SYBASE®

Resource Guide

PocketBuilder™

2.1

DOCUMENT ID: DC50061-01-0210-01

LAST REVISED: July 2007

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

Contents

About This Book

Audience

This guide is for programmers building applications with PocketBuilder™.

This guide assumes that you have a basic familiarity with Windows CE (Windows Mobile) devices. Although your development work in PocketBuilder is done on a desktop machine, you design applications for use on Windows CE devices such as the Pocket PC or Smartphone.

For information about developing applications for Microsoft Windows CE platforms, see the Microsoft Web site at http://msdn2.microsoft.com/en-us/library/ms950422.aspx. You can also find helpful information at the Pocket PC Developer Network Web site at http://www.pocketpcdn.com.

How to use this book

PocketBuilder is very similar to PowerBuilder®, the Sybase® 4GL development tool for desktop applications and application server components.

If you have never used PowerBuilder, use this book to learn some concepts and principles of programming in PowerScript®, the language used in both PowerBuilder and PocketBuilder, as well as programming techniques you can use with controls in windows and DataWindow® objects. This book also describes how to manage database connections and provides a reference to database connection parameters and preferences.

If you are a PowerBuilder user, the following chapters are probably the most useful:

- Chapter 13, “Database Connectivity in PocketBuilder”
- Chapter 17, “Using MobiLink Synchronization”
- Chapter 19, “Working with Unicode”
- Chapter 20, “Using External Functions and Other Processing Extensions”

Related documents

PocketBuilder documentation The PocketBuilder documentation set also includes the following manuals:

- *Introduction to PocketBuilder* - Provides an overview of PocketBuilder features and the PocketBuilder development environment and a tutorial that leads the new user through the basic process of creating and deploying PocketBuilder applications.
- *User's Guide* - Gives an overview of the PocketBuilder development environment and explains how to use the interface. Describes basic techniques for building the objects in a PocketBuilder application, including windows, menus, DataWindow objects, and user-defined objects. An appendix summarizes the differences between PocketBuilder and PowerBuilder.

PocketBuilder reference set The PocketBuilder reference set is made up of four manuals that are based on PowerBuilder documentation:

- *Connection Reference* - Describes the database parameters and preferences you use to connect to a database in PocketBuilder.
- *DataWindow Reference* - Lists the DataWindow functions and properties and includes the syntax for accessing properties and data in DataWindow objects.
- *Objects and Controls* - Describes the system-defined objects and their default properties, functions, and events.
- *PowerScript Reference* - Describes syntax and usage for the PowerScript language including variables, expressions, statements, events, and functions.

Online Help Reference information for PowerScript properties, events, and functions is available in the online Help with annotations indicating which objects and methods are applicable to PocketBuilder.

SQL Anywhere® Studio documentation PocketBuilder is tightly integrated with SQL Anywhere (formerly Adaptive Server Anywhere), UltraLite®, and MobiLink, which are components of SQL Anywhere Studio. You can install these products from the PocketBuilder setup program. For an introduction to these products, see Chapter 1 in the *Introduction to PocketBuilder*.

Sample applications

The PocketBuilder installation provides a Code Examples workspace with targets that illustrate many of the product's features. Commented text inside events of target objects helps explain the purpose of the sample code. The example workspace is installed in the Code Examples subdirectory under the main PocketBuilder directory.

More applications on the Web

You can find more sample PocketBuilder applications and techniques in the PocketBuilder project on the Sybase CodeXchange Web site at http://pocketbuilder.codexchange.sybase.com/. There is a link to this page on the Windows Start menu at Program Files>Sybase>PocketBuilder 2.0>Code Samples.

If you have not logged in to MySybase, you must log in to the Sybase Universal Login page to access CodeXchange. If you do not have a MySybase account, you can sign up. MySybase is a free service that provides a personalized portal into the Sybase Web site.

Other sources of information

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

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

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

  Refer to the *SyBooks Installation Guide* on the Getting Started CD, or the *README.txt* file on the SyBooks CD for instructions on installing and starting SyBooks.
- The Sybase Product Manuals Web site is an online version of the SyBooks CD that you can access using a standard Web browser. In addition to product manuals, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

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

Sybase EBFs and software maintenance

❖ **Finding the latest information on EBFs and software maintenance**

1. Point your Web browser to the Sybase Support Page at http://www.sybase.com/support.
2. Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.
3. Select a product.
4. Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.

   Padlock icons indicate that you do not have download authorization for certain EBF/Maintenance releases because you are not registered as a Technical Support Contact. If you have not registered, but have valid information provided by your Sybase representative or through your support contract, click Edit Roles to add the "Technical Support Contact" role to your MySybase profile.
5. Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.

Conventions

The formatting conventions used in this manual are:

Formatting example	To indicate
Retrieve and Update	When used in descriptive text, this font indicates: • Command and function names • Keywords such as true, false, and null • Datatypes such as integer and char • Database column names such as emp_id and f_name • User-defined objects such as dw_emp or w_main
variable or *file name*	When used in descriptive text and syntax descriptions, oblique font indicates: • Variables, such as *myCounter* • Parts of input for which you must substitute text, such as *pklname*.pkd • File and path names

Formatting example	To indicate
File>Save	Menu names and menu items are displayed in plain text. The greater than symbol (>) shows you how to navigate menu selections. For example, File>Save indicates "select Save from the File menu."
`dw_1.Update()`	Monospace font indicates: • Information that you enter in a dialog box or on a command line • Sample script fragments • Sample output fragments

If you need help

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

PART 1

Using the PowerScript Language

This part contains an overview of object-oriented features in PocketBuilder, and presents programming techniques for handling PowerScript language features such as inheritance, exception handling, and class definition objects.

CHAPTER 1

Implementing Object-Oriented Programming Techniques

About this chapter

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

Contents

Terminology review

Classes, properties, and methods

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

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

The rest of this chapter uses this PocketBuilder terminology.

Fundamental principles

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

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

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

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

Visual objects

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

Nonvisual objects

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

Standard class user objects Inherit their definitions from built-in PocketBuilder system objects, such as Transaction, Message, or Error. Creating customized standard class user objects allows you to provide powerful extensions to built-in PocketBuilder system objects.

Custom class user objects Inherit their definitions from the PocketBuilder NonVisualObject class. Custom class user objects encapsulate data and code. This type of class user object allows you to define an object class from scratch. To make the most of PocketBuilder's object-oriented capabilities, you must use custom class user objects. Typical uses include:

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

PocketBuilder techniques

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

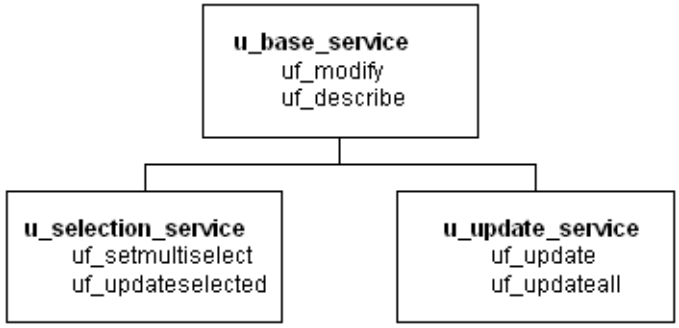
Creating reusable objects
In most cases, the person developing reusable objects is not the same person using the objects in applications. This discussion describes defining and creating reusable objects. It does not discuss usage.

Implementing inheritance

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

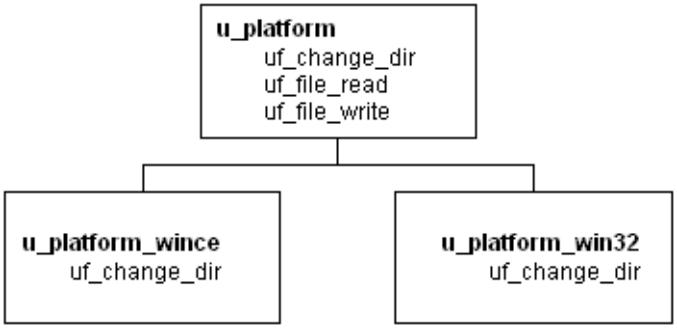
Example of ancestor service object One example of using inheritance in custom class user objects is creating an ancestor service object that performs basic services, and several descendent service objects. The descendent objects perform specialized services and also have access to the ancestor's services:

Figure 1-1: Ancestor service object



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

Figure 1-2: Virtual function in ancestor object



For more on virtual functions, see "Other techniques" on page 8.

Implementing encapsulation

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

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

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

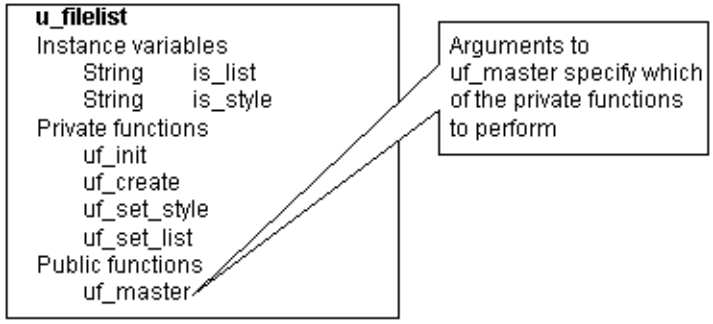
Table 1-1: Defining object functions

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

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

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

Figure 1-3: Defining a public function for encapsulation

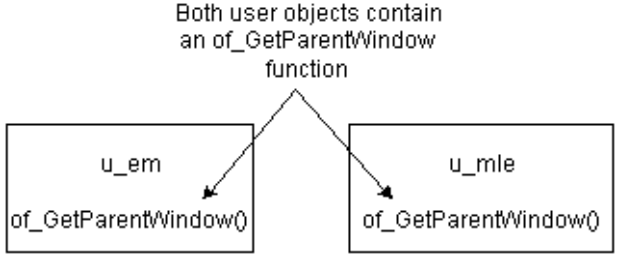


Implementing polymorphism

Polymorphism means that functions with the same name behave differently depending on the referenced object. Although there is some discussion over an exact meaning for polymorphism, many people find it helpful to think of it as follows:

