, then you can:

- Sort the data
- Create groups in the DataWindow object to organize the data in the report and force page breaks when the group values change
- Enhance the DataWindow to look like a report

  For example, you might want to add a title, column headers, and a computed field to number the pages.

Using fonts
Printer fonts are usually shorter and fatter than screen fonts, so text may not print in the report exactly as it displays in the DataWindow painter. You can pad the text fields to compensate for this discrepancy.

You should test the report with a small amount of data before you print a large report.

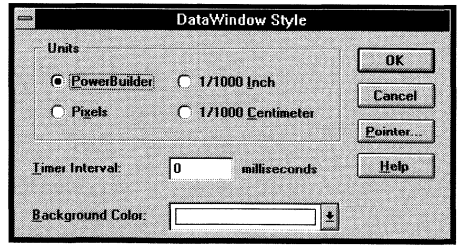
Defining print specifications

When you are satisfied with the look of the DataWindow object, you are ready to create the print specifications for the report.

❖ To create print specifications:

1 In the DataWindow painter, select Design➤DataWindow Style from the menu bar.

The DataWindow Style dialog box displays.



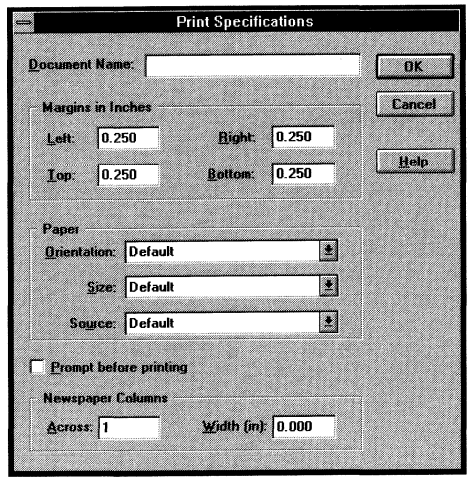
2 Select a unit of measure. The default unit of measure is PowerBuilder units, but it is easier to specify the margins when the unit of measure is inches or centimeters.

3 Click OK.

The DataWindow Style window closes.

4 Select Design➤Print Specifications from the menu bar.

The Print Specifications dialog box displays. It uses the units of measure you specified in the DataWindow Style window.



5 Specify a name in the Document Name box. This is the name that will be used in the print queue to identify the report.

6 Specify the margins for the report.

7 Select the paper's orientation, size, and source from the dropdown listboxes. For orientation, choose from the following:

Setting	Result
Default	Uses the default printer setup
Portrait	Prints the contents of the DataWindow across the width of the paper
Landscape	Prints the contents of the DataWindow across the length of the paper

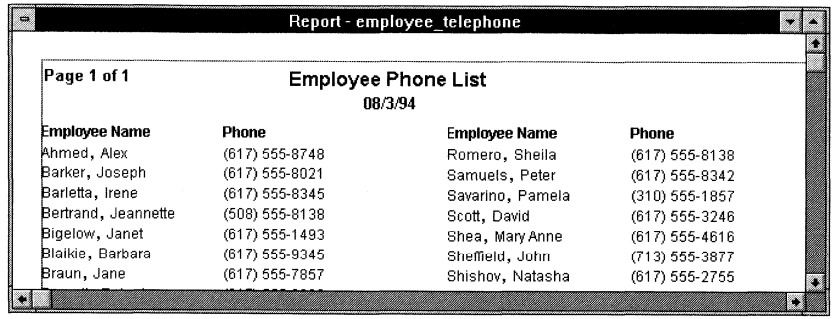
8 If you want the user prompted to specify the print setup before printing during execution, select the Prompt before printing checkbox. PowerBuilder will display the standard Print Setup dialog each time the user goes to print.

9 If you want a multiple-column report where the data fills one column on a page, then the second, and so on, as in a newspaper, select the number and width of the columns in Newspaper Columns box (next).

10 Click OK.

Printing with newspaper-style columns

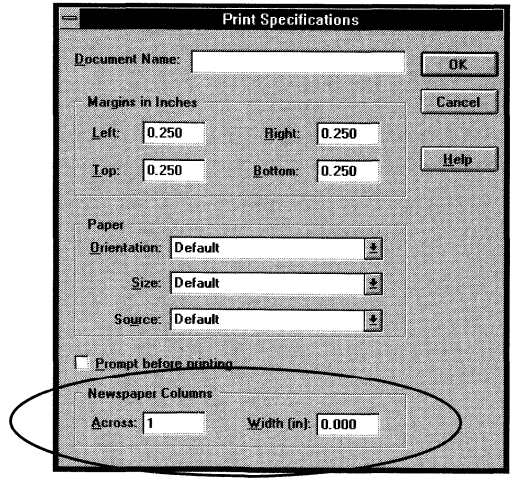
When you define a DataWindow, you can specify that it is printed in multiple columns across the page, like a newspaper. A typical use of newspaper-style columns is a phone list, where you want to have more than one column of names on a printed page. The following DataWindow (shown in print preview) has two newpaper-style columns.



❖ To define newspaper-style columns for a DataWindow:

1 Build a tabular DataWindow with the data you want.

2 Select Design➤Print Specifications from the menu bar to display the Print Specifications dialog box.



3 Specify the number of columns across the page and the width of columns.

4 Click OK to close the dialog box.

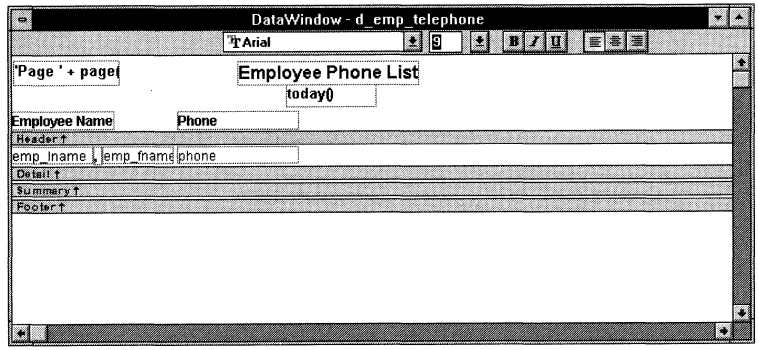
5 For those objects in the DataWindow that you do *not* want to appear multiple times on the page (such as headers), select Layer➤Suppress After First from the popup menu.

This process is illustrated with the following example.

Example

This section describes how you would create the preceding newspaper-style DataWindow.

First create a tabular DataWindow with the last name, first name, and phone number columns. Then add a title, page number, and date. The DataWindow looks like this in the workspace.

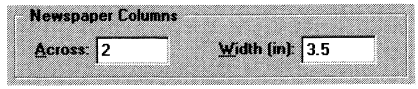


Tip

The Emp_Fname column is defined as Slide Left so it displays just to the right of the last name.

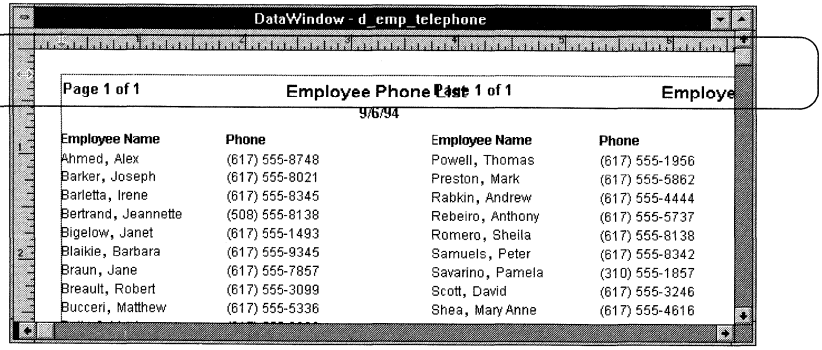
For more information on sliding columns, see the *User's Guide*.

Next you specify 2 columns across and a column width of 3.5 inches in the Newspaper Columns box in the Print Specifications dialog box.



Now the DataWindow displays in two columns in Print Preview.

This information should show once per page

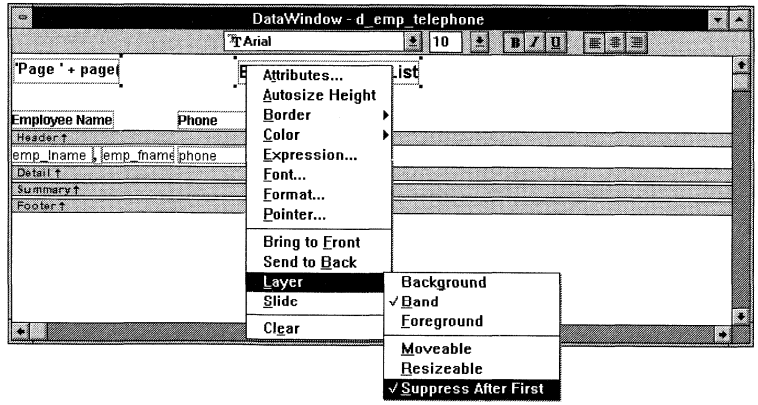


Use Print Preview to see the printed output

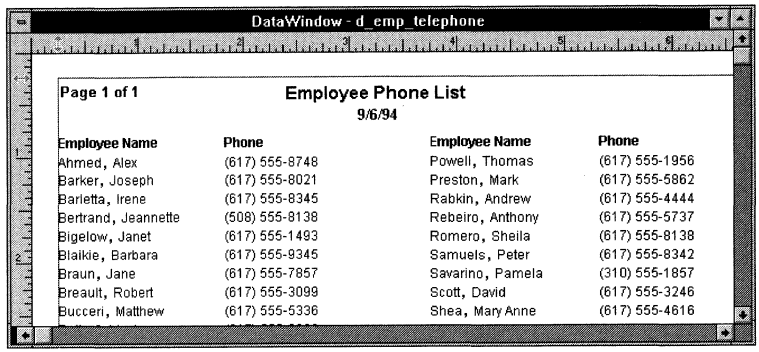
Newspaper-style columns are used only when the DataWindow is printed. They are not used when a DataWindow executes (or in Preview). Therefore, to see them in PowerBuilder, use Print Preview in the DataWindow painter.

Notice that everything above the column headers, which includes page number, title, and date, should show only once per page, but it shows twice because of the two column specification.

To specify that page number, title, and date should appear only once on the page, select them in the workspace and choose Layer➤Suppress After First from the popup menu.



The finished DataWindow looks like this, with one set of page heading information and two columns of column header and detail information.



Printing the report

After you build the DataWindow object and create print specifications, place the DataWindow object in a DataWindow control in a window as described in "Associating a DataWindow object with its control" on page 307.

To allow users to print the report, you need a script for an event in the window or a control. For example, you can place a CommandButton or PictureButton in the window, then write a script for the Clicked event in the button to print the report.

If the window has a single DataWindow control, use Format 1 of the Print function to print the contents of the DataWindow. If the window has multiple DataWindow controls, you can use multiple PrintDataWindow function calls in a script to print the contents of all the DataWindow controls in one print job.

For information about Print and PrintDataWindow, see the *Function Reference*.

Examples

This statement prints the contents of the DataWindow control dw_Sales:

```
dw_Sales.Print( )
```

These statements print the contents of three DataWindow controls in a single print job:

```
int   job
job = PrintOpen("Employee Reports")
// Each DataWindow starts printing on a new page.
PrintDataWindow(job, dw_EmpHeader)
PrintDataWindow(job, dw_EmpDetail)
PrintDataWindow(job, dw_EmpDptSum)
PrintClose(job)
```

Enhancing the reporting options

There are many ways to create an end-user reporter using the basic techniques described above.

Examples

Allow users to specify a date range in two SingleLineEdits, then use the Filter function to limit the report to data in the range.

Create a number of DataWindow controls and then allow the users to select the report they want from a list.

Place PictureButtons across the top of a window to provide reporting options for users.

CHAPTER 11

Using Dynamic DataWindow Objects

About this chapter

This chapter describes how to modify and create DataWindow objects during execution.

Contents

Overview

DataWindow objects and all entities in DataWindow objects (such as columns, text, graphs, and pictures) each have a set of attributes. You can look at and change the values of these attributes during execution using PowerScript functions. You can also create DataWindow objects during execution.

A DataWindow object that is modified or created during execution is called a **dynamic DataWindow object**. Using dynamic DataWindow objects, you can allow users to change the appearance of the DataWindow object (for example, change the color and font of the text) or create ad hoc queries by redefining the data source.

After you create a dynamic DataWindow object and the user is satisfied with the way it looks and the data that is displayed, the user can print the contents as a report and save the DataWindow object in a library for use in the future.

Modifying a DataWindow object

During execution, you can modify the appearance of a DataWindow object by changing the value of its attributes or adding or deleting objects from the DataWindow object.

Using Modify

You use the Modify function to modify an existing DataWindow dynamically during execution. Essentially, Modify does at execution time what you can do during development in the DataWindow painter.

Modify has the following syntax:

DataWindowControl.**Modify** (*ModString*)

Parameter	Description
DataWindowControl	The name of the DataWindow control in which PowerBuilder will modify the DataWindow object
ModString	A string that defines the modifications to the DataWindow object

Using Describe to get current attribute values

Before calling Modify to change an attribute, you might want to get the current value and save it in a variable, so you can restore the original value later. To obtain information about the current attributes of a DataWindow object or an element in a DataWindow object, use the Describe function.

For more information about Describe, see the *Function Reference.*

Easier coding of DataWindow syntax

Included with PowerBuilder is DWSYN40.EXE, an application that makes it very easy for you to specify Describe, Modify, and SyntaxFromSQL statements. You can simply click buttons to specify which attributes of a DataWindow you want to use, and the application automatically builds the appropriate Describe, Modify, or SyntaxFromSQL syntax.

The application is installed in the PowerBuilder directory.

Types of modifications

You can specify three types of requests in the *ModString* argument of Modify:

- Attributc assignments
- CREATE element
- DESTROY element

Making attribute assignments

You use an assignment statement to set an attribute of the DataWindow object or one of its elements dynamically. The syntax is:

ElementName.Attribute = Value

Parameter	Description
ElementName	The name of the element whose attribute you want to change the value of. To change the attribute for the DataWindow object itself (as opposed to changing an attribute for an element in the DataWindow), use **DataWindow** as the object name. You can refer to a column by its name or by its number preceded by a pound sign (#2).
Attribute	The name of the attribute you want to set to a new value.
Value	The value to which you want to set the attribute. *Value* can be a specific value or a valid DataWindow expression that returns a value that is the appropriate data type for the specified attribute.

You can have multiple assignment statements in the Modify *ModString* parameter; separate the assignments with a space.

For information about DataWindow attributes, see the *Function Reference.*

Displaying the attributes online
You can use the Object browser within the PowerScript painter to get a list of DataWindow attributes: select a DataWindow object as the Object Type and Attributes as the Paste Category. To see the attributes for an element in a DataWindow object, double-click the DataWindow object name, then click the element.

Using expressions

With some DataWindow attributes, you can assign a value through an expression that PowerBuilder evaluates during execution, instead of having to directly assign a value.

When using an expression, use this syntax:

' *DefaultValue* ~t *DataWindowExpression* '

Parameter	Description
DefaultValue	The value to assign the attribute if *DataWindowExpression* doesn't return a valid value
~t	The tab separator
DataWindowExpression	A DataWindow expression that results in a value appropriate for the attribute you are assigning a value to

For information about which attributes can be assigned expressions, see Appendix A, "DataWindow Object Attributes," in the *Function Reference*.

Examples

The following statement changes the DataWindow background color to red:

```
dw_1.Modify("DataWindow.Color = 255")
```

The following statement changes the text of the static text named title to *Department ID*. Note the use of single quotes to embed a string within another string:

```
dw_1.Modify("title.Text = 'Department ID'")
```

The following statement displays the names in the product column in italics:

```
dw_1.Modify("product.Font.Italic = 1")
```

The following statement changes the display format of the salary column:

```
dw_1.Modify("salary.Format='[red]$#,###,##0.00'")
```

The following statement displays the salary for an employee in red if it is less than $12,000 and in black if it is greater than or equal to $12,000. The statement uses the tab-separated expression syntax described above:

```
dw_1.Modify( &
"salary.Color = '0 ~t if(salary <12000,255,0)'")
```

The following statements change the display of units that are greater than 20 to be red, in bold font, and left-aligned:

```
dw_1.Modify( &
"units.Color       = '0   ~t if(units>20,255,0)' &
 units.Font.Weight = '400 ~t if(units>20,700,400)' &
 units.Alignment   = '1   ~t if(units>20,0,1)' ")
```

Using CREATE to add elements

Use CREATE to add DataWindow elements (such as text, bitmaps, and graphic objects) dynamically to the DataWindow object.

Example

The following statement adds an ellipse (oval) to the detail band of the DataWindow object:

```
dw_1.Modify( &
"CREATE ellipse(band=detail &
x='1229' y='0' height='112' &
width='739' brush.hatch='6' &
name=oval2 &
brush.color='65535' pen.style='0' &
pen.width='10' pen.color='0' &
background.mode='1' ")
```

For more information

For complete descriptions of the syntax to use when creating DataWindow objects or elements, see the *Function Reference.*

Tip
To get a good idea of the CREATE syntax, go the Library painter, export a DataWindow object, then look at the text file.

Using DESTROY to remove elements

Use DESTROY to remove elements dynamically from the DataWindow object.

Example

The following statement deletes an ellipse from the DataWindow object:

```
dw_1.Modify("DESTROY oval2")
```

Creating a DataWindow object

There are two ways to create a DataWindow object: in the DataWindow painter or using a script during execution. Creating DataWindow objects in the DataWindow painter is described in Part Four, "Working with Databases," in the *User's Guide*.

This section describes how to create a DataWindow object during execution.

Using Create

You use the Create function to create a DataWindow object dynamically during execution. Create has the following syntax:

DataWindowControl.**Create** (*Syntax* {, *ErrorBuffer* })

Parameter	Description
DataWindowControl	The name of the DataWindow control in which PowerBuilder will create the new DataWindow object.
Syntax	A string containing the DataWindow source code that will be used to create the DataWindow object. For more information, see "Specifying the DataWindow object syntax" on page 350.
ErrorBuffer	(Optional) The name of a string that you want to fill with any error messages that occur. If you do not specify an error buffer, a message box will display the error messages.

When you call **Create**, PowerBuilder creates a DataWindow object using the source code in *Syntax* and replaces the DataWindow object currently associated with the specified *DataWindowControl* with the new DataWindow object.

Reassociating the DataWindow control with a transaction object
Create destroys the association between the DataWindow control and the transaction object. Therefore, always call the SetTransObject or SetTrans function after you call Create.

Specifying the DataWindow object syntax

There are three ways to specify the syntax required for Create. You can:

- Use the SyntaxFromSQL function
- Use the LibraryExport function
- Create the syntax yourself

You will use SyntaxFromSQL to create the syntax for most dynamic DataWindow objects. If you use SyntaxFromSQL, all you have to do is provide the SELECT statement and the presentation style and PowerBuilder will do the rest.

However, you will need to create the syntax yourself to use some of the more advanced features of a dynamic DataWindow object. For example, to create group breaks dynamically, you will have to create the syntax yourself.

Using SyntaxFromSQL

The SyntaxFromSQL function has four required arguments:

- The name of a connected transaction object
- A string containing the SELECT statement that PowerBuilder will use to create the DataWindow object
- A string identifying the presentation style you want for the DataWindow object
- The name of a string that you want to fill with any error messages that may result

SyntaxFromSQL returns the complete syntax for a DataWindow object that is built using the specified SELECT statement.

If using SQL Server or ORACLE
If your DBMS is SQL Server or ORACLE, set the AutoCommit attribute of the transaction object to TRUE before you call SyntaxFromSQL. This ensures that the syntax PowerBuilder generates includes the tab order and update capability.

For more information about the SyntaxFromSQL function, see the *Function Reference*.

Examples

The following statements store in the variable Syntax the DataWindow source for a tabular DataWindow object generated by the SyntaxFromSQL function. If an error occurs, PowerBuilder stores the error messages in the string Errmsg:

```
string   Syntax, Sqlselect, Errmsg

Sqlselect = "SELECT emp_data.emp_id, "&
+"emp_data.emp_name FROM emp_data "&
+"where emp_data.emp_salary >45000"

CONNECT ;
Syntax = &
SyntaxFromSQL(SQLCA,Sqlselect, &
"style(type=tabular)",Errmsg))
```

The following statements create a grid DataWindow object from the DataWindow source generated in the SyntaxFromSQL function. If errors occur, the string Errors will contain any error messages that are generated. The new DataWindow object is associated with the existing DataWindow control dw_emp:

```
string   Sqlselect, Errors

Sqlselect = "SELECT emp_data.emp_id,"&
   +"emp_data.emp_name FROM emp_data "&
   +"where emp_data.emp_salary >45000"

dw_emp.Create(SyntaxFromSQL(SQLCA, &
Sqlselect,"style(type=grid)", Errors))

if Len(Errors) > 0 then MessageBox("Caution", &
   "These errors occurred: " + Errors)
```

Using LibraryExport

You can use the LibraryExport PowerScript function to export the syntax for a DataWindow object and store the syntax in a string.

You can then use the exported syntax (or a modification of the syntax) in Create to create a DataWindow object.

For more information about LibraryExport, see the *Function Reference.*

Example

These statements export the DataWindow object d_emp from the library named dwTemp to a string named DWsyn and then use the syntax to create a DataWindow object and associate it with the DataWindow control dw_emp:

```
string   DWsyn, Errors

DWsyn = LibraryExport("d:\test\dwTemp.pbl", &
   "d_emp", ExportDataWindow!)
dw_emp.Create(DWsyn,Errors)
```

Creating the syntax yourself

You will use SyntaxFromSQL to create the syntax for most dynamic DataWindow objects. However, to use some of the advanced dynamic DataWindow features—such as creating a group break—you will need to create the syntax yourself.

To create your own DataWindow object syntax, use the format described in the section on DataWindow syntax in the *Function Reference.*

> **Tip**
> To see examples of DataWindow object syntax, go to the Library painter and export a DataWindow object to a text file, then view the file in a text editor.

Providing query ability to users

When you call the Retrieve function for a DataWindow, the rows specified in the SELECT statement defined for the DataWindow are retrieved. You can give your users the ability to further specify which rows are retrieved during execution by putting the DataWindow into **query mode** using Modify.

Limitations
You cannot use query mode in a DataWindow whose data source you have modified *syntactially* (as opposed to graphically) and that contains the UNION keyword or nested SELECT statements.

How query mode works

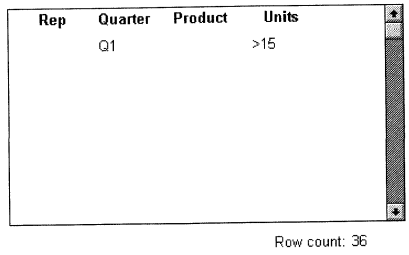
Once the DataWindow is in query mode, users can specify selection criteria using query by example—just as you do when you use Quick Select to define a data source. When criteria have been defined, they are added to the WHERE clause of the SELECT statement the next time data is retrieved.

The following windows show what happens when query mode is used. First, data is retrieved into the DataWindow. There are 36 rows.

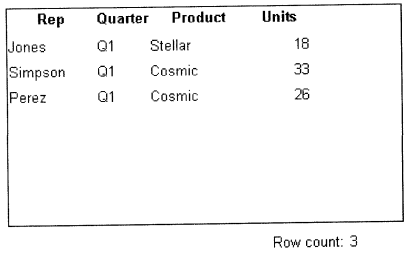
Rep	Quarter	Product	Units
Simpson	Q1	Stellar	12
Jones	Q1	Stellar	18
Perez	Q1	Stellar	15
Simpson	Q1	Cosmic	33
Jones	Q1	Cosmic	5
Perez	Q1	Cosmic	26
Simpson	Q1	Galactic	6

Row count: 36

Next, query mode is turned on. The retrieved data disappears and users are presented with empty rows where they can specify selection criteria. Here the user wants to retrieve rows where Quarter = Q1 and Units > 15.



Next, Retrieve is called and query mode is turned off. PowerBuilder adds the criteria to the SELECT statement, retrieves the three rows that meet the criteria, and displays them to the user.



You can turn query mode back on and allow the user to revise the selection criteria and retrieve again.

Using query mode

❖ To use query mode:

1 When you want to provide query mode to your user during execution, turn on query mode by coding the following in a script:

DataWindowControl.**Modify**("datawindow.querymode=yes")

All data displayed in the DataWindow is blanked out (though it is still in the DataWindow control's Primary buffer) and the user can enter selection criteria where the data had been.

2 The user specifies selection criteria in the DataWindow, using the same techniques you use to define criteria when using Quick Select to define a DataWindow object's data source. Criteria entered in one row are ANDed together; criteria in different rows are ORed. Valid operators are =, <>, <, >, <=, >=, LIKE, IN, AND, and OR.

For more information about Quick Select, see the *User's Guide.*

3 Call Retrieve in a script, then turn off query mode to display the newly retrieved rows:

```
DataWindowControl.Retrieve( )
DataWindowControl.Modify("datawindow.querymode=no")
```

PowerBuilder adds the newly defined selection criteria to the WHERE clause of the SELECT statement, then retrieves and displays the specified rows.

Tip

You can look at the revised SELECT statement that is sent to the DBMS when data is retrieved with criteria by calling the DataWindow function GetSQLPreview in the SQLPreview event of the DataWindow control.

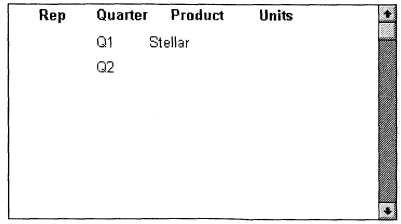
How the criteria affect the SELECT statement

Criteria specified by the user are added to the SELECT statement that originally defined the DataWindow.

For example, if the original SELECT was:

```
SELECT printer.rep, printer.quarter,
   printer.product, printer.units
FROM printer
WHERE printer.units < 70
```

and the following criteria are specified:



the new SELECT statement is:

```
SELECT printer.rep, printer.quarter,
   printer.product, printer.units
FROM printer
WHERE printer.units < 70
AND (printer.quarter = 'Q1'
AND printer.product = 'Stellar'
OR printer.quarter = 'Q2')
```

Clearing selection criteria

To clear the selection criteria, code the following:

DataWindowControl.DataObject = *DataWindowControl*.DataObject
DataWindowControl.**SetTransObject**(*TransactionObject*)

Sorting in query mode

You can allow users to sort rows in a DataWindow while specifying criteria in query mode. The following statement:

DataWindowControl.**Modify**("datawindow.querysort=yes")

makes the first row in the DataWindow dedicated to sort criteria (just as in Quick Select in the DataWindow painter).

Overriding column properties during query mode

By default, query mode uses edit styles and other definitions of the column (such as the number of allowable characters). If you want to override these properties during query mode and provide a standard edit control for the column, code the following:

DataWindowControl.**Modify**("*Column*.criteria.override_edit=yes")

(You can also specify this in the DataWindow painter by choosing Query Criteria➤Override Edit from the column's popup menu.)

With properties overridden for criteria, users can specify any number of characters in a cell (they are not constrained by the number of characters allowed in the column in the database).

Forcing users to specify criteria for a column

You can force users to specify criteria for a column during query mode by coding the following:

DataWindowControl.**Modify**("*Column*.criteria.required=yes")

(You can also specify this in the DataWindow painter by choosing Query Criteria➤Equality Required from the column's popup menu.)

Doing this ensures that the user specifies criteria for the column and that the criteria for the column uses = rather than other operators, such as < or >=.

Providing Help buttons

A DataWindow object has attributes related to online Help. By initializing the DataWindow.Help.File attribute to the name of a Help file, you can display Help command buttons on dialog boxes that display for a DataWindow during execution.

For complete information on the help-related DataWindow attributes, see Appendix A, "DataWindow Object Attributes," in the *Function Reference*.

Reusing a DataWindow object

You can reuse a DataWindow object by retrieving its syntax from the library it is stored in, then using the syntax to create a DataWindow object dynamically in a DataWindow control.

Here is a typical way to accomplish this in an application. Use:

- The LibraryDirectory function to obtain a list of DataWindow objects and other library entries in the current library.
- A DropDownListBox to list the DataWindow objects in the library and then allow the user to select a DataWindow from the list.
- The LibraryExport function to export the selected DataWindow object syntax into a string variable.
- Create to use the DataWindow syntax to create the DataWindow object in the specified DataWindow control.
- The Describe function to get the current DataWindow object syntax. For example:

```
string   dwSyntax
dwSyntax = dw_1.Describe("datawindow.syntax")
```

- The Modify function to allow the user to modify the DataWindow object.
- The LibraryImport function to save the user-modified DataWindow object in a library.

For information about these functions, see the *Function Reference*.