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

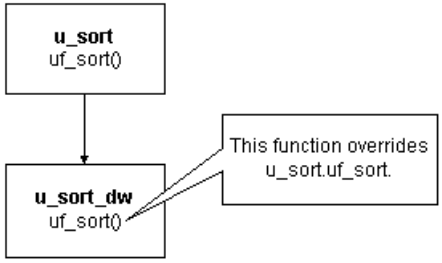
Figure 1-4: Operational polymorphism



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

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

Figure 1-5: Inclusional polymorphism



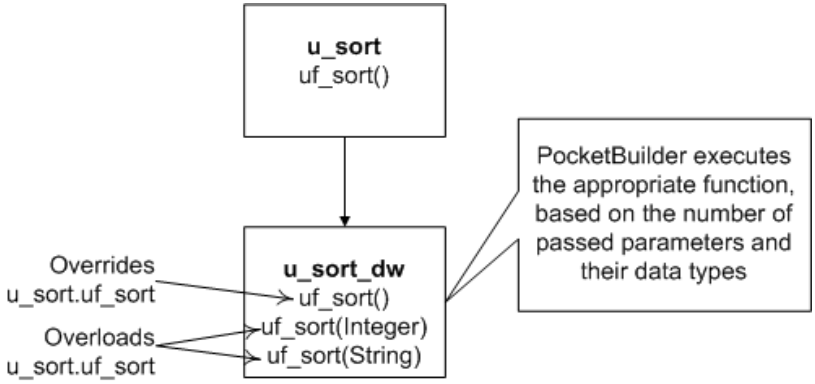
Other techniques

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

Using function overloading

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

Figure 1-6: Function overloading



Global functions
Global functions cannot be overloaded.

Using dynamic versus static lookup

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

In this example, u_platform has two descendent objects, u_platform_wince and u_platform_win. Instantiate the appropriate object at runtime, as shown in the following code example:

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

Although dynamic lookup provides flexibility, it also slows performance.

Static lookup To ensure fast performance, static lookup is a better option. However, PocketBuilder enables object access using the reference variable's datatype (not the datatype specified in a CREATE statement). Therefore, when using static lookup, you must define default implementations for functions in the ancestor. These ancestor functions return an error value (for example, -1) and are overridden in at least one of the descendent objects.

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

For example, using the objects in the previous example, suppose u_platform_wince has a function of_get_this, and u_platform_win has a function of_get_that. Both of_get_this and of_get_that have default implementations in u_platform. These implementations return -1 and are overridden in one of the descendent objects. Then when you use the following statements to call of_get_this, the of_get_this function in the descendant is called:

```
u_platform iuo_platform
iuo_platform = CREATE u_platform_wince
u_platform_wince.of_get_this()
```

Using delegation

Delegation occurs when objects offload processing to other objects.

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

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

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

1. Create a service object (u_sort_dw in this example).
2. Create an instance variable (also called a reference variable) in the owner (a DataWindow control in this example):

   ```
   u_sort_dw iuo_sort
   ```

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

   ```
   iuo_sort = CREATE u_sort_dw
   ```

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

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

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

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

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

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

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

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

Using user objects as structures

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

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

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

1 Create the user object, defining instance variables only.

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

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

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

Subclassing DataStores

Many applications use a DataWindow visual user object instead of the standard DataWindow window control. This allows you to standardize error checking and other, application-specific DataWindow behavior.

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

CHAPTER 2

Using Drag and Drop in a Window

About this chapter

This chapter describes how to make applications graphical by dragging and dropping controls.

Contents

About drag and drop

Drag and drop allows users to initiate activities by dragging a control and dropping it on another control. It provides a simple way to make applications graphical and easy to use. For example, in a manufacturing application you might allow the user to pick parts from a bin for an assembly by dragging a picture of the part and dropping it in the picture of the finished assembly.

Drag and drop involves at least two controls: the control that is being dragged (the **drag control**) and the control to which it is being dragged (the **target**). In PocketBuilder, all controls except drawing objects (lines, ovals, rectangles, and rounded rectangles) can be dragged.

Platform notes
Actions that require an application user to drag a control should be avoided since these actions are not very practical for users of handheld devices. Although you can script calls to drag events on Smartphone platforms, controls cannot be moved with a mouse or stylus, and the user has no direct way of dragging a control.

Automatic drag mode

When a control is being dragged, it is in drag mode. You can define a control so that PocketBuilder 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 events

Window objects and all controls except drawing objects have events that occur when they are the drag target. When a dragged control is within the target or dropped on the target, these events can trigger scripts. The scripts determine the activity that is performed when the drag control enters, is within, leaves, or is dropped on the target.

Drag-and-drop properties, events, and functions

Drag-and-drop properties

Each PocketBuilder control has a boolean DragAuto property.

Table 2-1: DragAuto property values

Value	Meaning
true	When the object is tapped, the control is placed automatically in drag mode.
false	When the object is tapped, the control is not placed automatically in drag mode; you have to put the object in drag mode manually by using the Drag function in a script.

❖ **To specify automatic drag mode for a control in the Window painter:**

1 Select the Other property page in the Properties view for the control.

2 Check the Drag Auto check box.

Drag-and-drop events

The drag-and-drop events listed in Table 2-2 are supported in PocketBuilder.

Table 2-2: Drag-and-drop events

Event	Occurs
BeginDrag	When the user taps in a ListView or TreeView control and begins dragging
DragDrop	When the pointer is over a target (a PocketBuilder control or window to which you drag a control) and the user stops dragging
DragEnter	When the pointer enters the boundaries of a target
DragLeave	When the pointer leaves the boundaries of a target
DragWithin	When the pointer moves within the boundaries of a target

Identifying the dragged control

To identify the type of control that was dropped, use the source argument of the DragDrop event.

This script for the DragDrop event in a picture declares two variables, then determines the type of object that was dropped:

```
CommandButton lcb_button
StaticText lst_info

IF source.TypeOf() = CommandButton! THEN
       lcb_button = source
       lcb_button.Text = "You dropped a Button!"
ELSEIF source.TypeOf() = StaticText! THEN
       lst_info = source
       lst_info.Text = "You dropped the text!"
END IF
```

Using CHOOSE CASE
If your window has a large number of controls that can be dropped, use a CHOOSE CASE statement.

CHAPTER 3

Using the PowerScript Language

About this chapter

This chapter describes how to use elements of the PowerScript language in an application. For more complete information, see the *PowerScript Reference*.

Contents

Dot notation

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

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

Qualifying a reference

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

objectvariable.item

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

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

parent.objectvariable.item

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

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

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

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

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

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

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

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

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

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

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

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

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

Referencing objects

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

- Unqualified:

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

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

- Qualified with the object's name:

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

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

- Qualified with a generic reference to the object:

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

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

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

This.*property*

This.*function*

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

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

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

This example in a script for a DataWindow control sets an instance variable of type DataWindow so that other functions can use it to access the most recently used DataWindow control:

```
idw_currentdw = This
```

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

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

```
// Two ways to call the window function
w_main.Resize(240, 320)
Parent.Resize(240, 320)

// Three ways to call the control's function
Resize(200, 200)
dw_data.Resize(200, 200)
This.Resize(200, 200)
```

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

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

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

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

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

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

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

Objects in a container object

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

control.Object.*objectname*.*property*

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

You can access DataWindow objects in DataWindow controls using the Object property.

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

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

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

For more information For more information about runtime checking, see "Optimizing expressions for DataWindow and external objects" on page 29.

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

Constant declarations

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

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

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

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

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

Advantages of constants

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

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

Controlling access for instance variables

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

You can specify that a variable is:

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

For example:

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

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

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

```
public privatewrite ii_averagerpm
```

Private variables for encapsulation

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

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

For more information

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

For more about encapsulation, see Chapter 1, "Implementing Object-Oriented Programming Techniques."

Resolving naming conflicts

There are two areas in which name conflicts occur:

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

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

Hidden instance variables

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

objectname.instancevariable

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

```
w_main.birthdate
```

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

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

Hidden global variables

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

::*globalvariable*

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

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

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

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

Overridden functions and events

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

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

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

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

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

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

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

Overloaded functions

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

Return values from ancestor scripts

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

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

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

Extending event scripts

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

CALL SUPER::*event_name*

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

Overriding event scripts

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

CALL SUPER::*event_name*

or

CALL *ancestor_name*::*event_name*

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

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

ancestor_name::EVENT *event_name* ( )

Example

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

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

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

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

Types of arguments for functions and events

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

There are three ways to pass an argument:

- **By value** Is the default

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

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

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

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

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

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

Matching argument types when overriding functions

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

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

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

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

Ancestor and descendent variables

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

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

As specific as possible

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

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

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

When the application requires flexibility

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

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

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

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

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

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

This technique is described in more detail in "Using dynamic versus static lookup" on page 9.

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

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

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

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

Dynamic object selection for windows and visual user objects

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

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

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

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

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

Optimizing expressions for DataWindow and external objects

No compiler validation for container objects

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

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

Everything you specify after the Object property passes the compiler and is checked during execution. Because of runtime syntax checking, using many expressions like these can impact performance.

Establishing partial references

To improve efficiency when you refer repeatedly to the same DataWindow component object or external object, you can define a variable of the appropriate type and assign a partial reference to the variable. The script evaluates most of the reference only once and reuses it.

The datatype of a DataWindow component object is DWObject:

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

Handling errors

The Error event is triggered when errors occur in evaluating DataWindow expressions. If you write a script for this event, you can catch an error before it triggers the SystemError event. This event lets you ignore an error or substitute an appropriate value. However, you must be careful to avoid setting up conditions that cause another error. You can also use try-catch blocks to handle exceptions, as described in "Exception handling in PocketBuilder" next.

For information

For information about DataWindow data expressions and property expressions and DWObject variables, see the *DataWindow Reference* in the online Help.

Printing at runtime

To print from a Pocket PC device or emulator, you must install the FieldSoftware PrinterCE SDK, available from the FieldSoftware Web site at http://www.fieldsoftware.com. After you install this software to the target platform, you can use DataWindow and PowerScript methods to print DataWindow or DataStore objects, visual objects, or lines of text from runtime applications.

The graphs that you print from a Pocket PC device or emulator do not expand to fill the print paper (as they do when you print from the desktop). The size of a graph that you print from these platforms is not modified from its screen display size.

Using a registered copy of the FieldSoftware PrinterCE SDK

You must use the SetRegistrationCode system function to supply a registration code authorizing the use of the FieldSoftware printing software. If you do not call this function, PocketBuilder assumes you are using an evaluation copy of the FieldSoftware PrinterCE SDK and attempts to make subsequent print function calls using the evaluation software.

For more information, see "SetRegistrationCode" in the online Help.

Sending mail from a device or emulator

You can send mail from a PocketBuilder application through a Microsoft ActiveSync connection that is configured to synchronize mail files with a desktop mail client. Microsoft Outlook can be configured to work with ActiveSync. ActiveSync does not support synchronization with Microsoft Outlook Express.

Configuring ActiveSync for mail synchronization with Microsoft Outlook

On your desktop machine, you configure ActiveSync to work with Outlook by selecting Tools>Options from the ActiveSync menu bar. In the list box on the Sync Options page of the Options dialog box, select Inbox for synchronization. If you select the check box to enable synchronization with a server, you cannot synchronize mail directly with Outlook.

To send mail in a PocketBuilder mail application, you must instantiate a mailSession object and call the PowerScript mail functions mailLogon, mailSend, and mailLogoff. You can use the mail Message and mailRecipient system structure objects to hold content and destination information for the messages you want to send by e-mail. The mailFileDescription object can be used to describe mail attachments, although mailAttach! is the only supported value for an attachment's file type.

Mail that you send from a PocketBuilder application is immediately placed in the Outbox of the Microsoft Outlook account that is synchronized to your Pocket PC through ActiveSync. The mail remains in the Outbox until a preset time for mail transfer has elapsed, or until an action on the desktop triggers the sending of mail in the Outbox.

Exception handling in PocketBuilder

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

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

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

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

Basics of exception handling

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

User-defined exceptions

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

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

Objects for exception handling support

Several system objects support exception handling within PocketBuilder.

Throwable object type

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

RuntimeError and its descendants

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

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

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

Exception object type

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

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

Handling exceptions

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

TRY-CATCH-FINALLY block

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

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

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

Example

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

```
Double ld_num
ld_num = Double (sle_1.text)
TRY
   sle_2.text = string (acos (ld_num))
CATCH (runtimeerror er)
   MessageBox("Runtime Error", er.GetMessage())
FINALLY
```

```
// Add cleanup code here
   of_cleanup()
   Return
END TRY
MessageBox("After", "We have finished.")
```

The system runtime error message might be confusing to the end user, so for production purposes, it would be better to catch a user-defined exception—see the example in "Creating user-defined exception types" on page 35—and set the message to something more understandable.

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

CATCH order

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

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

FINALLY clause

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

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

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

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

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

Creating user-defined exception types

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

Inherit from Exception object type

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

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

Throwing exceptions

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

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

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

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

Declaring exceptions thrown from functions

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

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

Example

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

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

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

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

Adding flexibility and facilitating object reuse

You can use exception handling to add flexibility to your PocketBuilder applications and to help separate business rules from presentation logic. For example, business rules can be stored in a non-visual object that has:

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

  ```
  powerobject my_presenter
  ```

- A function that registers the presentation object

  The registration function could use the following syntax:

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

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

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

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

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

  ```
  uo_exception luo_exception

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

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

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

Using the SystemError and Error events

Error event

If a runtime error occurs, an error structure that describes the error is created. An error can occur when an expression that uses dot notation to refer to data and properties of a DataWindow object is invalid under some runtime conditions. The DataWindow error Event is triggered, with the information in the error structure as arguments.

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

SystemError event

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

Precedence of exception handlers and events

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

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

Error handling for new applications

In PocketBuilder applications, you can handle errors by using try-catch blocks or by coding the Error event. You should always have a SystemError event coded in your Application object to handle any uncaught exceptions. The SystemError event essentially becomes a global exception handler for a PocketBuilder application.

Garbage collection

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

When garbage collection occurs

Garbage collection occurs:

- **When a reference is removed from an object** A reference to an object is any variable whose value is the object. When the variable goes out of scope, or when it is assigned a different value, PocketBuilder removes a reference to the object, counts the remaining references, and destroys the object if no references remain.

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

- **When the garbage collection interval is exceeded** When PocketBuilder completes the execution of a system-triggered event, it makes a garbage collection pass if the set interval between garbage collection passes has been exceeded. The default interval is 0.5 seconds. The garbage collection pass removes any objects and classes that cannot be referenced, including those containing circular references (otherwise unreferenced objects that reference each other).

Exceptions to garbage collection

There are a few objects that are not collected:

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

Controlling when garbage collection occurs

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

For information about these functions, see the online Help.

Efficient compiling and performance

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

Short scripts for faster compiling

Long scripts take a long time to compile. Break scripts up so that instead of one long script, you have a shorter script that makes calls to several other functions. Consider defining functions in user objects so that other objects can call the same functions.

Local variables for faster performance

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

CHAPTER 4

Getting Information About PocketBuilder Class Definitions

About this chapter

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

Contents

Overview of class definition information

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

Desktop only
ClassDefinition, ScriptDefinition, and other objects that descend from the ClassDefinitionObject object can be used in the development environment, but not in applications deployed to Windows CE devices or emulators.

From the ClassDefinition object, you can discover:

- The variables, functions, and events defined for the class
- The class's ancestor

- The class's parent
- The class's children (nested classes)

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

For more information, see the Browser or the online Help.

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

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

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

Terminology

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

object instance

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

class

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

system class

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

parent

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

Table 4-1: Classes and parents

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

child

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

ancestor

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

descendant

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

inheritance hierarchy

An object and all its ancestors.

collapsed hierarchy	A view of an object class definition that includes information from all the ancestors in the object's inheritance tree, not just items defined at the current level of inheritance.
scalar	A simple datatype that is not an object or an array. For example, Integer, Boolean, Date, Any, and String.
instance variable and property	Built-in properties of PocketBuilder system objects are called properties, but they are treated as instance variables in the class definition information.

Who uses PocketBuilder class definitions

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

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

Scenarios Class information might be used when developing:

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

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

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

Examining a class definition

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

Getting a class definition object

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

For an instantiated object in your application

Use its ClassDefinition property.

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

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

For an object stored in a PKL

Call FindClassDefinition.

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

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

Getting detailed information about the class

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

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

Library

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

```
s = cd_windef.LibraryName
```

Ancestor

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

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

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

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

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

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

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

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

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

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

mle_1.Text = s
```

The list might look like this:

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

Parent

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

```
ClassDefinition cd_parentwindef
```

```
cd_parentwindef = cd_windef.ParentClass
```

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

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

Nested or child classes

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

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

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

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

s = s + "Nested classes:" + lineend

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

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

```
integer li
ClassDefinition nested_cd

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

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

Getting information about a class's scripts

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

For examples of assigning a value to cd_windef, see "Getting a class definition object" on page 45.

List of scripts

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

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

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

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

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

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

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

   CHOOSE CASE sd.Kind
   CASE ScriptEvent!
      // Events have three relevant properties
      // regarding where code is defined
      s = s + "Event, "
      IF sd.IsScripted = TRUE then
         s = s + "scripted, "
      END If
      IF sd.IsLocallyScripted = TRUE THEN
         s = s + "local, "
      END IF
      IF sd.IsLocallyDefined = TRUE THEN
         s = s + "local def,"
      END IF

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

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

   li_bound = UpperBound(sd.ArgumentList)
   IF li_bound = 0 THEN s = s + "None"
   FOR lis = 1 to li_bound
   CHOOSE CASE sd.ArgumentList[lis]. &
      CallingConvention
      CASE ByReferenceArgument!
      s = s + "REF "
      CASE ByValueArgument!
      s = s + "VAL "
      CASE ReadOnlyArgument!
      s = s + "READONLY "
      CASE ELSE
      s = s + "BUILTIN "
   END CHOOSE
```

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

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

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

This example looks at the scripts for the class associated with the ClassDefinition cd_windef, and if a script's code is defined at this level, the script's name is added to a drop-down list. It also saves the script's position in the ScriptList array in the instance variable *ii_localscript_idx*.

The DropDownListBox is not sorted, so the positions in the list and the array stay in sync:

```
integer li_pos, li

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

Matching function signatures

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

For an example, see FindMatchingFunction in the online Help.

Getting information about variables

This section has code fragments illustrating how to get information about variables from a ClassDefinition object called cd_windef. For examples of assigning a value to cd_windef, see "Getting a class definition object" on page 45.

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

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

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

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

Details about variables

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

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

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

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

   CHOOSE CASE vard.Kind
   CASE VariableGlobal!
      s = s + ", global"
   CASE VariableShared!
      s = s + ", shared"
```

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

PART 2

Implementing User Interface Features

This part describes how to implement user interface features in the applications you develop with PocketBuilder.

CHAPTER 5

Using Tab Controls in a Window

About this chapter

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

Contents

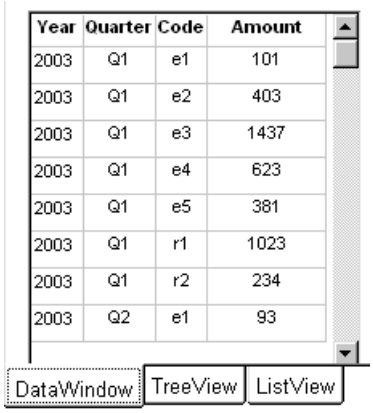
About Tab controls

Windows CE platforms
On the Pocket PC, you can use a tab page approach to application design as a substitute for MDI windows, which are not supported on Windows CE platforms. Tab controls are not supported on Smartphone devices or emulators.

A Tab control is a container for tab pages that display other controls. One page at a time fills the display area of the Tab control.