CHAPTER 12

Piping Data Between Data Sources

About this chapter

This chapter tells how you can use a pipeline object in your application to pump data from one or more source tables to a new or existing destination table.

Contents

Overview

Earlier in this manual (in Chapter 2, "Designing an Application"), you learned about the data pipeline feature that PowerBuilder provides 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 may 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 do that migration right then and there.

  For more information on using the Data Pipeline painter this way, see the *User's Guide*.

- **To implement data migration capabilities in an application**

  If you're 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, the next thing to do is figure out the various steps that this involves. At the most general level, there are five basic steps that you'll 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.

Sample application

To see a working example of using data pipelines, you can examine the sample Order Entry application of the Anchor Bay Nut Company. This technique is implemented in the following window:


Library: abnc_gde.pbl


Window object: w_sales_extract

You'll learn more about that example on the coming pages.

Building the objects you need

To implement data piping in an application, you'll 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:

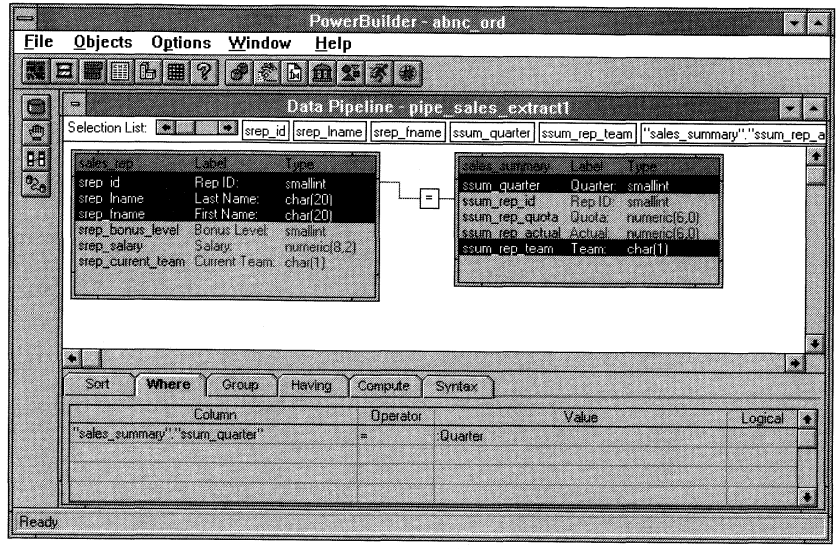
- **The source tables** to access and the data to retrieve from them
- **The destination table** to which you want that data piped
- **The piping operation** to perform (create, replace, refresh, append, or update)
- **The frequency of commits** during the piping operation (either after every *n* rows are piped or only after all rows are piped)
- **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 all of the details on using the Data Pipeline painter to build your pipeline object, see the *User's Guide*.

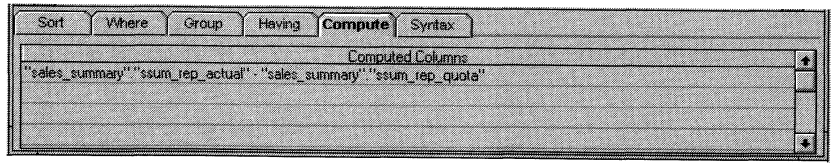
Example

Here's an example of how the Data Pipeline painter was used to define the pipeline object named pipe_sales_extract1 (one of two pipeline objects employed by the w_sales_extract window in the sample Order Entry application).

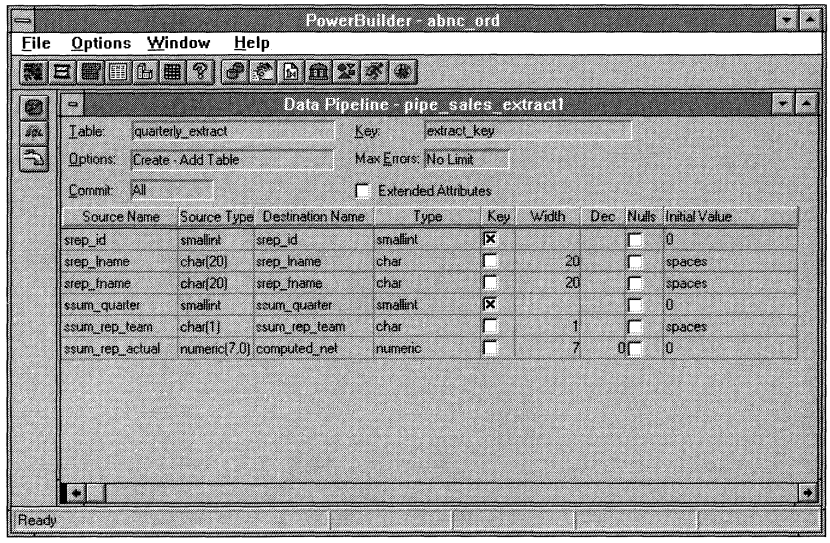
The source data to pipe This pipeline object joins two tables (Sales_rep and Sales_summary) from the company's sales database to provide the source data to be piped. It retrieves just the rows from a particular quarter of the year (which the application must specify by supplying a value for the retrieval argument named *quarter*).



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:



How to pipe the data The details of how pipe_sales_extract1 is to pipe its source data are specified here:



Notice that this pipeline object is defined to create a new destination table named Quarterly_extract. A little later you'll 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've seen how your pipeline object defines the details of the data and access for a pipeline. But a pipeline object doesn't include the logistical supports—attributes, events, and functions—that an application requires to handle pipeline execution and control.

Introducing the pipeline system object

To provide these logistical supports, you must build an appropriate user object inherited from the PowerBuilder **pipeline system object**. This system object contains various attributes, events, and functions that enable your application to manage a pipeline object at execution time. These include:

Attributes	Events	Functions
DataObject	PipeStart	Start
RowsRead	PipeMeter	Repair
RowsWritten	PipeEnd	Cancel
RowsInError		
Syntax		

A little later in this chapter you'll learn how to use most of these attributes, events, and functions in your application.

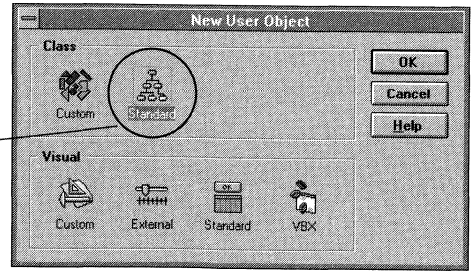
❖ To build the supporting user object for a pipeline:

1 Open the User Object painter.

The Select User Object dialog box displays.

2 Click the New button.

The New User Object dialog box displays, prompting you to specify which kind of user object you want to create.

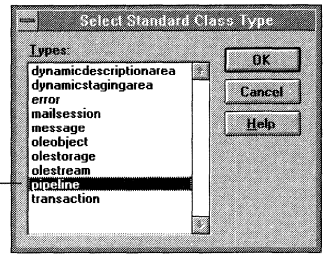


Select this

3 Select the Standard Class kind of user object and click OK.

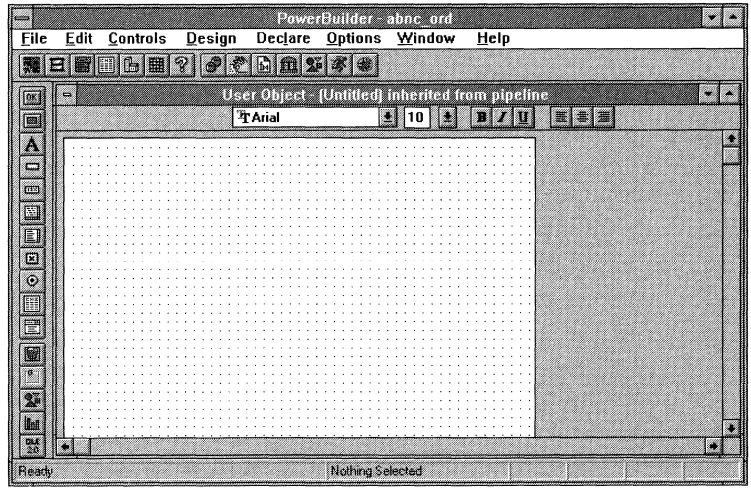
The Select Standard Class Type dialog box displays, prompting you to specify the name of the PowerBuilder system object (class) from which you want to inherit your new user object.

Select this



4 Select pipeline and click OK.

The User Object painter workspace displays, enabling you to work with your new user object.



5 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 376.

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 isn't automatically tied in any way to a particular pipeline object you've built in the Data Pipeline painter.

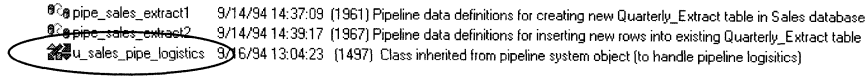
To take advantage of this flexibility, make sure that the events, functions, and variables you code in the user object are generic enough to accommodate any pipeline object.

6 Save the user object.

For more information on working with the User Object painter, see the *User's Guide*.

Example

At the Anchor Bay Nut Company, they followed the preceding steps to create their standard-class user object inherited from the PowerBuilder pipeline system object. Then they saved this user object under the name u_sales_pipe_logistics in the library ABNC_GDE.PBL:



Building a window

One other object you need when piping data in your application is a window. You'll 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

As you proceed through this chapter, you'll learn the details of how to implement these capabilities in your window.

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 can't pipe to the destination table for some reason). You don't have to associate a DataWindow object with this DataWindow control—the pipeline will provide one of its own at execution time.

To learn about how you'll work with this DataWindow control in your application, see "Starting the pipeline" on page 375 and "Handling row errors" on page 383.

Optional features for your window

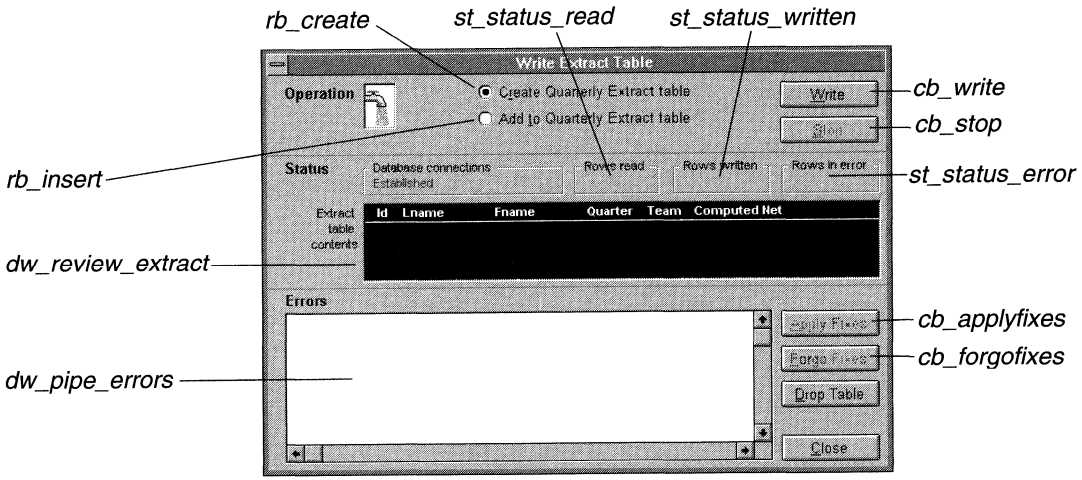
Other than that, you can design the window as you like. You'll typically want to include various other controls, such as:

- **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 *User's Guide*.

Example

At the Anchor Bay Nut Company, they built the following window to handle the user-interface aspect of the data piping in their Order Entry application. They named this window w_sales_extract.



Several of the controls in this window are used to implement particular pipeline-related capabilities. Here's more information about them:

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 can't 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're 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 9, "Using Transaction Objects."

2 Create an instance of your supporting user object (so that the application can use its attributes, 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 attribute of your user-object instance.

For more information on coding the CREATE and Assignment statements, see *PowerScript Language*.

Example

Here's what they did at the Anchor Bay Nut Company to take care of these pipeline setup chores in their 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, they wrote code in a user event named uevent_pipe_setup (which is posted from the Open event of the w_sales_extract window).

To establish their *source database connection*, they coded:

```
// 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 attributes of the
// itrans_source transaction object.
                  .
                  .
                  .
// Now connect to the source database.

CONNECT USING itrans_source;
```

To establish their *destination database connection*, they coded:

```
// 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 attributes of the
// itrans_destination transaction object.
                  .
                  .
                  .
// Now connect to the destination database.

CONNECT USING itrans_destination;
```

Creating an instance of the user object Earlier you learned how they developed a supporting user object named u_sales_pipe_logistics. To use u_sales_pipe_logistics in the application, they first declared a variable of its type:

```
// This is an instance variable for the
// w_sales_extract window.

u_sales_pipe_logistics iuo_pipe_logistics
```

Then, they wrote 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 exists).

To choose a pipeline object and prepare to use it, they wrote 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 execution time (through string variables), you must package these objects in one or more PBD files when deploying the application. You cannot include pipeline objects in an executable (EXE) file.

For more information, see Chapter 5, "Deploying an Application."

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'll 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. It will accomplish this by automatically associating the PowerBuilder pipeline-error DataWindow object with your DataWindow control when needed.
- Values for retrieval arguments you've defined in the pipeline object. If you omit these values, the Start function will prompt the user for them automatically at execution time.

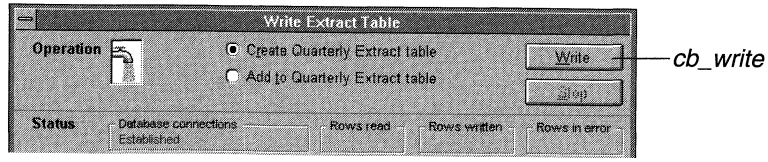
2 Test the result of the Start function.

For more information on coding the Start function, see the *Function Reference*.

Example

Here's what they did at the Anchor Bay Nut Company to start pipeline execution in their Order Entry application.