Each page has a tab like an index card divider. The user can click the tab to switch among the pages:

Figure 5-1: Tab control



The Tab control allows you to present many pieces of information in an organized way. This is particularly useful when building an application for a handheld device with a limited display area.

When sizing the tab control, remember that the height and width properties of the control refer to the size of the control *without* the tabs. Therefore, if you are sizing a tab control with tabs at the bottom to fill the window, set the height to about 1024 PBUs if your application uses the default window height of 1280 PBUs.

You add, resize, and move Tab controls just as you do any control. The *User's Guide* describes how to add controls to a window or custom visual user object.

Tab terms

You need to know these definitions:

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

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

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

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

Defining and managing tab pages

A tab page is a user object.

Two methods

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

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

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

Creating tab pages

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

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

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

2 Select Insert TabPage from the pop-up menu.

3 Add controls to the new page.

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

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

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

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

4 In the user object's Properties view, click the TabPage tab and fill in information to be used by the tab page.

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

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

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

3 Select a user object.

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

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

Managing tab pages

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

❖ **To view a different tab page:**

- Click the page's tab.

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

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

1 Click the Page Order tab on the Tab control's Properties view.

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

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

1 Click the page's tab.

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

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

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

Controls on tab pages

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

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

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

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

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

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

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

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

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

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

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

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

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

Customizing the Tab control

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

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

Pop-up menus and Properties views for Tab controls and tab pages

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

Where you click determines what element you access.

Table 5-1: Accessing Tab control elements

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

Position and size of tabs

The General tab in the Tab control's Properties view has several settings for controlling the position and size of the tabs. For example:

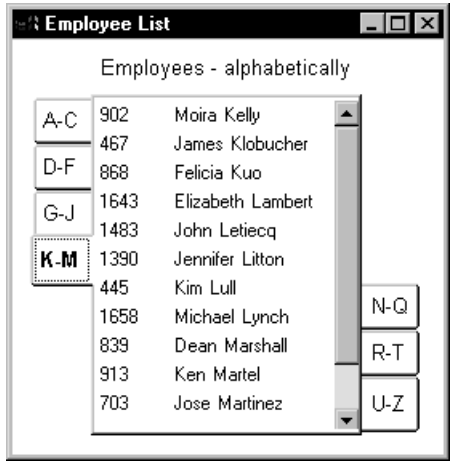
Table 5-2: Controlling size and position of tabs

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

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

The sample Tab control in Figure 5-2 is set up like an address book. It has tabs that flip between the left and right sides. With the Bold Selected Text setting on and the changing tab positions, it is easy to see which tab is selected.

Figure 5-2: Address book tab control



Tab labels

You can change the appearance of the tab using the Properties views of both the Tab control and the Tab page.

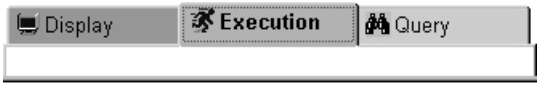
Table 5-3: Changing the appearance of a tab

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

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

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

Figure 5-3: Tabs with pictures and text



Changing tab appearance in scripts

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

Using Tab controls in scripts

This section provides examples of tabs in scripts:

- Referring to tab pages in scripts
- Referring to controls on tab pages
- Opening, closing, and hiding tab pages

- Keeping track of tab pages
- Events for the parts of the Tab control

Referring to tab pages in scripts

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

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

  window.tabcontrol

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

  window.tabcontrol.tabpageuo

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

  window.tabcontrol.tabpageuo.controlonpage

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

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

This example sets the PowerTipText property of tab page tabpage_1:

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

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

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

Generic coding

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

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

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

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

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

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

```
tab itab_settings
```

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

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

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

```
tab tab_mytab
userobject tabpage_generic

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

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

tab_mytab.SelectTab(tabpage_generic)
```

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

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

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

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

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

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

The Tab control's Control property

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

Referring to controls on tab pages

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

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

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

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

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

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

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

```
        ib_Retrieved = TRUE
END IF

RETURN dw_dir.RowCount()
```

Opening, closing, and hiding tab pages

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

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

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

```
userobject i_tabpage
```

This code opens the tab page:

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

This statement closes the tab page:

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

Keeping track of tab pages

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

Control property for tab pages

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

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

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

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

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

Adding a new tab page

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

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

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

Closing a tab page

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

Moving a tab page

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

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

Events for the parts of the Tab control

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

Table 5-4: Coding scripts for Tab control events

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

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

Examples

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

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

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

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

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

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

```
long ll_row


string ls_name
tabpage_listbox luo_tabpage

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

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

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

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

END IF
```

CHAPTER 6

Using Lists and Tree Views in a Window

About this chapter

This chapter describes how to use lists to present information in an application.

Contents

About presenting lists

You can choose a variety of ways to present lists in your application:

- ListBoxes display available choices that can be used for invoking an action or viewing and displaying data.
- DropDownListBoxes also display available choices to the user. However, you can make them editable to the user.
- ListView controls present lists in a combination of graphics and text. You can allow the user to add, delete, edit, and rearrange ListView items, or you can use them to invoke an action.
- TreeView controls also combine graphics and text in lists. The difference is that TreeView controls show the hierarchical relationship among the TreeView items. As with ListView controls, you can allow the user to add, delete, edit, and rearrange TreeView items. You can also use them to invoke actions.

Platform notes
Support is not available for pictures in list boxes on Windows CE platforms.

List boxes are automatically converted by PocketBuilder to spinner controls when deployed to Smartphone platforms, and extended or multiple selections for these controls are not supported. Arrow keys on a Smartphone allow the user to navigate within list view or tree view controls, but you must program a menu item to move the focus from one of these controls to a different control in the same main window.

For more information on spinner controls, see the appendix on designing applications for Windows CE platforms in the *User's Guide*.

Using ListBox controls

You can present information to the user in simple lists with scrollbars. Depending on how you design your application, the user can select one or more list items to perform an action, based on the list selection.

You add ListBox controls to windows in the same way you add other controls: select ListBox from the Insert>Control menu and click the window.

Adding items to list controls

In the painter To add new items, use the control's Items property page.

❖ **To add items to a ListBox:**

1 Select the Items tab in the Properties view for the control.

2 Enter the names of the items for the ListBox.

In a script Use the AddItem and InsertItem functions to dynamically add items to a ListBox at runtime. AddItem adds items to the end of the list. However, if the list is sorted, the item will then be moved to its position in the sort order. Use InsertItem if you want to specify where in the list the item will be inserted.

Table 6-1: Using the InsertItem and AddItem functions

Function	You supply
InsertItem	Item name Position in which the item will be inserted
AddItem	Item name

For example, this script adds items to a ListBox:

```
This.AddItem ("Vaporware")
This.InsertItem ("Software",2)
This.InsertItem ("Hardware",2)
This.InsertItem ("Paperware",2)
```

Using the Sort property
You can set the control's sort property to true or check the Sorted check box on the General property page to ensure that the items in the list are always arranged in ascending alphabetical order.

Using DropDownListBox controls

Drop-down lists are another way to present simple lists of information to the user. You add DropDownListBox controls to windows in the same way you add other controls: select DropDownListBox from the Insert>Control menu and click the window.

Adding items to drop-down list controls

In the painter Use the Items property page for the control to add items.

❖ **To add items to a DropDownListBox or DropDownPictureListBox:**

1 Select the Items tab in the Properties view for the control.

2 Enter the name of the items for the ListBox.

In a script Use the AddItem and InsertItem functions to dynamically add items to a DropDownListBox at runtime.

AddItem adds items to the end of the list. However, if the list is sorted, the item will then be moved to its position in the sort order. Use InsertItem if you want to specify where in the list the item will be inserted.

Table 6-2: Using the InsertItem and AddItem functions

Function	You supply
InsertItem	Item name Position in which the item will be inserted
AddItem	Item name

This example inserts three items into a DropDownListBox in the first, second, and third positions:

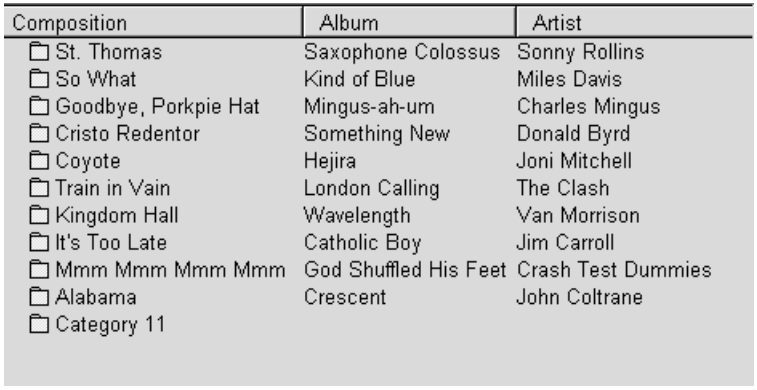
```
This.InsertItem ("Atropos", 1)
This.InsertItem ("Clotho", 2)
This.InsertItem ("Lachesis", 3)
```

Using the Sort property
You can set the control's sort property to true to ensure that the items in the list are always arranged in ascending sort order.

Using ListView controls

A **ListView control** allows you to display items and icons in a variety of arrangements. You can display large icon or small icon freeform lists, or you can display a vertical static list. You can also display additional information about each list item by associating additional columns with each list item:

Figure 6-1: ListView control with additional columns



ListView controls consist of **ListView items**, which are stored in an array. Each ListView item consists of a:

- **Label** The name of the ListView item
- **Index** The position of the ListView item in the control
- **Picture index** The number that associates the ListView item with an image

  Depending on the style of the presentation, an item can be associated with a large picture index and a small picture index.
- **Overlay picture index** The number that associates the ListView item with an overlay picture
- **State picture index** The number that associates the ListView item with a state picture

For more information about ListView items, picture indexes, and presentation style, see the *User's Guide*.

You add ListView controls to windows in the same way you add other controls: select ListView from the Insert>Control menu and click the window.

Adding ListView items

In the painter Use the Items property page for the control to add items.