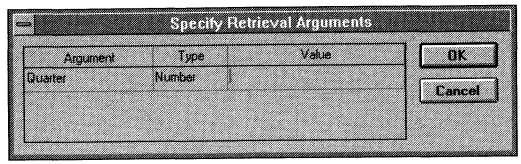
Calling the Start function When users want to start their selected pipeline, they click the cb_write CommandButton in the w_sales_extract window:



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 they didn't supply a value for the pipeline's retrieval argument (*quarter*). As a consequence, the Start function prompts the user for it:



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 isn't 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:

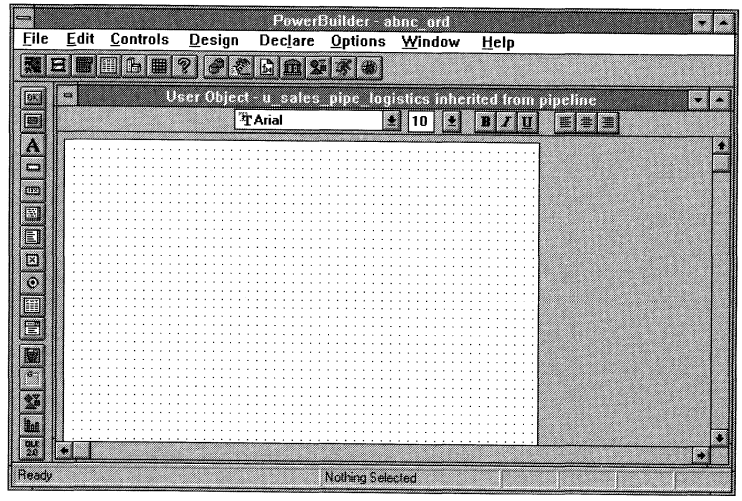
- **The rows read** by the pipeline from the source tables
- **The rows written** by the pipeline to the destination table or to the error DataWindow control
- **The rows in error** that the pipeline has written to the error DataWindow control (but not to the destination table)

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.

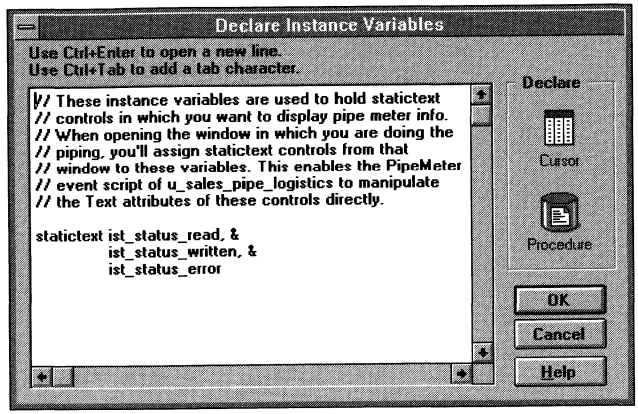
The User Object painter workspace displays, enabling you to edit your user object.



2 Declare three instance variables of type StaticText.

You'll 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.

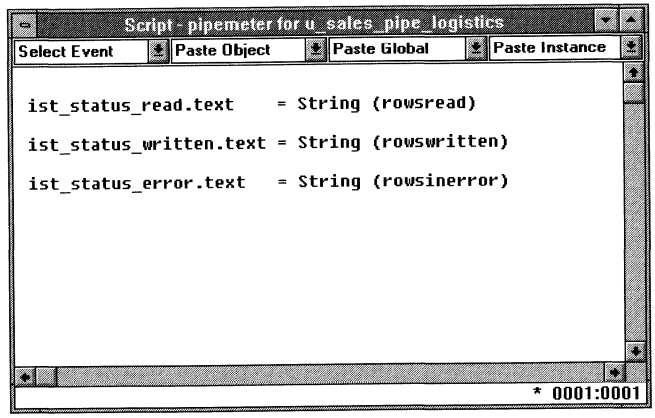
Here's how they did this at the Anchor Bay Nut Company for the user object u_sales_pipe_logistics:



3 Edit the user object's PipeMeter event script. In it, code statements to assign the values of these attributes (inherited from the pipeline system object):

- RowsRead
- RowsWritten
- RowsInError

to the Text attribute of your three StaticText instance variables. Here's how they did this at the Anchor Bay Nut Company for the user object u_sales_pipe_logistics:



4 Save your changes to the user object. Then close the User Object painter.

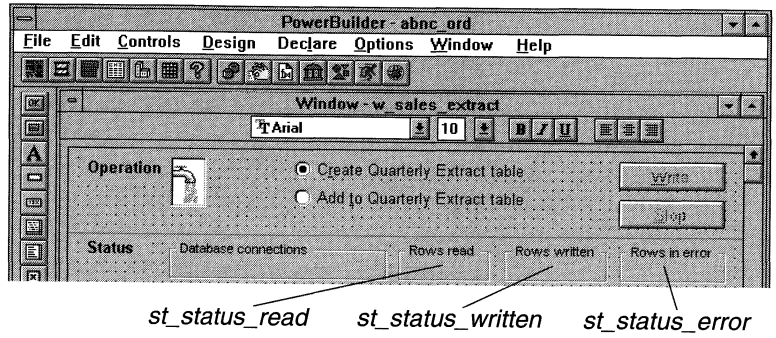
5 Open your window in the Window painter.

The Window painter workspace displays, enabling you to edit your window.

6 Paint three StaticText controls in the window:

- One to display the RowsRead value,
- One to display the RowsWritten value, and
- One to display the RowsInError value

Here's how they did this at the Anchor Bay Nut Company for the window w_sales_extract:



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 your three StaticText controls (which you just painted 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.

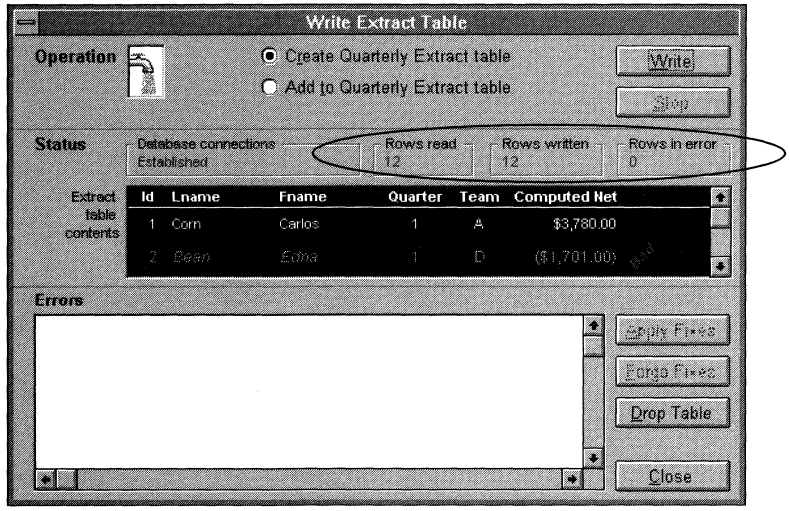
In the sample Order Entry application, they coded this logic in a user event named uevent_pipe_setup (which is posted from the Open event of the w_sales_extract window):

```
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.

Example

To see how this technique works, look at what happens once you start a pipeline in the w_sales_extract window of the sample Order Entry application. As the pipeline executes, the user object's PipeMeter event triggers and executes its code to display pipeline row statistics in the three StaticText controls:



Canceling pipeline execution

In many cases, you'll want to provide users (or your application itself) with the ability to stop execution of a pipeline while it is in progress. For instance, you might want to give users a way out if they start the pipeline by mistake or if execution is taking longer than desired (maybe because a large number of 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 attribute you specified when building your pipeline object in the Data Pipeline painter. You'll learn more about committing in the next section.

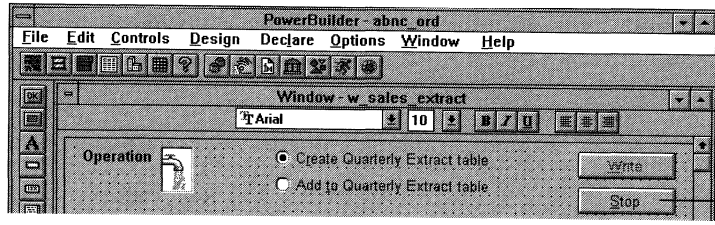
2 Test the result of the Cancel function.

For more information on coding the Cancel function, see the *Function Reference*.

Example

Here's what they did at the Anchor Bay Nut Company to let users cancel pipeline execution in their Order Entry application.

Providing a CommandButton When painting the w_sales_extract window, they included a CommandButton control named cb_stop:



They also wrote 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, they wrote 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

   MessageBox("Operation Status", &
              "Error when trying to stop piping.")

END IF
```

Together, these features let a user of the application click the cb_stop CommandButton to cancel a pipeline that's 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 don't need to write any COMMIT statements in your application's scripts to do this.

Example

For instance, both of the pipeline objects in the sample 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.

But you might want 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's currently executing. In this case too, the Commit attribute in the Data Pipeline painter determines what to do:

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 *User's Guide*.

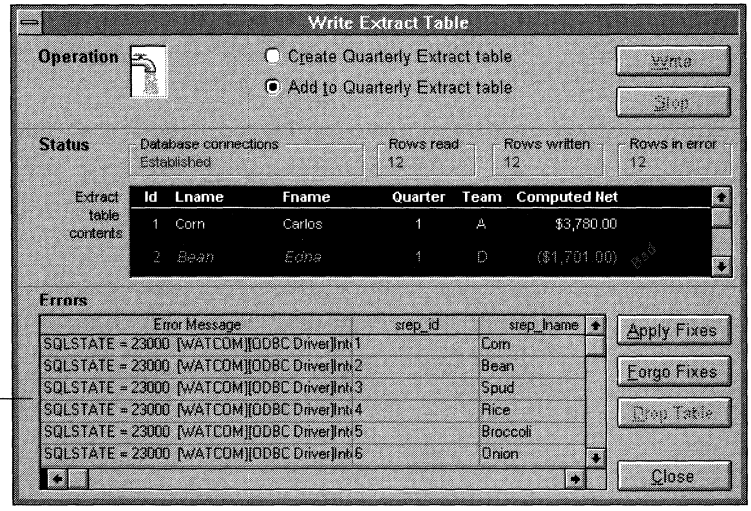
Handling row errors

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 sample 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:



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 won't appear.

For more information, see *Connecting to Your Database.*

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

You'll learn about these alternatives in the remainder of this section.

Repairing error rows

In many situations, it's 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 (this is the easy way for you, because it doesn't 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:

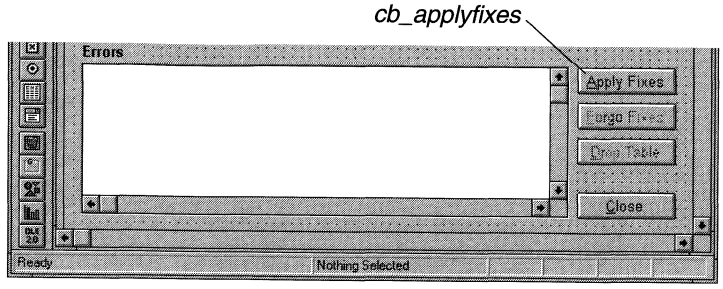
1. Code the Repair function in an appropriate script. In this function, you'll specify the transaction object for the destination database.
2. Test the result of the Repair function.

For more information on coding the Repair function, see the *Function Reference*.

Example

At the Anchor Bay Nut Company, they decided to let users edit the contents of the dw_pipe_errors DataWindow control to fix error rows that appear. That meant they needed to provide users with a way to apply those modified rows to the destination table. Here's what they did.

Providing a CommandButton When painting the w_sales_extract window, they included a CommandButton control named cb_applyfixes:



They also wrote 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, they wrote 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

   MessageBox("Operation Status", &
              "Error when trying to apply fixes.")

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 380.

Committing row repairs

The Repair function commits (or rolls back) database updates in the same way that the Start function does.

For details, see "Committing updates to the database" on page 382.

Handling rows that still aren't 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. But there's also the alternative of abandoning one or more of these rows—you'll 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 are not desirable to repair.

Techniques you can use for abandoning such error rows include these:

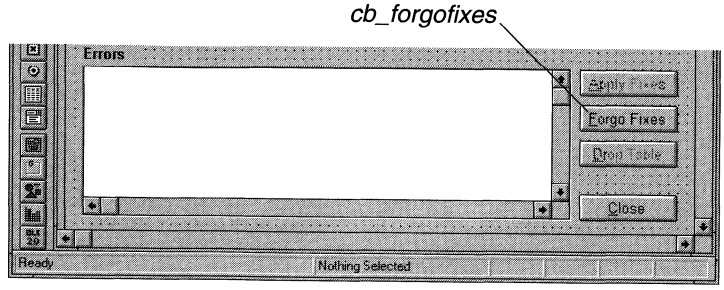
If you want to abandon	Use
All error rows in the error DataWindow control	The Reset function
One or more particular error rows in the error DataWindow control	The RowsDiscard function

For more information on coding these functions, see the *Function Reference*.

Example

At the Anchor Bay Nut Company, they decided to give users the ability to abandon all error rows in the dw_pipe_errors DataWindow control. Here's how they implemented this.

Providing a CommandButton When painting the w_sales_extract window, they included a CommandButton control named cb_forgofixes:



They also wrote 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, they wrote 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 is all done 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.

❖ To clean up when you're done 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 *PowerScript Language*.

Example

At the Anchor Bay Nut Company, they took care of these cleanup chores by writing code for the Close event of their w_sales_extract window.

Destroying the user-object instance At the beginning of this script, they coded the following statement to destroy their 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, they coded these statements to disconnect from the source database, test the result of the disconnection, and destroy their 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

   MessageBox("Database Connection Error", &
       "Problem when disconnecting from " +&
       "the source (Sales) database. " +&
       "Please call the Anchor Bay support team." +&
       "~n~r~n~rDetails follow: " +&
       String(itrans_source.SQLDBCode) + " " +&
       itrans_source.SQLErrText)

END IF

DESTROY itrans_source
```

Disconnecting from the destination database Finally, they coded 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

   MessageBox("Database Connection Error", &
       "Problem when disconnecting from " +&
       "the destination (Sales) database. " +&
       "Please call the Anchor Bay support team." +&
       "~n~r~n~rDetails follow: " +&
       String(itrans_destination.SQLDBCode) + " " +&
       itrans_destination.SQLErrText)

END IF

DESTROY itrans_destination
```

CHAPTER 13

Reading and Writing Text or Binary Files

About this chapter

This chapter describes how to manipulate files during execution.

Contents

Overview

You use PowerScript text file functions to read and write ASCII text in line mode or stream 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 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 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 function.

File functions

The following table lists the built-in PowerScript functions that manipulate files:

Function	Data type returned	Action
FileClose	Integer	Closes the specified file
FileDelete	Boolean	Deletes the specified file
FileExists	Boolean	Determines whether the specified file exists
FileLength	Long	Obtains the length of the specified file
FileOpen	Integer	Opens the specified file
FileRead	Integer	Reads from the specified file
FileSeek	Long	Seeks to a position in the specified file
FileWrite	Integer	Writes to the specified file

For complete information about these functions, see the *Function Reference.*

PART FOUR

Program Access Techniques

A collection of techniques you can use to implement program access features in the applications you develop with PowerBuilder. Includes: using DDE in an application, using OLE in an application, building a mail-enabled application, and adding other processing extensions.

CHAPTER 14

Using DDE in an Application

About this chapter

This chapter describes how PowerBuilder supports DDE.

Contents

Overview

One of the most useful and powerful features of Microsoft Windows is Dynamic Data Exchange (DDE). DDE makes it possible for two Windows applications to communicate with each other by sending and receiving commands and data. In this way, 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 *Function Reference*.

Return values
All DDE functions return an integer.

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.

Function	Action
StopHotLink	Ends a hot link with a DDE server application.

DDE client event

Event	Occurs when
HotLinkAlarm	A DDE server application has sent new (changed) data.

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.

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 15

Using OLE in an Application

About this chapter

This chapter describes several ways of implementing Object Linking and Embedding (OLE) in your PowerBuilder applications. You can display data from another application and allow users to edit that data in a blob (binary large-object) column in a DataWindow or in an OLE control in a window. You can use OLE automation to programmatically cause an OLE server application to modify its OLE object.

Contents

OLE support in PowerBuilder

Object Linking and Embedding (OLE) is a facility that allows Windows programs to share data and program functionality. Your PowerBuilder application is an OLE *container* application that can call upon OLE *server* applications to display and manipulate OLE objects.

DataWindows support OLE 1.0. They can include blob columns whose data is an OLE object. Because DataWindows support the OLE 1.0 standard, you need to be sure your server application supports OLE 1.0 objects. The user or a script can activate the OLE column and send simple commands to the server application.

The OLE 2.0 control in the Window painter supports OLE 2.0, as well as OLE 1.0. OLE 2.0 is an extension of OLE 1.0 and allows you to create compound documents with components from several applications. For most servers, you can also control the server application using functions and attributes available for that server.

In PowerBuilder, the OLE 2.0 control is a container for an OLE object. (The object can be an OLE 1.0 or OLE 2.0 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.

You can use PowerScript automation on an OLE 2.0 control that is visible in a window or invisibly on an object whose reference is stored in a OLEObject variable. The OLEObject data type lets you create an OLE object without having an OLE container visible in a window.

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 won't require these objects, but if you need to do something complicated, for example, combining several OLE objects into a single data structure, you can use these objects and their associated functions.

Using OLE columns in a DataWindow

You can create OLE columns in a DataWindow object. An OLE column allows you to:

- Store blob (binary large-object) data, such as Excel worksheets or Word for Windows documents, in the database
- Retrieve the blob data from the database into a DataWindow
- Use an OLE server application, such as Microsoft Excel or Word for Windows, to modify the data
- Store the modified data back in the database

This section illustrates how to do this with an example. In a database table is a blob column that contains Word for Windows 2.0 documents. The PowerBuilder user can retrieve the data into a DataWindow object, then open the stored documents in Word (the OLE server application) by double-clicking the column in the DataWindow (the OLE client application). Word for Windows opens the document and displays it in the Word workspace.

The user can modify the document in Word, then update the data in the DataWindow. When the database is updated, the OLE column, which contains the modified document, is stored in the database.

Database support for OLE columns
If your database supports a blob data type, then you can implement OLE columns in a DataWindow. The name of the data type that supports blob data varies. See your DBMS documentation for information on its data types.

Creating an OLE column

This section describes how to create an OLE column in a DataWindow object. The steps are illustrated using a table named Word that contains two columns, ID and File:

- The ID column is an integer and serves as the table's key.
- The File column is a blob data type and contains Word for Windows documents.

❖ To create a DataWindow with an OLE column:

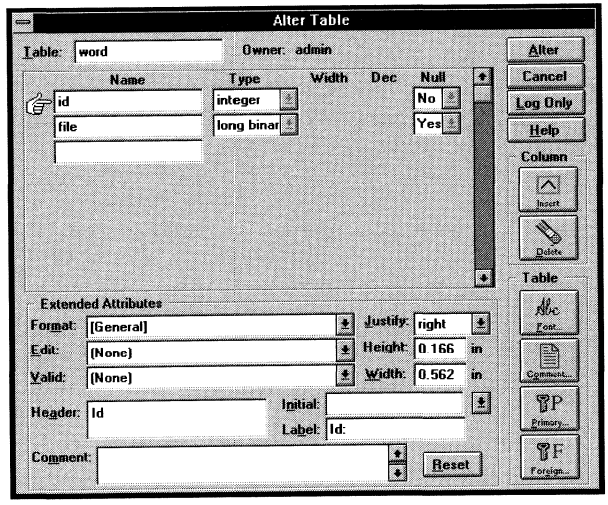
1 In the Database painter, create a table to hold the blob (binary large-object) data. The table must have at least two columns: a key column and a column with the blob data type.

The actual data type you choose depends on your DBMS. In Watcom SQL, choose long binary as the data type for the blob column. In SQL Server, choose Image.

For information about data types, see your DBMS documentation.

Define the blob columns as allowing NULLs (this allows you to store a row that doesn't contain a blob).

Here is the definition of the Word table used in the example.

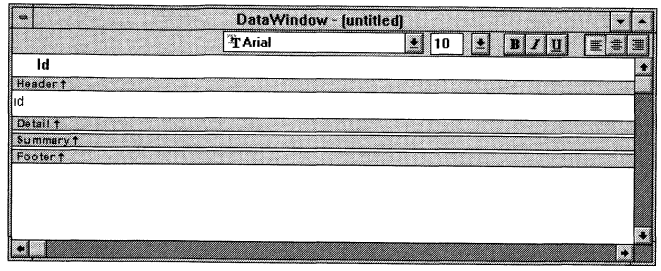


2 Open the DataWindow painter and create a new DataWindow object.

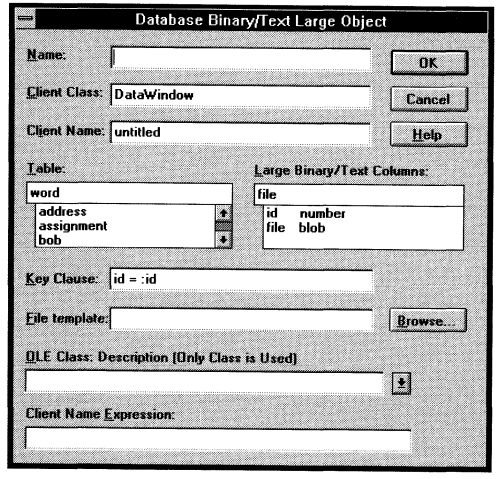
3 Specify the table containing the blob as the data source for the DataWindow object. Be sure to include the key column, but *do not include the blob column in the data source* (you will add it later in the DataWindow painter workspace).

In the example, you would choose ID as the only column.

The DataWindow painter workspace displays.



4 Add the blob column to the DataWindow object by selecting OLE Database Blob from the Objects menu and clicking the place in the DataWindow where you want the blob object. The Database Binary/Text Large Object dialog box displays.



5 (Optional) Enter a name for the object in the Name box. You don't need to name the object, but doing so allows you to refer to it in scripts.

6 (Optional) Enter the client class in the Client Class box. The default is DataWindow.

This value is used in some OLE server applications to build the title that displays at the top of the server window.

7 (Optional) Enter the client name in the Client Name box. The default is Untitled.

This value is used in some OLE server applications to build the title that displays at the top of the server window.

8 In the Table box, select the database table that contains the blob database column you want to place in the DataWindow.

The names of the columns in the selected table display in the Large Binary/Text Columns listbox.

9 In the Large Binary/Text Columns box, select the column that contains the blob data type from the list.

10 If necessary, change the default key clause in the Key Clause box.

PowerBuilder uses the key clause to build the WHERE clause of the SELECT statement used to retrieve and update the blob column in the database. It can be any valid WHERE clause.

Use colon variables to specify DataWindow columns. For example, if you enter this key clause:

```
id = :id
```

the WHERE clause will be:

```
WHERE id = :id
```

11 Identify the OLE server application by doing one of the following:

- If you always want to open the same file in the OLE server application, enter the name of the file in the File Template box.

  For example, to specify a particular Word for Windows document, enter the name of the DOC file. If the file is not on the current path, enter the fully qualified name.

  Tip
  If you do not know the name of the file you want to use, click the Browse button to display a list of available files. Select the file you want from the resulting window.

- If you do not want to open the same file each time, select an OLE server application from the OLE Class: Description dropdown listbox.

  If your OLE server application does not appear in this list, run the Windows RegEdit utility to add it.

  For more information about RegEdit, see the Windows online Help REGEDIT.HLP and REGEDITV.HLP.

12 Enter text or an expression that evaluates to a string in the Client Name Expression box.

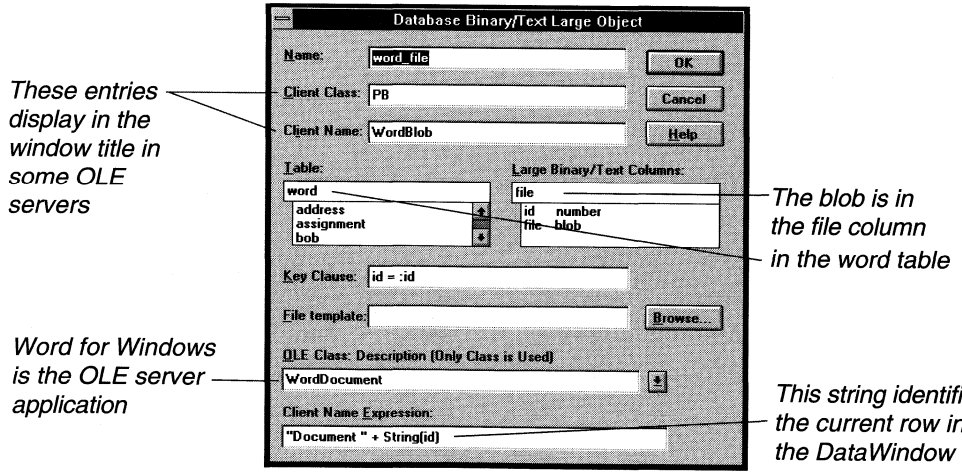
This expression will display in the title of the window in the OLE server application and can be used to identify the current row in the DataWindow.

> **Tip**
> Use an expression to make sure the name is unique. For example, you might enter the following expression to identify a document (where ID is the integer key column):
>
> ```
> "Document " + String(id)
> ```

13 Click OK.

PowerBuilder closes the Database Binary/Text Large Object dialog box and displays the DataWindow workspace. The blob column is represented by a box labeled Blob.

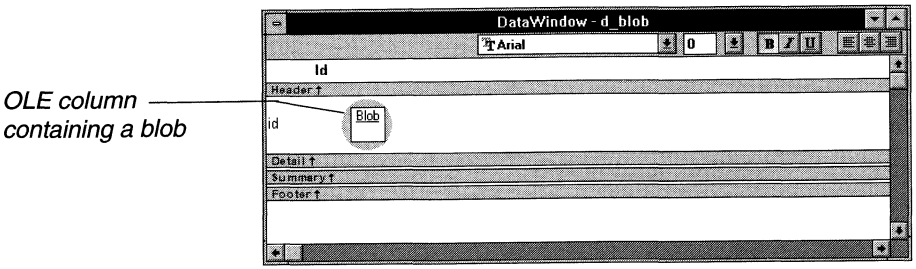
Here is the completed Database Binary/Text Large Object dialog box for the example.



Making the blob column visible

In some cases, the blob column is invisible in the DataWindow object until you activate the OLE server. To make it easy to find the blob column, you can place a DataWindow drawing object behind the blob object. When the DataWindow displays, the drawing object will indicate the location of the blob column until the user double-clicks the column to open the server application.

Here is the completed workspace for the example, showing the blob column in place, with a gray oval behind it.

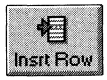


Previewing an OLE column

Before using the DataWindow object in an application, you should preview it to see how it works.

❖ To preview an OLE column:

1 Click the Preview icon or select Preview from the Design menu.



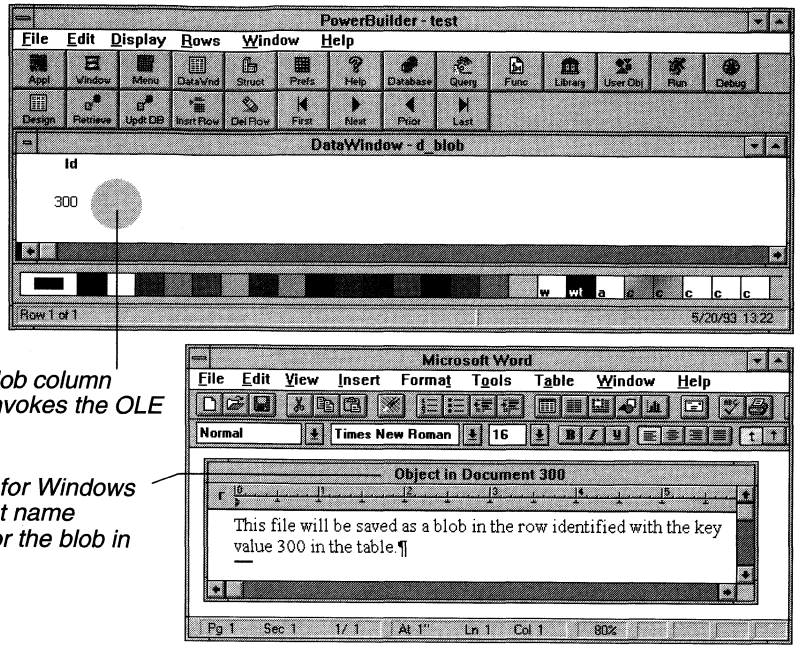
2 Click the Insert Row icon.

PowerBuilder adds a blank row.

3 Enter a value in the key column.

4 Double-click the column that contains the blob data type.

The OLE server application starts and displays the file you specified in the File Template box or an empty workspace if you only specified the OLE server name.



Double-clicking the blob column in the DataWindow invokes the OLE server application

The title of the Word for Windows document is the client name expression defined for the blob in the DataWindow

5 Review the file in the OLE server application and make changes if you want.

When you use an OLE column to access an OLE server application, the server application adds an item to its File menu that allows you to update the data in the server application and in the client (the DataWindow). The text of the menu item depends on the OLE server application. In most applications, it is Update.