❖ **To add items to a ListView:**

1 Select the Items tab in the Properties view for the control.

2 Enter a name and a picture index number for each of the items you want to add to the ListView.

Note Setting the picture index for the first item to zero clears all the settings on the tab page.

For more information about adding pictures to a ListView control, see "Adding pictures to ListView controls" on page 74.

In a script Use the AddItem and InsertItem functions to add items to a ListView dynamically at runtime. There are two levels of information you supply when you add items to a ListView using AddItem or InsertItem.

You can add an item by supplying the picture index and label, as this example shows:

```
lv_1.AddItem ("Item 1", 1)
```

or you can insert an item by supplying the item's position in the ListView, label, and picture index:

```
lv_1.InsertItem (1,"Item 2", 2)
```

You can add items by supplying the ListView item itself. This example in the ListView's DragDrop event inserts the dragged object into the ListView:

```
listviewitem lvi
This.GetItem(index, lvi)
lvi.label = "Test"
lvi.pictureindex = 1
This.AddItem (lvi)
```

You can insert an item by supplying the ListView position and ListView item:

```
listviewitem l_lvi
//Obtain the information for the
//second listviewitem
lv_list.GetItem(2, l_lvi)
//Change the item label to Entropy
//Insert the second item into the fifth position
lv_list.InsertItem (5, l_lvi)
lv_list.DeleteItem(2)
```

Adding pictures to ListView controls

PocketBuilder stores ListView images in four image lists:

- Small picture index
- Large picture index
- State picture index
- Overlay picture index

You can associate a ListView item with these images when you create a ListView in the painter, or you can use the AddItem and InsertItem at runtime.

However, before you can associate pictures with ListView items, they must be added to the ListView control.

In the painter To add pictures, use the control's Pictures and Items property pages.

❖ **To add pictures to a ListView control:**

1 Select the Large Picture, Small Picture, or State tab in the Properties view for the control.

Overlay images
You can add overlay images only to a ListView control in a script.

---

2 Select an image from the stock image list, or use the Browse button to select a bitmap, cursor, or icon image.

3 Select a color from the PictureMaskColor drop-down menu for the image.

The color selected for the picture mask appears transparent in the ListView.

4 Select a picture height and width for your image.

This controls the size of the image in the ListView.

Dynamically changing image size
The image size can be changed at runtime by setting the PictureHeight and PictureWidth properties before you add any pictures when you create a ListView. For more information about PictureHeight and PictureWidth, see the online Help.

---

5 Repeat the procedure for the:

- Number of image types (large, small, and state) you plan to use in your ListView
- Number of images for each type

In a script Use the functions in Table 6-3 to add pictures to a ListView image.

Table 6-3: Functions that add pictures to a ListView image

Function	Adds a picture to this list
AddLargePicture	Large image
AddSmallPicture	Small image
AddStatePicture	State image

Adding large and small pictures This example sets the height and width for large and small pictures and adds three images to the large picture image list and the small picture image list:

```
//Set large picture height and width
```

```
lv_1.LargePictureHeight=32
lv_1.LargePictureWidth=32

//Add large pictures
lv_1.AddLargePicture("c:\ArtGal\bmps\celtic.bmp")
lv_1.AddLargePicture("c:\ArtGal\bmps\list.ico")
lv_1.AddLargePicture("Custom044!")

//Set small picture height and width
lv_1.SmallPictureHeight=16
lv_1.SmallPictureWidth=16

//Add small pictures
lv_1.AddSmallPicture("c:\ArtGal\bmps\celtic.bmp")
lv_1.AddSmallPicture("c:\ArtGal\bmps\list.ico")
lv_1.AddSmallPicture("Custom044!")

//Add items to the ListView
lv_1.AddItem("Item 1", 1)
lv_1.AddItem("Item 2", 1)
lv_1.AddItem("Item 3", 1)
```

Adding overlay pictures Use the SetOverLayPicture function to use a large picture or small picture as an overlay for an item. This example adds a large picture to a ListView, and then uses it for an overlay picture for a ListView item:

```
listviewitem lvi_1
int li_index

//Add a large picture to a ListView
li_index = lv_list.AddLargePicture &
   ("c:\ArtGal\bmps\dil2.ico")

//Set the overlay picture to the
//large picture just added
lv_list.SetOverlayPicture (1, li_index)

//Use the overlay picture with a ListViewItem
lv_list.GetItem(lv_list.SelectedIndex (), lvi_1)
lvi_1.OverlayPictureIndex = 1
lv_list.SetItem(lv_list.SelectedIndex (), lvi_1)
```

Adding state pictures This example uses an item's state picture index property to set the state picture for the selected ListView item:

```
listviewitem lvi_1
lv_list.GetItem(lv_list.SelectedIndex (), lvi_1)
```

```
lvi_1.StatePictureIndex = 2
lv_list.SetItem(lv_list.SelectedIndex (), lvi_1)
```

Deleting ListView items and pictures

You can delete items from a ListView one at a time with the DeleteItem function, or you can use the DeleteItems function to purge all the items in a ListView. Similarly, you can delete pictures one at a time with the DeleteLargePicture, DeleteSmallPicture, and DeleteStatePicture functions, or purge all pictures of a specific type by using the DeleteLargePictures, DeleteSmallPictures, and DeleteStatePictures functions.

This example deletes one item and all the small pictures from a ListView:

```
int li_index
li_index = This.SelectedIndex()
This.DeleteItem (li_index)
This.DeleteSmallPictures ()
```

Using report view

ListView report view requires more information than the large icon, small icon, and list view. To enable report view in a ListView control, you must write a script that establishes columns with the AddColumn and SetColumn functions, and then populate the columns using the SetItem function.

Populating columns

Use AddColumn to create columns in a ListView. When you use the AddColumn function, you specify the:

- **Column label** The name that will display in the column header
- **Column alignment** Whether the text will be left-aligned, right-aligned, or centered
- **Column size** The width of the column in PowerBuilder units

This example creates three columns in a ListView:

```
This.AddColumn("Name", Left!, 1000)
This.AddColumn("Size", Left!, 400)
This.AddColumn("Date", Left!, 300)
```

Setting columns

Use SetColumn to set the column number, name, alignment, and size:

```
This.SetColumn (1, "Composition", Left!, 860)
This.SetColumn (2, "Album", Left!, 610)
This.SetColumn (3, "Artist", Left!, 710")
```

Setting column items

Use SetItem to populate the columns of a ListView:

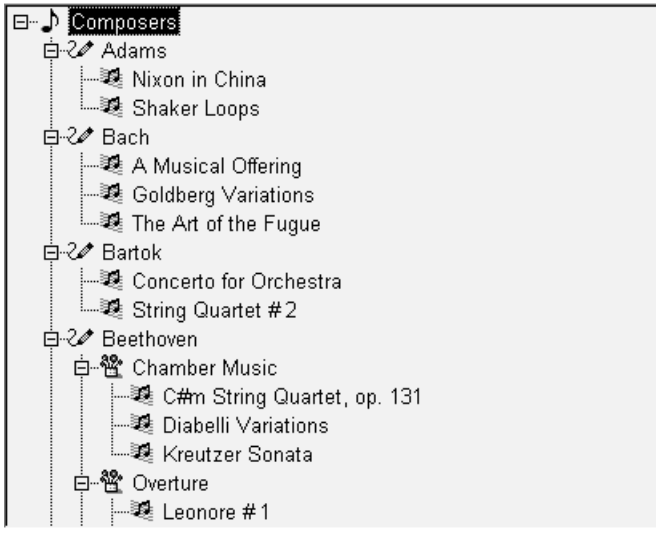
```
This.SetItem (1, 1, "St.Thomas")
```

```
This.SetItem (1, 2, "Saxophone Colossus")
This.SetItem (1, 3, "Sonny Rollins")
This.SetItem (2, 1, "So What")
This.SetItem (2, 2, "Kind of Blue")
This.SetItem (2, 3, "Miles Davis")
This.SetItem (3, 1, "Good-bye, Porkpie Hat")
This.SetItem (3, 2, "Mingus-ah-um")
This.SetItem (3, 3, "Charles Mingus")
```

Using TreeView controls

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

Figure 6-2: TreeView control with pictures



When to use a TreeView

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

Hierarchy of items

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

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

Number of levels in each branch

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

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

Content sources for a TreeView

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

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

For an example using DataStores, see "Using DataWindow information to populate a TreeView" on page 97.

Pictures for items

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

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

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

Appearance of the TreeView

You can control the appearance of the TreeView by setting property values. Properties that affect the overall appearance are shown in Table 6-4.

Table 6-4: TreeView properties

Properties	Effect when set
HasButtons	Puts + and - buttons before items that have children, showing the user whether the item is expanded or collapsed (use with HasLines)
HasLines and LinesAtRoot	Displays lines connecting items within a branch and connecting items at the root level
SingleExpand	Expands the selected item and collapses the previously selected item automatically
Indent	Sets the amount an item is indented
Font properties	Specifies the font for all the labels
Various picture properties	Controls the pictures and their size

For more information about these properties, see the online Help.

User interaction

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

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

Populating TreeViews

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

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

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

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

Functions for inserting items

There are several functions for adding items to a TreeView control, as shown in Table 6-5.

Table 6-5: Functions for adding items to TreeView control

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

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

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

Method 1: specifying the label and picture index only

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

Example This example inserts a new item after the currently selected item on the same level as that item. First it gets the handles of the currently selected item and its parent, and then it inserts an item labeled Hindemith after the currently selected item. The item's picture index is 2:

```
long ll_tvi, ll_tvparent
ll_tvi = tv_list.FindItem(CurrentTreeItem!, 0)
ll_tvparent = tv_list.FindItem(ParentTreeItem!, &
   ll_tvi)
tv_list.InsertItem(ll_tvparent, ll_tvi, &
   "Hindemith", 2)
```

Method 2: setting item properties in a TreeViewItem structure

You can add items by supplying a TreeViewItem structure with properties set to specific values. The only required property is a label. Properties you might set are shown in Table 6-6.

Table 6-6: TreeViewItem properties

Property	Value
Label	The text that is displayed for the item.
PictureIndex	A value from the regular picture list.
SelectedPictureIndex	A value from the regular picture list, specifying a picture that is displayed only when the item is selected. If 0, no picture is displayed for the item when selected.
StatePictureIndex	A value from the State picture list. The picture is displayed to the left of the regular picture.
Children	Must be true if you want double-clicking to trigger the ItemPopulate event. That event script can insert child items.
Data	An optional value of any datatype that you want to associate with the item. You might use the value to control sorting or to make a database query.

Example This example sets all these properties in a TreeViewItem structure before adding the item to the TreeView control. The item is inserted as a child of the current item:

```
treeviewitem tvi
long h_item = 0, h_parent = 0

h_parent = tv_1.FindItem(CurrentTreeItem!, 0)

tvi.Label = "Choral"
tvi.PictureIndex = 1
tvi.SelectedPictureIndex = 2
```

```
tvi.Children = true
tvi.StatePictureIndex = 0

h_item = tv_1.InsertItemSort(h_parent, tvi)
```

Inserting items at the root level

The very first item you insert does not have any sibling for specifying a relative position, so you cannot use the InsertItem function. You must use InsertItemFirst or InsertItemLast. For an item inserted at the root level, you specify 0 as its parent.

This sample code is in a user event triggered from the Open event of the window containing the TreeView. It assumes two instance variable arrays:

- A string array called item_label that contains labels for all the items that will be inserted at the root level (here, composer names)
- An integer array that has values for the Data property (the century for each composer); the century value is for user-defined sorting:

```
int ct
long h_item = 0
treeviewitem tvi

FOR ct = 1 TO UpperBound(item_label)
      tvi.Label = item_label[ct]
      tvi.Data = item_data[ct]
      tvi.PictureIndex = 1
      tvi.SelectedPictureIndex = 2
      tvi.Children = TRUE
      tvi.StatePictureIndex = 0
      tvi.DropHighlighted = TRUE

      h_item = tv_1.InsertItemSort(0, tvi)
NEXT
```

After inserting all the items, this code scrolls the TreeView back to the top and makes the first item current:

```
// Scroll back to top
h_item = tv_1.FindItem(RootTreeItem!, 0)
tv_1.SetFirstVisible(h_item)
tv_1.SelectItem(h_item)
```

Inserting items below the root level

The first time a user tries to expand an item to see its children, PocketBuilder triggers the ItemPopulate event *if and only if* the value of the item's Children property is true. In the ItemPopulate event, you can add child items for the item being expanded.

Parent item's Children property
If the ItemPopulate event does not occur when you expect, make sure the Children property for the expanding item is true. It should be set to true for any item that will have children.

Inserting items not restricted to the ItemPopulate event The ItemPopulate event helps you design an efficient program. It will not populate an item that the user never looks at. However, you do not have to wait until the user wants to view an item's children. You can add children in any script, just as you added items at the root level.

For example, you might fully populate a small TreeView when its window opens and use the ExpandAll function to display its items fully expanded.

Has an item been populated? You can check an item's ExpandedOnce property to find out if the user has looked at the item's children. If the user is currently looking at an item's children, the Expanded property is also true.

Example This TreeView lists composers and their music organized into categories. The script for its ItemPopulate event checks whether the item being expanded is at level 1 (a composer) or level 2 (a category). Level 3 items are not expandable.

For a level 1 item, the script adds three standard categories. For a level 2 item, it adds pieces of music to the category being expanded, in this pattern:

```
Mozart
   Orchestral
   Symphony No. 33
   Overture to the Magic Flute
   Chamber
   Quintet in Eb for Horn and Strings
   Eine Kleine Nachtmusik
   Vocal
   Don Giovanni
   Idomeneo
```

This is the script for ItemPopulate:

```
TreeViewItem tvi_current, tvi_child, tvi_root
```

```
long hdl_root
Integer ct
string categ[]

// The current item is the parent for the new items
This.GetItem(handle, tvi_current)

IF tvi_current.Level = 1 THEN
   // Populate level 2 with some standard categories
   categ[1] = "Orchestral"
   categ[2] = "Chamber"
   categ[3] = "Vocal"

   tvi_child.StatePictureIndex = 0
   tvi_child.PictureIndex = 3
   tvi_child.SelectedPictureIndex = 4
   tvi_child.OverlayPictureIndex = 0
   tvi_child.Children = TRUE

   FOR ct = 1 to UpperBound(categ)
      tvi_child.Label = categ[ct]
      This.InsertItemLast(handle, tvi_child)
   NEXT
END IF

// Populate level 3 with music titles
IF tvi_current.Level = 2 THEN
   // Get parent of current item - it's the root of
   // this branch and is part of the key for choosing
   // the children

   hdl_root = This.FindItem(ParentTreeItem!, handle)
   This.GetItem(hdl_root, tvi_root)

   FOR ct = 1 to 4
      // This statement constructs a label -
      // it is more realistic to look up data in a
      // table or database or accept user input
      This.InsertItemLast(handle, &
      tvi_root.Label + " Music " &
      + tvi_current.Label + String(ct), 3)
   NEXT
END IF
```

Managing TreeView items

An item in a TreeView is a TreeViewItem structure. The preceding section described how to set the item's properties in the structure and then insert it into the TreeView.

This code declares a TreeViewItem structure and sets several properties:

```
TreeViewItem tvi_defined
tvi_defined.Label = "Symphony No. 3 Eroica"
tvi_defined.StatePictureIndex = 0
tvi_defined.PictureIndex = 3
tvi_defined.SelectedPictureIndex = 4
tvi_defined.OverlayPictureIndex = 0
tvi_defined.Children = TRUE
```

For information about Picture properties, see "Managing TreeView pictures" on page 94.

When you insert an item, the inserting function returns a handle to that item. The TreeViewItem structure is copied to the TreeView control, and you no longer have access to the item's properties:

```
itemhandle = This.InsertItemLast(parenthandle, &
   tvi_defined)
```

Procedure for items: get, change, and set

If you want to change the properties of an item in the TreeView, you:

1 *Get* the item, which assigns it to a TreeViewItem structure.

2 Make the *changes*, by setting TreeViewItem properties.

3 *Set* the item, which copies it back into the TreeView.

When you work with items that have been inserted in the TreeView, you work with item handles. Most TreeView events pass one or two handles as arguments. The handles identify the items the user is interacting with.

This code for the Clicked event uses the handle of the clicked item to copy it into a TreeViewItem structure whose property values you can change:

```
treeviewitem tvi
This.GetItem(handle, tvi)
tvi.OverlayPictureIndex = 1
This.SetItem(handle, tvi)
```

Important
Remember to call the SetItem function after you change an item's property value. Otherwise, nothing happens in the TreeView.

Items and the hierarchy

You can use item handles with the FindItem function to navigate the TreeView and uncover its structure. The item's properties tell you what its level is, but not which item is its parent. The FindItem function does:

```
long h_parent
h_parent = This.FindItem(ParentTreeItem!, handle)
```

You can use FindItem to find the children of an item or to navigate through visible items regardless of level.

For more information, see the FindItem function in the online Help.

Enabling TreeView functionality in scripts

By setting TreeView properties, you can enable or disable user actions like deleting or renaming items without writing any scripts. You can also enable these actions by calling functions. You can:

- Delete items
- Rename items
- Move items using drag and drop
- Sort items

Deleting items

To allow the user to delete items, enable the TreeView's DeleteItems property. When the user presses the Delete key, the selected item is deleted and the DeleteItem event is triggered. Any children are deleted too.

If you want more control over deleting, such as allowing deleting of detail items only, you can call the DeleteItem function instead of setting the property. The function also triggers the DeleteItem event.

Example

This script is for a TreeView user event. Its event ID is pbm_keydown and it is triggered by key presses when the TreeView has focus. The script checks whether the Delete key is pressed and whether the selected item is at the detail level. If both are true, it deletes the item.

The value of the TreeView's DeleteItems property is false. Otherwise, the user can delete any item, despite this code:

```
TreeViewItem tvi
long h_item

IF KeyDown(KeyDelete!) = TRUE THEN
   h_item = This.FindItem(CurrentTreeItem!, 0)
   This.GetItem(h_item, tvi)
```

```
   IF tvi.Level = 3 THEN
      This.DeleteItem(h_item)
   END IF
END IF
RETURN 0
```

Renaming items

If you enable the TreeView's EditLabels property, the user can edit an item label by clicking twice on the text.

Events

There are two events associated with editing labels.

The BeginLabelEdit event occurs after the second click when the EditLabels property is set or when the EditLabel function is called. You can disallow editing with a return value of 1.

This script for BeginLabelEdit prevents changes to labels of level 2 items:

```
TreeViewItem tvi
This.GetItem(handle, tvi)
IF tvi.Level = 2 THEN
   RETURN 1
ELSE
   RETURN 0
END IF
```

The EndLabelEdit event occurs when the user finishes editing by pressing ENTER, clicking on another item, or clicking in the text entry area of another control. A script you write for the EndLabelEdit event might validate the user's changes—for example, it could invoke a spelling checker.

EditLabel function

For control over label editing, the BeginLabelEdit event can prohibit editing of a label, as shown above. Alternatively, you can set the EditLabels property to false and call the EditLabel function when you want to allow a label to be edited.

When you call the EditLabel function, the BeginLabelEdit event occurs when editing begins and the EndLabelEdit event occurs when the user presses enter or clicks another item.

This code for a CommandButton puts the current item into editing mode:

```
long h_tvi
h_tvi = tv_1.findItem(CurrentTreeItem!, 0)
tv_1.EditLabel(h_tvi)
```

Moving items using drag and drop

At the window level, PocketBuilder provides functions and properties for dragging controls onto other controls. Within the TreeView, you can also let the user drag items onto other items. Users might drag items to sort them, move them to another branch, or put child items under a parent.

Platform notes
Actions that require an application user to drag a control should be avoided since these actions are not very practical for users of handheld devices. Although you can script calls to drag events on Smartphone platforms, controls cannot be moved with a mouse or stylus, and the user has no direct way of dragging a control.