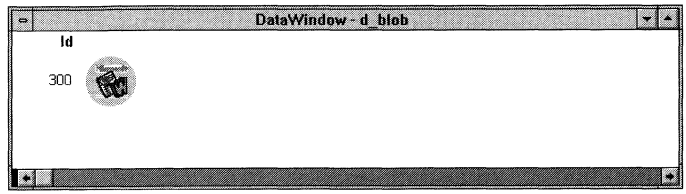
6 Select the menu item in the OLE server that updates the OLE client with the modifications. In the example, you would select Update from the File menu in Word for Windows.

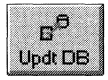
The OLE server application sends the updated information to the DataWindow.

7 Close the file in the server application (typically by selecting Close from the File menu).

8 Return to the DataWindow painter.

The updated blob is represented in the DataWindow using an icon for the OLE server application.





9 To save the blob data in the database, click the Update Database icon.

The new row, including the key value and the blob, are stored in the database.

Later, after you retrieve the rows from the database, you can view and edit the blob by double-clicking it, which invokes the OLE server application and opens the stored document. If you make changes and then update the database, all the modified OLE columns are stored in the database.

Using OLE columns in an application

To use the OLE column in an application, simply place a DataWindow control in a window and associate it with the DataWindow object. 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.

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.

Activating OLE in a DataWindow's script

If you want to activate OLE in an application through a procedure other than the user double-clicking the blob, you can use the OLEActivate function in a script. Calling OLEActivate simulates double-clicking a specified blob.

The OLEActivate function has this syntax:

datawindowname.**OLEActivate** (*row, column, verb*)

Parameter	Description
datawindowname	The name of the DataWindow control containing the OLE column.
row	A long identifying the row of the OLE column.
column	The OLE column; *column* can be the column number (an integer) or a string containing the column name.
verb	The action to perform (see below).

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 RegEdit utility (run REGEDIT /V).

For information about RegEdit, see the Windows online Help REGEDIT.HLP and REGEDITV.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 that 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 is an OLE column):

```
dw_1.OLEActivate(dw_1.GetRow(), 2, 0)
```

For more information

For complete information about OLEActivate, see the *Function Reference*.

Using the OLE 2.0 control in a window

You can place an OLE object directly in a window using the OLE 2.0 control. The control has attributes that specify the control's appearance and how the user interacts with the object it contains. When you place an OLE control in a window, you can link or embed an object into it at development time or leave it empty and insert an object during execution. You can write scripts that allow the application or the user to choose an object for the control.

> **About user objects**
> You can also place OLE 2.0 controls in custom visual user objects. All the information below about using OLE 2.0 controls in windows also applies to OLE 2.0 controls in user objects.
>
> You can also create a standard visual user object of type OLEControl.

This section explains:

- How to define an OLE 2.0 control, select its object, and activate it
- Linking versus embedding the control's object
- Offsite versus in-place activation of the OLE server
- How to control the merging of your menus with the server menus for in-place activation
- How the user interacts with the control

Defining the OLE 2.0 control

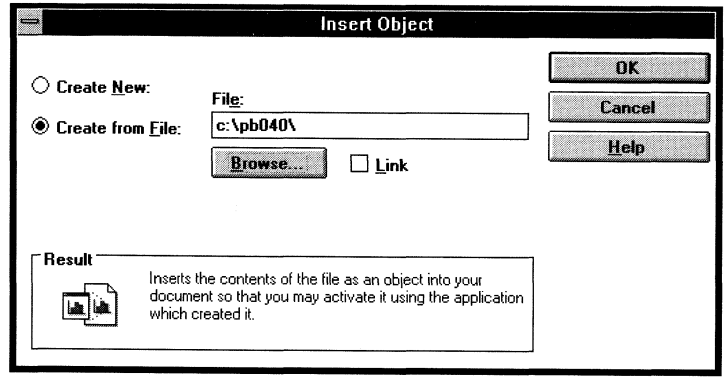
❖ **To place an OLE 2.0 control in a window:**

1. Open the Window painter and select the window that will contain the OLE 2.0 control.
2. Click the OLE 2.0 button in the PainterBar.
3. Click where you want the control.

   PowerBuilder displays an empty rectangle and then displays the Insert Object dialog box. You can choose a server application or a specific object for the control, or you can leave the control empty.

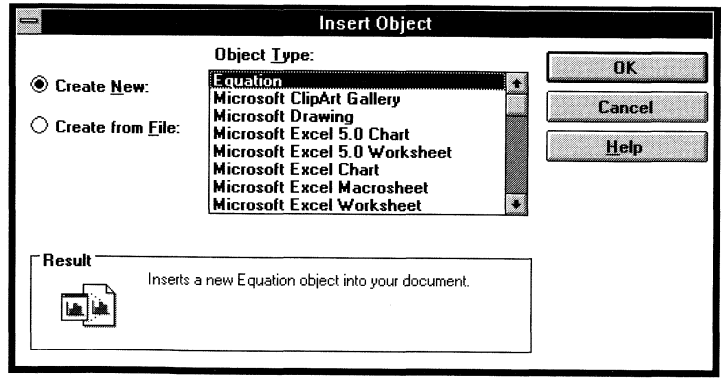
- To choose an existing object for the control, select Create from File. In the File box, specify the filename. If you do not know the name of the file, click the Browse button and select a file in the dialog box.

  If you want to create a link to the file, rather than embed a copy of the object in the control, select the Link checkbox. (See "Linking versus embedding" on page 418 for more information.)



  After you've specified the file, click OK in the Insert Object dialog box. PowerBuilder activates the server application so you can view and edit the object.

- To create and embed a new object, select Create New.



  You see the list of OLE server applications installed on your system. Choose the server application from the list and click OK. PowerBuilder activates the server application so you can edit the new object. A new object is always embedded.

- To leave the control empty, click Cancel. You don't have to choose an OLE object now.

If your OLE server application is not in the list, see the server documentation to find out how to install it.

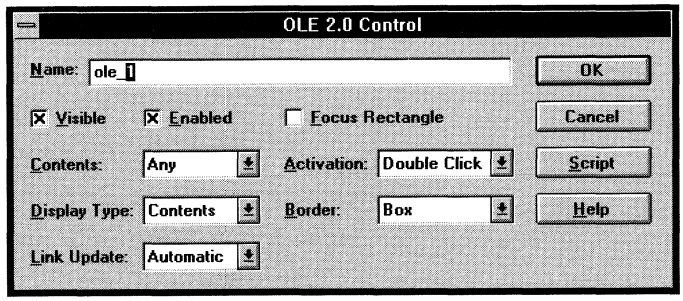
❖ To specify the control's appearance and behavior:

1 Double-click on the control.

or

Select Name from the control's popup menu.

The OLE 2.0 Control dialog box 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 the control's appearance and behavior by choosing appropriate settings in the dialog box.

In addition to the Visible, Enabled, Focus Rectangle, and Border attributes, which are available for most controls, there are several options that control the object's interaction with the server.

4 Click OK when you are done.

The options that are unique to the OLE 2.0 control are described in this table.

Option	Meaning
Contents	Whether the control's object is linked or embedded. The default is Any, which allows either method. If you choose Linked, the Link checkbox is automatically selected in the Insert Object dialog box. If you choose Embedded, the Link checkbox is not available.
Display Type	What the control displays. You can choose: ♦ 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.
Link Update	When the object in the control is linked, the method for updating link information. Choices 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 reestablish the link in a script using the LinkTo function.
Activation	How the control is activated. Choices are: ♦ Double Click — When the user double-clicks on 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 only be activated programmatically with the Activate function. The control can always be activated programmatically, regardless of the Activation setting.

Activating the object

The object in the OLE control needs to be activated to manipulate it using the server application. Double-clicking is the default method for activating the object during execution. You can choose other methods by setting the control's Activation attribute, as described in the previous table. During development, you activate the object in the Window painter.

❖ To activate an OLE object in the Window painter:

1. Choose Object➤Open from the control's popup menu.

   If the control is empty, Open is unavailable. You must choose 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.

Changing the object in the control

In the Window painter, you can change or remove the object in the control.

❖ To delete the object in the control:

- Choose Object➤Delete from the control's popup menu.

  The control is now empty and cannot be activated.

❖ To change the object in the control:

1. Choose Object➤Insert from the control's popup menu.

   PowerBuilder displays the Insert Object dialog box.

2. Choose Create New and select a server application or choose Create from File and specify a file, as you did when you defined the control.

3. Click OK.

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. 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. To change the embedded data, you have to rebuild the application.

Embedding is suitable for data that will not change, for example, the body of a form letter, or as a starting point for data that will be changed and stored elsewhere.

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 actual data is stored 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, and the data can be changed even if your PowerBuilder application is the only one using the data.

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).

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.

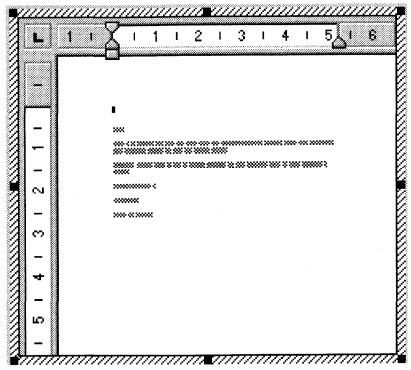
If the link is broken because the file has been moved, renamed, or deleted, the Update setting of the control determines how the problem is handled. When Update is set to Automatic, PowerBuilder displays a dialog box that prompts the user to find the file. You can establish a link in a script using the 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 421).

When the control is active in place, it is 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, with a few extra OLE-specific items such as Update. 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 won't activate

You cannot activate an empty control, that is, a control that doesn't 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, you cannot activate the control. If the Update attribute is set to Automatic, PowerBuilder displays a dialog box so that the user can find the 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 In Place settings in the Menu painter let you control how the menus of the two applications are merged. In Place has values that are standard menu names, as well as the choices, Merge and Exclude.

To control what happens to a MenuItem in your PowerBuilder application, select it and choose the appropriate In Place setting, as described in the following table. You can choose any of the options for items on the menu bar. The settings only apply to items on the menu bar, not items on dropdown menus.

In Place setting	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.	

The effect of the In Place 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. The menu bar also includes other menus that the server has decided are appropriate.

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 may only see 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 choose the server's Update command before ending the offsite editing session.

Manipulating OLE objects in scripts

In "Using OLE columns in a DataWindow" on page 412, you learned how to activate the OLE object in a DataWindow column for offsite editing. When the object is an OLE 2.0 object, the OLE 2.0 standards allow you to do much more. You can modify:

- An object in a control, by activating it and using the command set provided by the server to manipulate the object
- An object in memory, by defining an OLEObject variable and using the server's command set to manipulate the object without displaying it to the user

Using the server's command set to manipulate the object is called OLE **automation**. The server determines what commands are available.

This section describes these methods of modifying the OLE object.

Modifying an object in an OLE control

When an OLE object is displayed in the OLE 2.0 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 the object in the control
- Use the server's functionality to manipulate the object

It also introduces:

- The principles behind the qualifiers for server commands
- The OLE 2.0 control's events that respond to server messages

Activating the OLE object

Generally, the OLE 2.0 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 attribute 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)
```

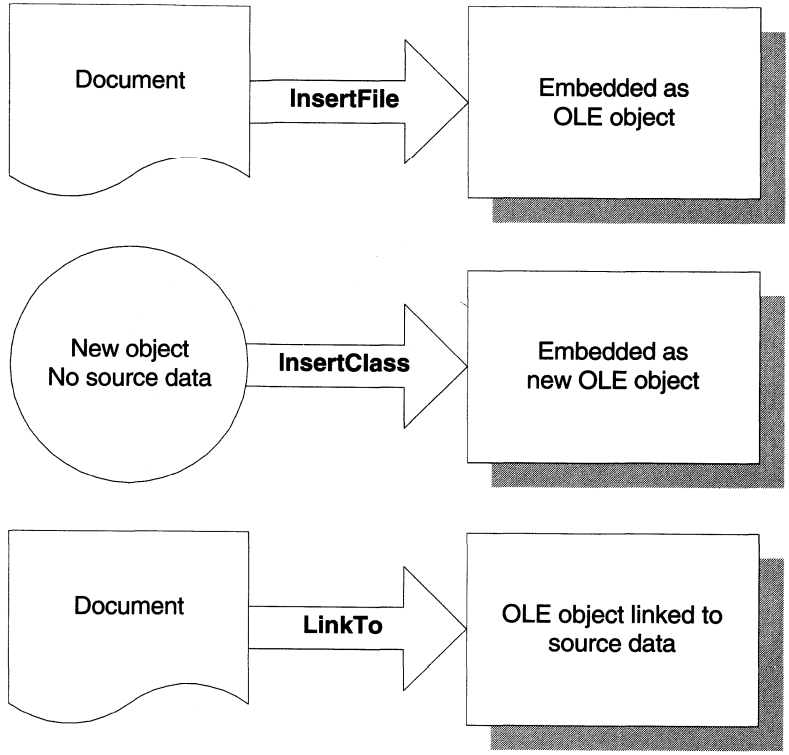
Check the server's documentation to see what verbs it supports. OLE verbs are a relatively limited means of working with objects; OLE automation provides a more flexible interface (see page 428). OLE 1.0 servers only support verbs, not automation.

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.

Choose the function	When you want to
InsertObject	Let the user choose an object and, if the control's Contents attribute is set to Any, whether to link or embed it.
InsertClass	Create a new object for a specified server and embed it in the control.
InsertFile	Embed a copy of an existing object in the control.
LinkTo	Link to an existing object in the control.
Open	Open an existing object from a file or storage. Information in the file determines whether the object is linked or embedded.

The following figure illustrates the behavior of the three functions that don't allow a choice of linking or embedding.



You can also assign OLE object data stored in a blob to the ObjectData attribute of the OLE 2.0 control.