When you implement drag and drop as a way to move items, you decide whether the dragged item becomes a sibling or child of the target, whether the dragged item is moved or copied, and whether its children get moved with it.

There are several properties and events that you need to coordinate to implement drag and drop for items, as shown in Table 6-7.

Table 6-7: Drag-and-drop properties and events

Property or event	Setting or purpose
DragAuto property	True or false. If false, you must call the Drag function in the BeginDrag event.
DisableDragDrop property	False.
DragIcon property	An appropriate icon, or None!, which means the user drags an image of the item.
BeginDrag event	Script for saving the handle of the dragged item and optionally preventing particular items from being dragged.
DragWithin event	Script for highlighting drop targets.
DragDrop event	Script for implementing the result of the drag operation.

Example

The key to a successful drag-and-drop implementation is in the details. This section illustrates one way of moving items. In the example, the dragged item becomes a sibling of the drop target, inserted after it. All children of the item are moved with it, and the original item is deleted.

A function called recursively moves the children, regardless of the number of levels. To prevent an endless loop, an item cannot become a child of itself. This means a drop target that is a child of the dragged item is not allowed.

BeginDrag event The script saves the handle of the dragged item in an instance variable:

```
ll_dragged_tvi_handle = handle
```

If you want to prevent some items from being dragged—such as items at a particular level—that code goes here too:

```
TreeViewItem tvi
This.GetItem(handle, tvi)
IF tvi.Level = 3 THEN This.Drag(Cancel!)
```

DragWithin event The script highlights the item under the cursor so the user can see each potential drop target. If only some items are drop targets, your script should check an item's characteristics before highlighting it. In this example, you can check whether an item is a parent of the dragged item and highlight it only if it is not:

```
TreeViewItem tvi
This.GetItem(handle, tvi)
tvi.DropHighlighted = TRUE
This.SetItem(handle, tvi)
```

DragDrop event This script does all the work. It checks whether the item can be inserted at the selected location and inserts the dragged item in its new position—a sibling after the drop target. Then it calls a function that moves the children of the dragged item too:

```
TreeViewItem tvi_src, tvi_child
long h_parent, h_gparent, h_moved, h_child
integer rtn

// Get TreeViewItem for dragged item
This.GetItem(ll_dragged_tvi_handle, tvi_src)

// Don't allow moving an item into its own branch,
// that is, a child of itself
h_gparent = This.FindItem(ParentTreeItem!, handle)

DO WHILE h_gparent <> -1
   IF h_gparent = ll_dragged_tvi_handle THEN
      MessageBox("No Drag", &
      "Can't make an item a child of itself.")
      RETURN 0
   END IF
   h_gparent=This.FindItem(ParentTreeItem!, h_gparent)
LOOP

// Get item parent for inserting
```

```
h_parent = This.FindItem(ParentTreeItem!, handle)
// Use 0 if no parent because target is at level 1
IF h_parent = -1 THEN h_parent = 0

// Insert item after drop target
h_moved = This.InsertItem(h_parent, handle, tvi_src)
IF h_moved = -1 THEN
   MessageBox("No Dragging", "Could not move item.")
   RETURN 0

ELSE
   // Args: old parent, new parent
   rtn = uf_movechildren(ll_dragged_tvi_handle, &
      h_moved)

   / If all children are successfully moved,
   // delete original item


   IF rtn = 0 THEN
      This.DeleteItem(ll_dragged_tvi_handle)
   END IF
END IF
```

The DragDrop event script shown above calls the function uf_movechildren. The function calls itself recursively so that all the levels of children below the dragged item are moved:

```
// Function: uf_movechildren
// Arguments:
// oldparent - Handle of item whose children are
// being moved. Initially, the dragged item in its
// original position
//
// newparent - Handle of item to whom children are
// being moved. Initially, the dragged item in its
// new position.

long h_child, h_movedchild
TreeViewItem tvi

// Return -1 if any Insert action fails

// Are there any children?
h_child = tv_2.FindItem(ChildTreeItem!, oldparent)
IF h_child <> -1 THEN
   tv_2.GetItem(h_child, tvi)
```

```
      h_movedchild = tv_2.InsertItemLast(newparent, tvi)
      IF h_movedchild = -1 THEN RETURN -1

      // Move the children of the child that was found
      uf_movechildren(h_child, h_movedchild)

      // Check for more children at the original level
      h_child = tv_2.FindItem(NextTreeItem!, h_child)
      DO WHILE h_child <> -1
         tv_2.GetItem(h_child, tvi)
         h_movedchild= tv_2.InsertItemLast(newparent,tvi)
         IF h_movedchild = -1 THEN RETURN -1
         uf_movechildren(h_child, h_movedchild)

         // Any more children at original level?
         h_child = tv_2.FindItem(NextTreeItem!, h_child)
      LOOP
   END IF
   RETURN 0 // Success, all children moved
```

Sorting items

A TreeView can sort items automatically, or you can control sorting manually. Manual sorting can be alphabetic by label text, or you can implement a user-defined sort to define your own criteria. The SortType property controls the way items are sorted. Its values are of the enumerated datatype grSortType.

Automatic alphabetic sorting To enable sorting by the text label, set the SortType property to Ascending! or Descending!. Inserted items are sorted automatically.

Manual alphabetic sorting For more control over sorting, you can set SortType to Unsorted! and sort by calling the functions in Table 6-8.

Table 6-8: TreeView sorting functions

Use this function	To do this
InsertItemSort	Insert an item at the correct alphabetic position, if possible
Sort	Sort the immediate children of an item
SortAll	Sort the whole branch below an item

If users will drag items to organize the list, you should disable sorting.

Sorting by other criteria To sort items by criteria other than their labels, implement a user-defined sort by setting the SortType property to UserDefinedSort! and writing a script for the Sort event. The script specifies how to sort items.

PocketBuilder triggers the Sort event for each pair of items it tries to reorder. The Sort script returns a value reporting which item is greater than the other. The script can have different rules for sorting, based on the type of item. For example, level 2 items can be sorted differently from level 3. The TreeView is sorted whenever you insert an item.

Example of Sort event

This sample script for the Sort event sorts the first level by the value of the Data property and other levels alphabetically by their labels. The first level displays composers chronologically, and the Data property contains an integer identifying a composer's century:

```
//Return values
//-1   Handle1 is less than handle2
// 0   Handle1 is equal to handle2
// 1   Handle1 is greater than handle2

TreeViewItem tvi1, tvi2

This.GetItem(handle1, tvi1)
This.GetItem(handle2, tvi2)

IF tvi1.Level = 1 THEN
   // Compare century values stored in Data property
   IF tvi1.data > tvi2.Data THEN
      RETURN 1
   ELSEIF tvi1.data = tvi2.Data THEN
      RETURN 0
   ELSE
      RETURN -1
   END IF
   ELSE
   // Sort other levels in alpha order
   IF tvi1.Label > tvi2.Label THEN
      RETURN 1
   ELSEIF tvi1.Label = tvi2.Label THEN
      RETURN 0
   ELSE
      RETURN -1
   END IF
END IF
```

Managing TreeView pictures

PocketBuilder stores TreeView images in three image lists:

- Picture list (called the *regular picture list* here)
- State picture list
- Overlay picture list

You add pictures to these lists and associate them with items in the TreeView.

Pictures for items

There are several ways to use pictures in a TreeView. You associate a picture in one of the picture lists with an item by setting one of the item's picture properties, described in Table 6-9.

Table 6-9: TreeView picture properties

Property	Purpose
PictureIndex	The primary picture associated with the item is displayed just to the left of the item's label.
StatePictureIndex	A state picture is displayed to the left of the regular picture. The item moves to the right to make room for the state picture. If the Checkboxes property is true, the state picture is replaced by a pair of check boxes. Because a state picture takes up room, items without state pictures will not align with items that have pictures. So that all items have a state picture and stay aligned, you can use a blank state picture for items that do not have a state to be displayed. A use for state pictures might be to display a check mark beside items the user has chosen.

Property	Purpose
OverlayPictureIndex	An overlay picture is displayed on top of an item's regular picture. You set up the overlay picture list in a script by designating a picture in the regular picture list for the overlay picture list. An overlay picture is the same size as a regular picture, but it often uses a small portion of the image space so that it only partially covers the regular picture. A typical use of overlay pictures is the arrow marking shortcut items in the Windows Explorer.
SelectedPictureIndex	A picture from the regular picture list that is displayed in place of the regular picture when the item is the current item. When the user selects another item, the first item gets its regular picture and the new item displays its selected picture. If you do not want a different picture when an item is current, set SelectedPictureIndex to the same value as PictureIndex.

How to set pictures You can change the pictures for all items at a particular level with the SetLevelPictures function, or you can set the picture properties for an individual item.

If you do not want pictures Your TreeView does not have to use pictures for items. If an item's picture indexes are 0, no pictures are displayed. However, the TreeView always leaves room for the regular picture.

You can set the PictureWidth property to 0 to eliminate that space:

```
tv_2.DeletePictures()
tv_2.PictureWidth = 0
```

Setting up picture lists

You can add pictures to the regular and state picture lists in the painter or at runtime. At runtime, you can assign pictures in the regular picture list to the overlay list.

Mask color

The mask color is a color used in the picture that becomes transparent when the picture is displayed. Usually you should pick the picture's background color so that the picture blends with the color of the TreeView.

Before you add a picture, in the painter or in a script, you can set the mask color to a color appropriate for that picture. This statement sets the mask color to white, which is right for a picture with a white background:

```
tv_1.PictureMaskColor = RGB(255, 255, 255)
```

Each picture can have its own mask color. A picture uses the color that is in effect when the picture is inserted. To change a picture's mask color, you have to delete the picture and add it again.

Image size

In the painter you can change the image size at any time by setting the Height and Width properties on each picture tab. All the pictures in the list are scaled to the specified size.

At runtime, you can change the image size for a picture list only when that list is empty. The DeletePictures and DeleteStatePictures functions let you clear the lists.