```
blob myblob
... // Code to assign OLE data to the blob
ole_1.ObjectData = myblob
```

The Contents attribute 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 attribute doesn't allow, the function will fail.

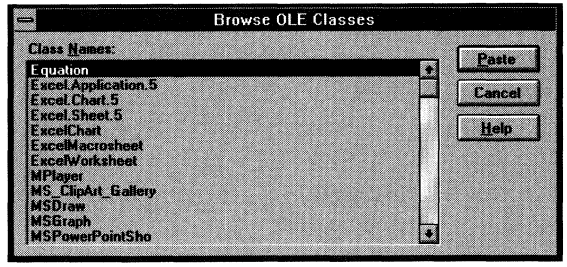
OLE Class browser

Use the OLE Class 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 "Modifying OLE objects in memory" on page 431).

❖ To find a class in the OLE Class browser:

- In the Script painter, choose Edit➤Browse OLE Classes.

  The Browse OLE Classes dialog box displays.



The list includes all available classes. Some of the classes are for OLE automation and some are for inserting objects in containers.

Using the clipboard

Using the Cut, Copy, and Paste functions in MenuItem 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 MenuItems.)

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.

Choose the function	When you want to
Paste	Embed the object on the clipboard in the control
PasteLink	Paste and link the object on the clipboard
PasteSpecial	Allow the user to choose whether to embed or link the pasted object

Linking an object
The server application provides object information about the source of OLE data. In most cases, PowerBuilder runs the server when it needs that information. However, when you paste an object from the clipboard, PowerBuilder doesn't know what server is involved. Therefore, the server application must be running to establish the link.

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 filename 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  myfilename, 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 attribute. 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 attribute 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 *PowerScript Language*).

You can also use SaveAs and Save to store OLE objects in PowerBuilder's OLEStorage variables (see "Opening and saving storages" on page 442).

When the user saves a linked object in the server, the link information is not affected and you don't 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.

Using the server application's functionality

OLE server applications publish the command set they support for OLE automation. Check your server application's documentation for information. The command set will include attributes and methods (functions).

> **Required parentheses**
> PowerScript considers all commands to the server either attribute settings or functions. To distinguish methods and functions from attribute settings, they must be followed by parentheses surrounding the parameters. If there are no parameters, specify empty parentheses.

You apply server commands to the object in the control by referring to its Object attribute using the following syntax:

olecontrolname.**Object**.*methodorfunction* ({ *arguments* })

olecontrolname.**Object**.*attributesyntax* = *value*

For example, the following commands for an Excel spreadsheet object activate the object and set the value attribute 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
```

Compiling scripts that include commands to the OLE server
When you compile scripts that apply methods to the control's Object attribute, PowerBuilder doesn't check the syntax of the rest of the command because it doesn't 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.

Microsoft Word 6.0 supports OLE automation with a command set similar to its Word Basic macro language. PowerBuilder supports functions, not statements, for OLE automation and it doesn't support named parameters. Therefore, for Word, you must follow an automation command with parentheses so that PowerBuilder sends it as a function (even if Word Basic considers it a statement), not an attribute setting. In the parentheses, specify the parameter values without the parameter names. For example, the following statements insert text at a bookmark.

```
ole_1.Activate(InPlace!)
Clipboard(mle_nameandaddress.Text)
ole_1.Object.application.wordbasic.&
   editgoto("NameandAddress")
ole_1.Object.application.wordbasic.editpaste(1)
```

The last two commands in a Word Basic macro would look like:

```
EditGoto .Destination = "NameandAddress"
EditPaste
```

OLE automation is not macro programming
You cannot send commands to the server application that require the allocation of memory. That is, you cannot define a variable on the server. OLE automation executes one function at a time and doesn't provide access to the program control statements of Word's macro language. Use PowerScript's conditional and looping statements to control program flow.

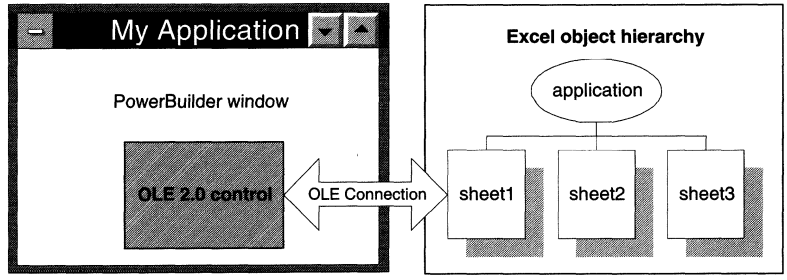
To illustrate how to combine PowerScript with server commands, the following script counts the number of bookmarks in a Microsoft Word 6.0 OLE object and displays their names.

```
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 the 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 looks like the following.



When the server is Excel, the following commands seem to mean the same thing but can have different effects.

```
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 doesn't matter whether it is the same one in the PowerBuilder control. If the user switches to Excel and activates a different sheet, the script will change that one instead. Therefore, you should avoid this syntax.

The second statement affects only the document in the PowerBuilder control. However, it will cause an execution 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 implements the application hierarchy differently and requires 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.&
   editgoto("NameandAddress")
```

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 "Modifying OLE objects in memory" on page 431.)

Events for the OLE 2.0 control

There are several events that let PowerBuilder know when actions take place in the server application that affect the OLE object. They are:

- **DataChange** The data has been changed.
- **Rename** The object has been renamed.
- **Save** 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.

Modifying OLE objects in memory

You don't need to place an OLE control on a window to manipulate an OLE object in a script. If the object doesn't 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 attributes for that object. The server application executes the functions and changes the object's attributes, 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 OLE automation. When you connect to an OLE object, the OLEObject variable stores a reference to the actual object—therefore it takes little memory. The actual object data is stored on the server side of the OLE conversation.

OLEObject is a dynamic object type, which means that the compiler will accept any attribute names, function names, and parameter lists for the object. PowerBuilder doesn't have to know whether the attributes and functions are valid. This allows you to call methods and set attributes for the object that are known to the server application that created the object. If the functions or attributes don't exist during execution, you will get execution errors.

You need to declare an OLEObject variable and allocate memory for it.

```
OLEObject myoleobject
myoleobject = CREATE OLEObject
```

Connecting to the server

You establish a connection between the OLEObject object and an OLE server with the ConnectToObject or ConnectToNewObject function. Connecting to an object starts the appropriate server.

Choose the function	When you want to
ConnectToNewObject	Create a new object for an OLE server that you specify. Its purpose is similar to InsertClass for a control.
ConnectToObject	Open an existing OLE object from a file. If you don't specify an OLE class, PowerBuilder uses the file's extension to determine what server to start.

After you establish a connection, you can use the server's command set for OLE automation to manipulate the object (see "Using the server application's functionality" on page 428).

You don't 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 connect to Microsoft Word's OLE interface, word.basic, and execute the editgoto and editpaste functions.

```
myoleobject.ConnectToNewObject("word.basic")
myoleobject.editgoto("NameandAddress")
myoleobject.editpaste(1)
```

Do not include word.basic (which is the class in ConnectToNewObject) as a qualifier:

```
// Incorrect command qualifiers
myoleobject.word.basic.editgoto("NameandAddress")
```

Microsoft Word implementation note
For an OLEObject variable, word.basic is the class name of Word 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.

Disconnecting from the server

After your application is finished with the OLE automation, you must disconnect from the server and release the memory for the object.

```
rtncode = myoleobject.DisconnectObject( )
DESTROY myoleobject
```

Disconnecting is important
If your OLEObject variable goes out of scope before you disconnect, you cannot close the server application. If the application is visible, the user can activate it and exit. However, if it is invisible, there is no way for your program or the user to close the server application. Therefore, remember to disconnect the OLEObject variable.

Declaring an untyped variable

Because PowerBuilder knows nothing about the commands and functions of the server application, it also knows nothing about the data types of returned information. PowerBuilder provides a general data type called *any* to which you can assign data from the server. The any data type circumvents PowerBuilder type-checking during compilation and acts as a placeholder for the actual data.

During execution, when data is assigned to the variable, it becomes a variable of the appropriate data type. You can use the ClassName function to determine the data type of the any variable and make appropriate assignments. If you make an incompatible data assignment with mismatched data types, you will get an execution error.

Don't use the any data type unnecessarily
If you know the data type of data returned by a server automation function, don't use the any data type. 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 data type.

```
string stringval
integer intval
any anyval

anyval = myoleobject.application.cells(1,1).value

CHOOSE CASE ClassName(anyval)
CASE "string"
   stringval = anyval
CASE "int"
   intval = anyval
...
END CHOOSE
```

OLE automation scenario

The steps involved in OLE 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 list boxes or edit boxes.
- Disconnect and destroy the object in the window's Close event.

If the automation doesn't involve the user, all the work can be done in a single script, as shown in the following form letter example.

Example: generating form letters via OLE

This example takes names and addresses from a DataWindow and letter body from a MultiLineEdit and creates and prints letters in Microsoft Word.

❖ 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:

- 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

Multimedia Promotions Inc.
1234 Technology Drive
Westboro, Massachusetts

January 12, 1995

[bookmark name1]
[bookmark address1]

Dear [bookmark name2]:

[bookmark body]

Sincerely,

Harry Mogul
President

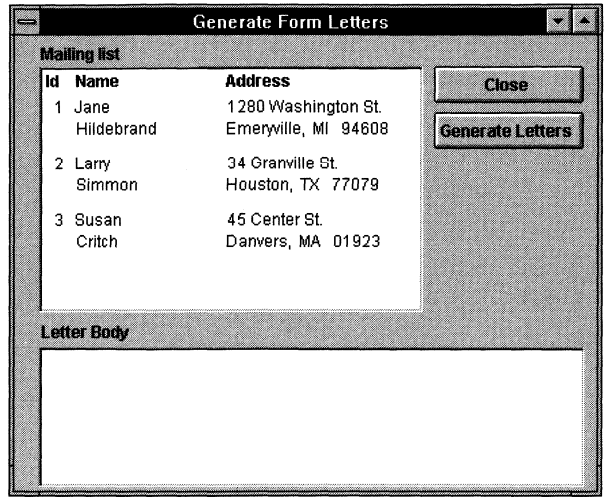
You could spruce up the letter with a company and a signature logo, if you want. The important items are the names and placement of the bookmarks.

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 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.



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. The following code connects to a Watcom database. (When the window is part of a larger application, the connection would typically be done by the application Open script.)

```
/************************************************
Set up the transaction object from the INI file
************************************************/
SQLCA.DBMS=ProfileString("PB.INI",
   "Database", "DBMS", "  ")
SQLCA.DbParm=ProfileString("PB.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.basic)
- For each row in the DataWindow, it generates a letter. To do so, it performs the following tasks:
  - Opens the document with the bookmarks (using the Word Basic statement fileopen)
  - Extracts the name and address information from a row in the DataWindow and inserts it into the appropriate places in the letter (editgoto and insert)
  - Inserts the text the user types in mle_body into the letter (editgoto and insert)
  - Prints the letter (fileprint)
  - Closes the letter document without saving it (fileclose(2))
- 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.basic")
```

```
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.fileopen("c:\pb040\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.editgoto("name1")
   contact_ltr.insert(ls_name)
   contact_ltr.editgoto("name2")
   contact_ltr.insert(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.editgoto("address1")
   contact_ltr.insert(ls_addr)

   /**********************************************
    Insert the letter text at the body bookmark
   **********************************************/
   contact_ltr.editgoto("body")
   contact_ltr.insert(mle_body.Text)
```

```
        /*********************************************
        Print the letter
        *********************************************/
        contact_ltr.fileprint()

        /*********************************************
        Close the document without saving
        *********************************************/
        contact_ltr.fileclose(2)
    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 press CTRL+SHIFT+W to run the window.

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.

Advanced ways to manipulate OLE objects

In addition to OLE objects in controls and objects for OLE automation, PowerBuilder provides an interface to the underpinnings of OLE data storage.

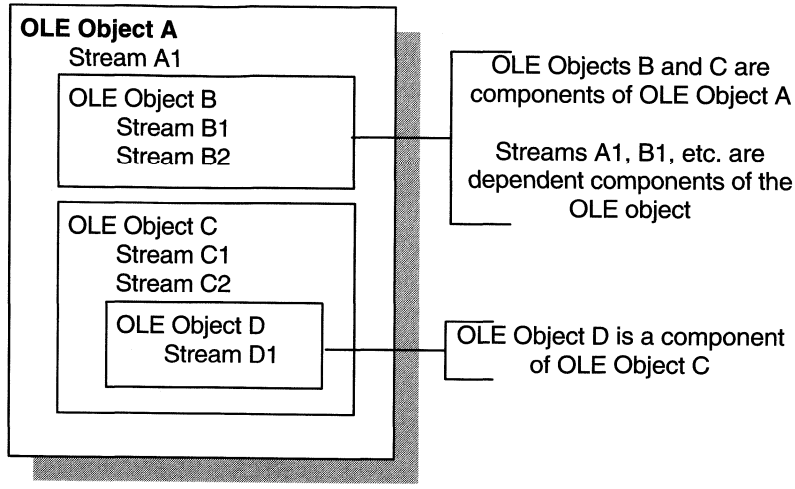
OLE data is stored in objects called **streams**, which live in objects called **storages**. Streams and storages are analogous to the files and directories of a file system. By opening, reading, writing, saving, and deleting streams and storages, you can create, combine, and delete your OLE objects. PowerBuilder provides access to storages and streams with the OLEStorage and OLEStream object types.

When you define OLE 2.0 controls and OLEObject variables, you have full access to the functionality of server applications and OLE automation, which already provide you with much of OLE's power. You may 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, the substorages can form a larger OLE object, such as a document that includes pictures.



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 only be manipulated by that server application. You might open a storage that is a server's object to extract an object within the storage but you shouldn't 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 doesn't exist, PowerBuilder creates it.

You can put OLE objects in a storage with the SaveAs function. You can establish a connection between an OLE 2.0 control in a window and a storage by calling the Open function for the OLE 2.0 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.

When you want to access OLE data in a file, call the Open function. Depending on the structure of the storage file, you may need to call Open more than once.

This code opens the root storage in the file into the control. 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")
```

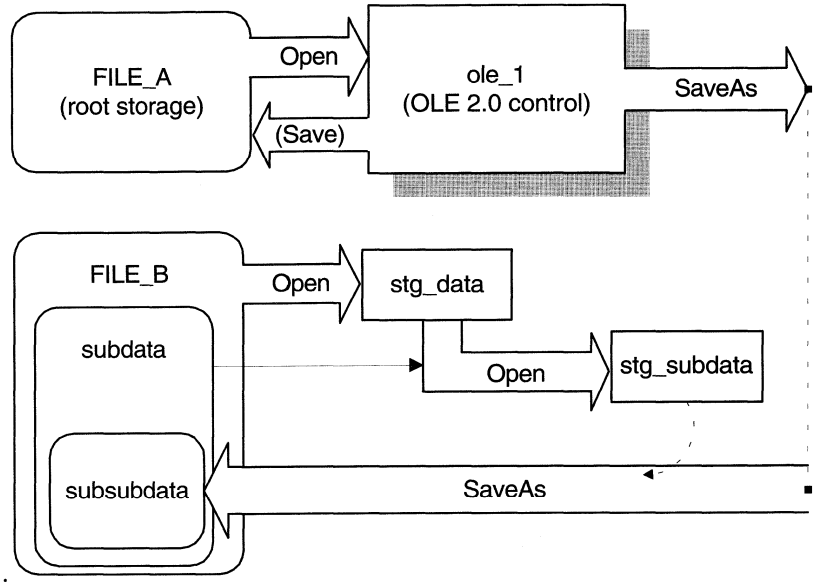
If the user activates the object in the control and edits it, then the server saves changes to the data in memory and sends a DataChange event to your PowerBuilder application. Then your application needs to call Save to make the changes in the storage file.

```
result = ole_1.Save()
IF result = 0 THEN result = stg_data.Save()
```

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(stg_data, "subdata")
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 did not exist, Open and SaveAs would 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.



Getting information about storage members

When a storage is open, you can use one of the Member functions to get information about and change the substorages and streams in that storage.

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 can't 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.

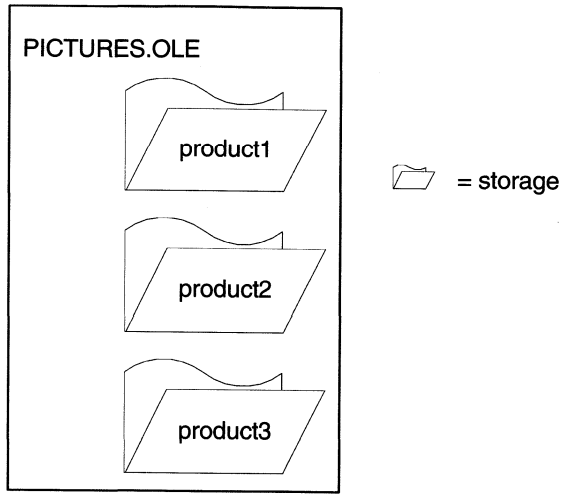
This code checks whether the storage subdata exists in stg_data before it opens it. (The code assumes that stg_data and stg_subdata have been declared and instantiated.)

```
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
```

Example: building a storage

Suppose you have several Visio drawings of products and you want to display the appropriate image for each product record in a DataWindow. The database record has an identifier for its drawing. In an application, you could call InsertFile using the identifier as the filename. 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.



The advantage to 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 doesn't need to start up to display each picture.

OLE objects in the storage
Although this example illustrates a storage file that holds Visio drawings only, the storages in a file don't 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 and an OLE 2.0 control. The DataWindow, called dw_prodid, has a single column of product identifiers. You should set up the database table so that the identifiers correspond to the filenames of the Visio product drawings. The OLE control displays the drawings.

The example has three scripts:

- The window's Open event script instantiates the storage variable, opens the storage file, and retrieves data for the DataWindow. (Note that the application's Open event connects to the database.)
- The RowFocusChanged event of the DataWindow opens the Visio drawing and saves it in the storage file.
- The window's Close event script saves the storage file and destroys the variable.

First, declare an OLEStorage variable as a instance variable of the window.

```
OLEStorage stg_prod_pic
```

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 come 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.

The InsertFile function displays the Visio drawing in the OLE control. This code in the RowFocusChanged event gets an identifier from the prod_id column in a DataWindow and uses that to build the drawing's filename 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 = dw_prodid.GetItemString(  &
   dw_prodid.GetRow(), "prod_id" )

/*************************************************
Use the id to build the filename. Insert the file's
object in the control.
*************************************************/
result = ole_product.InsertFile(  &
   "c:\visio\drawings\" + prodid + ".vsd")

/*************************************************
Save the OLE object to the storage. Use the same
identifier to name the storage.
*************************************************/
prodid = dw_prodid.GetItemString(  &
   dw_prodid.GetRow(), "prod_id" )
result = ole_product.SaveAs( stg_prod_pic, prodid)
```

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
```

In the application's Open event, connect to the database and open the window.

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 Visio drawing for each product identifier, run the application. As you scroll through the DataWindow, the application opens each Visio 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 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
prodid = dw_prodid.GetItemString(  &
   dw_prodid.GetRow(), "prod_id" )

//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.

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.
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.

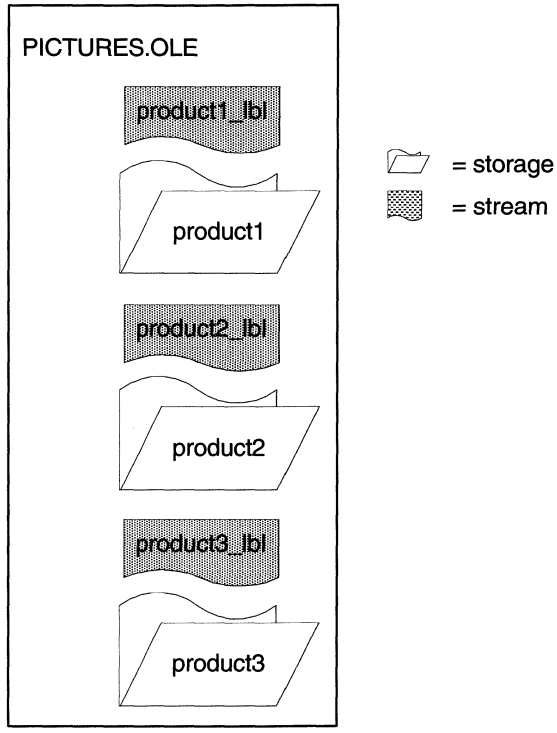
After you have finished with the OLEStream variable, you should release its memory with the DESTROY statement.

```
DESTROY stm_pic_label
```

Example: writing and reading streams

This example displays a picture of a product in the OLE control ole_product when the DataWindow 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 445). 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". The diagram shows the structure of the file.



The example has three scripts:

- The window's Open event script opens the storage file and retrieves data for the DataWindow. (Note that the application's Open event connects to the database.)
- The RowFocusChanged event of the DataWindow 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.
**************************************************/
prodid = dw_prod.GetItemString(  &
   dw_prod.GetRow(), "prod_id" )

/**************************************************
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
   END IF
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 philosophically opposed to storing data in a database. A storage file doesn't 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 doesn't support a blob data type or if your database administrator doesn't 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 16

Building a Mail-Enabled Application

About this chapter

This chapter describes how you can use the messaging application program interface (MAPI) with PowerBuilder applications to send and receive electronic mail.

Contents

Overview

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 during the application.
- Send mail requesting information.
- Receive mail containing information needed by the application's user.

How MAPI support is implemented

To support MAPI, PowerBuilder provides:

- A mail-related system object—MailSession
- Mail-related structures—MailFileDescription, MailMessage, and MailRecipient
- Object-level functions for the MailSession object—MailAddress, MailDeleteMessage, MailGetMessages, MailHandle, MailLogoff, MailLogon, MailReadMessage, MailRecipientDetails, MailResolveRecipient, MailSaveMessage, and MailSend
- Enumerated data types—MailFileType, MailLogonOption, MailReadOption, MailRecipientType, and 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
```

For more information

You can use the Object 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 data types.

 For information about using the Object browser, see the *User's Guide*.

 For complete information about the MailSession functions, see the *Function Reference*.

 For complete information about MAPI, see the *Microsoft Mail Technical Reference*.

CHAPTER 17

Adding Other Processing Extensions

About this chapter

This chapter describes how to use external functions and how to manage Windows messages in PowerBuilder.

Contents

Using external functions

External functions are functions that are written in languages other than PowerScript and stored in dynamic link libraries (DLLs). You can use external functions that are written in any language that supports the Pascal calling sequence.

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, user object, or user-defined function. 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.

Declaring external functions

❖ **To declare an external function:**

1 Do one of the following:

- To declare a global external function, select Declare➢Global External Functions from the menu bar of any painter that accesses the PowerScript painter.
- To declare a local external function, open a window, menu, user object, or user-defined function in a painter and select Declare➢Local External Functions from the menu bar.

The resulting dialog box displays the declared external functions.

2 Enter the function declaration in the edit box. See *PowerScript Language* or the examples below for the syntax to use.

3 Click OK.

PowerBuilder compiles the declaration. If there are syntax errors, an error message displays, and you must correct the errors before PowerBuilder will save the declaration.

Tip
You can also modify existing external function declarations in the edit box.

Examples of declarations

The following statements declare two external C functions that are stored in one of the standard Windows DLLs, KERNEL.EXE. You can use them to determine whether another Windows program is running:

```
FUNCTION long GetModuleHandle(string modulename)  &
   LIBRARY "KERNEL.EXE"
FUNCTION boolean GetModuleUsage(long hWnd)  &
   LIBRARY "KERNEL.EXE"
```

The following statement declares an external C function named IsZoomed stored in one of the standard Windows DLLs, USER.EXE. The function takes one argument (an integer window handle) and returns a boolean result: TRUE if the window is maximized; otherwise, FALSE:

```
FUNCTION boolean IsZoomed(Int handle)  &
   LIBRARY "USER.EXE"
```

(A script that uses IsZoomed is included as an example in the "Utility functions" section at the end of this chapter.)

For more information about these functions, see the Microsoft Windows Software Developer's Kit (SDK) documentation.

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*

PowerBuilder supports passing FAR pointers only. Therefore, the pointer in the external function must have the FAR qualifier as shown in the examples below.

The syntax for an argument that is passed by value is:

datatype arg

Passing numeric data types

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 far pascal TEMP(int far * 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 far pascal 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

Passing by value

The following statement declares the external C function NAME in PowerBuilder. This function expects a string argument 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 far * far pascal NAME(char far * 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 PowerBuilder a pointer to a string (that is, 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 far * far pascal NAME2(char far * 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.

Example

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 *Function Reference.*

Passing chars to C functions

Char variables passed to external C functions are converted to the C char type before passing. Arrays of chars are converted to the equivalent C array of chars.

An array of chars embedded in a structure produce 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.

Sending Windows messages

To send Windows messages to a window that you created in PowerBuilder or an external window (for example, a window you created using an external function), use the Post or Send function. To trigger a PowerBuilder event, use 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 a synchronous function: the window or control receives the message immediately.

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.

For more information about Post and Send, see the *Function Reference.*

Triggering PowerBuilder events

To trigger a PowerBuilder event, you should use the TriggerEvent or PostEvent function. Both functions bypass the messaging queue and are easier to code than the Send and Post functions.

TriggerEvent is a synchronous function: the event is triggered immediately in the window or control. PostEvent is asynchronous: the event is posted to the event queue for the window or control.

Example

Both 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!)
```

The TriggerEvent function identifies the object in which the event will be triggered and then uses the enumerated data type Clicked! to specify the clicked event.

For more information about TriggerEvent and PostEvent, see the *Function Reference*.

The Message object and the Other event

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. If there is no script for a user-defined event corresponding to the Windows event, PowerBuilder also triggers the Other event in the object.

The Other event receives most of the Windows messages that are not PowerBuilder events and can be triggered in all PowerBuilder objects except the application object.

Other uses of the Message object

The Message object is also used in the following ways:

- To communicate parameters between windows when you open and close them.

  For more information, see the descriptions of OpenWithParm, OpenSheetWithParm, and CloseWithReturn in the *Function Reference.*

- If optional parameters were used in TriggerEvent or PostEvent.

  For more information, see the *Function 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 attributes (instance variables) and functions. You then populate the user-defined attributes and call the functions as needed in your application.

For more information about defining standard class user objects, see the *User's Guide.*

Message object attributes

The first four attributes of the Message object correspond to the first four attributes of the Microsoft Windows message structure:

Attribute	Data type	Use
Handle	Integer	The handle of the window or control.
Number	Integer	The number that identifies the event (this number comes from Windows).
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 or the Other 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.

To process these messages, test the values in the Message object in the script for the user-defined event or the Other event for the object in which the event occurred.

For information on Microsoft message numbers and parameters, see the Microsoft Software Developer's Kit (SDK) documentation.

Using utility functions to manage information

The utility functions provide you with a way to obtain and pass Windows information to external functions and can be used as arguments in the PowerScript Send function. The utility functions are listed in the following table:

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 the high word into a long. Long is used to pass values to external functions.

For more information about these functions, see the *Function Reference*.

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, then return it to active:

```
// Loop counter
int   nLoop
// Handle to window object
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)
```

PART FIVE

Miscellaneous Techniques

A collection of techniques you can use to implement miscellaneous features in the applications you develop with PowerBuilder. Includes: printing from an application.

CHAPTER 18

Printing from an Application

About this chapter

This chapter describes how to use predefined functions to create lists and reports.

Contents

Overview

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.

Printing functions

The following table lists all functions concerned with printing.

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.
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.
PrintSend	Sends a specified string directly to the printer.
PrintSetFont	Sets the current font to one of the defined fonts for the current job.
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.
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.

Function	Description
PrintX	Returns the X value of the print cursor.
PrintY	Returns the Y value of the print cursor.

For more information about the printing functions, see the *Function Reference*.

Printing basics

All printing is defined in terms of the print area.

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 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 cursorto keep track of the print location. The print cursor stores the coordinates of the upper-left corner of the location at which printing will begin. 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 will 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 then when the count reaches a certain number (such as 50), call the PrintPage function, reset the counter, and print the title.

Example

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 on the preceding page 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, and 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.

Examples

In these examples, Job is the integer print job number.

The following statement tabs one inch from the left edge of the print area, prints Atlanta, and starts a new line:

```
Print(Job,1000,"Atlanta")
```

The following 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)
```

The following 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.

Example

The first Print statement shown below tabs one inch from the left edge of the print area and prints Powersoft, but it does not move to the next tab (.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 Corporation immediately after the t in Powersoft (PowersoftCorporation) and then starts a new line:

```
Print(Job,1000,"Powersoft",500)
Print(Job," Corporation")
```

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 will 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.

Example

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:

```
Int   JobNbr
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 Job 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 Job. The company name is printed in font 2:

```
Int   Job, Helv18BU
Job = PrintOpen( )
Helv18BU = PrintDefineFont(Job,2,"Helv",250,700, &
   Variable!,Swiss!,FALSE,TRUE)
PrintSetFont(Job,2)
Print(Job,"Powersoft Corporation")
```

For more information about PrintDefineFont and PrintSetFont, see the *Function Reference*.

Setting line spacing

PowerBuilder will take 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

The following 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(Job,1)
```

The following statement causes one-and-a-half-line spacing:

```
PrintSetSpacing(Job,1.5)
```

The following statement causes double spacing:

```
PrintSetSpacing(Job,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, you should 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), and therefore, if you place an object over text or another object, it will hide 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

The following statements draw a one-inch by three-inch rectangle and then print the company address in the rectangle. The rectangle is at the top of the page and centered:

```
Int   Job
Job = PrintOpen( )
PrintRect(Job,2500,0,3000,1000,40)
Print(Job,2525,"")

Print(Job,2525,"25 Mountain Road")
Print(Job,2525,"Milton, MA 02186")
PrintClose(Job)
```

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z