Example

This sample code illustrates how to change properties and add pictures to the regular picture list at runtime. Use similar code for state pictures:

```
tv_list.DeletePictures()
tv_list.PictureHeight = 32
tv_list.PictureWidth = 32

tv_list.PictureMaskColor = RGB(255,255,255)
tv_list.AddPicture("c:\apps_pb\kelly.bmp")

tv_list.PictureMaskColor = RGB(255,0,0)
tv_list.AddPicture("Custom078!")
tv_list.PictureMaskColor = RGB(128,128,128)
tv_list.AddPicture("Custom044!")
```

How picture deletion affects existing items

Deleting pictures from the picture lists can have an unintended effect on item pictures being displayed. When you delete pictures, the remaining pictures in the list are shifted to remove gaps in the list. The remaining pictures get a different index value. This means items that use these indexes get new images.

Deleting pictures from the regular picture list also affects the overlay list, since the overlay list is not a separate list but points to the regular pictures.

To avoid unintentional changes to item pictures, it is best to avoid deleting pictures after you have begun using picture indexes for items.

Using overlay pictures

The pictures in the overlay list come from the regular picture list. First you must add pictures to the regular list, either in the painter or at runtime. Then, at runtime, you specify pictures for the overlay picture list. After that you can assign an overlay picture to items, either individually or with the SetLevelPictures function.

This code adds a picture to the regular picture list and then assigns it to the overlay list:

```
integer idx
idx = tv_1.AddPicture("Custom085!")
IF tv_1.SetOverlayPicture(1, idx) <> 1 THEN
   sle_get.Text = "Setting overlay picture failed"
END IF
```

This code for the Clicked event turns the overlay picture on or off each time the user clicks an item:

```
treeviewitem tvi
This.GetItem(handle, tvi)
IF tvi.OverlayPictureIndex = 0 THEN
   tvi.OverlayPictureIndex = 1
ELSE
   tvi.OverlayPictureIndex = 0
END IF
This.SetItem(handle, tvi)
```

Using DataWindow information to populate a TreeView

A useful implementation of the TreeView control is to populate it with information that you retrieve from a DataWindow. To do this, your application must:

- Declare and instantiate a DataStore and assign a DataWindow object
- Retrieve information as needed
- Use the retrieved information to populate the TreeView
- Destroy the DataStore instance when you have finished

Because a TreeView can display different types of information at different levels, you will probably define additional DataWindows, one for each level. Those DataWindows usually refer to different but related tables. When an item is expanded, the item becomes a retrieval argument for getting child items.

Populating the first level

This example populates a TreeView with a list of composers. The second level of the TreeView displays music by each composer. In the database there are two tables: composer names and music titles (with composer name as a foreign key).

This example declares two DataStore instance variables for the window containing the TreeView control:

```
datastore ids_data, ids_info
```

This example uses the TreeView control's Constructor event to:

- Instantiate the DataStore
- Associate it with a DataWindow and retrieve information
- Use the retrieved data to populate the root level of the TreeView:

```
//Constructor event for tv_1
treeviewitem tvi1, tvi2
long ll_lev1, ll_lev2, ll_rowcount, ll_row

//Create instance variable datastore
ids_data = CREATE datastore
ids_data.DataObject = "d_composers"
ids_data.SetTransObject(SQLCA)
ll_rowcount = ids_data.Retrieve()

//Create the first level of the TreeView
tvi1.PictureIndex = 1
tvi1.Children = TRUE


//Populate the TreeView with
//data retrieved from the datastore
FOR ll_row = 1 to ll_rowcount
      tvi1.Label = ids_data.GetItemString(ll_row, &
      'name')
      This.InsertItemLast(0, tvi1)
NEXT
```

Populating the second level

When the user expands a root level item, the ItemPopulate event occurs. This script for the event:

- Instantiates a second DataStore

  Its DataWindow uses the composer name as a retrieval argument for the music titles table.

- Inserts music titles as child items for the selected composer

The handle argument of ItemPopulate will be the parent of the new items:

```
//ItemPopulate event for tv_1
TreeViewItem tvi1, tvi2
long ll_row, ll_rowcount

//Create instance variable datastore
ids_info = CREATE datastore
ids_info.DataObject = "d_music"
ids_info.SetTransObject(SQLCA)

//Use the label of the item being populated
// as the retrieval argument
This.GetItem(handle, tvi1)
ll_rowcount = ids_info.Retrieve(tvi1.Label)

//Use information retrieved from the database
//to populate the expanded item
FOR ll_row = 1 to ll_rowcount
      This.InsertItemLast(handle, &
      ids_info.GetItemString(ll_row, &
      music_title'), 2)
LOOP
```

Destroying DataStore instances

When the window containing the TreeView control closes, this example destroys the DataStore instances:

```
//Close event for w_treeview
DESTROY ids_data
DESTROY ids_info
```

CHAPTER 7

Manipulating Graphs in Windows

About this chapter

This chapter describes how to write code that allows you to access and change a graph in your application at runtime.

Contents

Using graphs

In PocketBuilder, there are two ways to display graphs:

- In a DataWindow, using data retrieved from the DataWindow data source
- In a graph control in a window or user object, using data supplied by your application code

This chapter discusses the graph control and describes how your application code can supply data for the graph and manipulate its appearance.

For information about graphs in DataWindows, see Chapter 12, "Manipulating Graphs in DataWindows," and the online Help.

To learn about designing graphs and setting graph properties in the painters, see the *User's Guide*.

Working with graph controls in code

Graph controls in a window can be enabled or disabled, visible or invisible, and can be used in drag and drop. You can also write code that uses events of graph controls and additional graph functions.

Properties of graph controls

You can access (and optionally modify) a graph by addressing its properties in code at runtime. There are two kinds of graph properties:

- **Properties of the graph definition itself** These properties are initially set in the painter when you create a graph. They include a graph's type, title, axis labels, whether axes have major divisions, and so on.
- **Properties of the data** These properties are relevant only at runtime, when data has been loaded into the graph. They include the number of series in a graph (series are created at runtime), colors of bars or columns for a series, whether the series is an overlay, text that identifies the categories (categories are created at runtime), and so on.

Events of graph controls

Graph controls have the events listed in Table 7-1.

Table 7-1: Graph control events

Clicked	DragLeave
Constructor	DragWithin
Destructor	GetFocus
DoubleClicked	LoseFocus
DragDrop	Other
DragEnter	RButtonDown

So, for example, you can write a script that is invoked when a user clicks a graph or drags an object on a graph (as long as the graph is enabled).

Functions for graph controls

You use the PowerScript graph functions in Table 7-2 to manipulate data in a graph.

Table 7-2: PowerScript graph functions

Function	Action
AddCategory	Adds a category
AddData	Adds a data point
AddSeries	Adds a series
DeleteCategory	Deletes a category
DeleteData	Deletes a data point
DeleteSeries	Deletes a series
ImportClipboard	Copies data from the clipboard to a graph
ImportFile	Copies the data in a text file to a graph
ImportString	Copies the contents of a string to a graph
InsertCategory	Inserts a category before another category
InsertData	Inserts a data point before another data point in a series
InsertSeries	Inserts a series before another series

Function	Action
ModifyData	Changes the value of a data point
Reset	Resets the graph's data

Populating a graph with data

This section shows how you can populate an empty graph with data.

Using AddSeries

You use AddSeries to create a series. AddSeries has this syntax:

graphName.**AddSeries** (*seriesName*)

AddSeries returns an integer that identifies the series that was created. The first series is numbered 1, the second is 2, and so on. Typically you use this number as the first argument in other graph functions that manipulate the series.

To create a series named Stellar, code:

```
int SNum
SNum = gr_1.AddSeries("Stellar")
```

Using AddData

You use AddData to add data points to a specified series. AddData has this syntax:

graphName.**AddData** (*seriesNumber, value*, *categoryLabel*)

The first argument to AddData is the number assigned by PocketBuilder to the series. To add two data points to the Stellar series, whose number is stored by the variable *SNum* (as shown above), code:

```
gr_1.AddData(SNum, 12, "Q1") // Category is Q1
gr_1.AddData(SNum, 14, "Q2") // Category is Q2
```

Getting a series number
You can use the FindSeries function to determine the number PocketBuilder has assigned to a series. FindSeries returns the series number. This is useful when you write general-purpose functions to manipulate graphs.

An example

Suppose you want to graph quarterly printer sales. Here is a script that populates the graph with data:

```
gr_1.Reset( All! ) // Resets the graph.
// Create first series and populate with data.
```

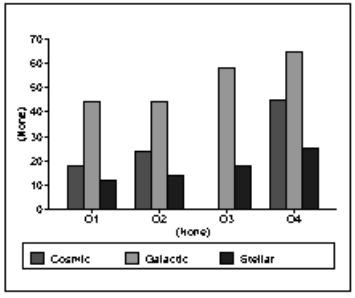
```
int SNum
SNum = gr_1.AddSeries("Stellar")
gr_1.AddData(SNum, 12, "Q1") // Category is Q1.
gr_1.AddData(SNum, 14, "Q2") // Category is Q2.
gr_1.Adddata(SNum, 18, "Q3") // Category is Q3.
gr_1.AddData(SNum, 25, "Q4") // Category is Q4.
// Create second series and populate with data.
SNum = gr_1.AddSeries("Cosmic")

// Use the same categories as for series 1 so the data
// appears next to the series 1 data.
gr_1.AddData(SNum, 18, "Q1")
gr_1.AddData(SNum, 24, "Q2")
gr_1.Adddata(SNum, 38, "Q3")
gr_1.AddData(SNum, 45, "Q4")

// Create third series and populate with data.
SNum = gr_1.AddSeries("Galactic")
gr_1.AddData(SNum, 44, "Q1")
gr_1.AddData(SNum, 44, "Q2")
gr_1.Adddata(SNum, 58, "Q3")
gr_1.AddData(SNum, 65, "Q4")
```

Figure 7-1shows the resulting graph.

Figure 7-1: Quarterly printer sales



You can add, modify, and delete data in a graph in a window through graph functions anytime during execution.

For more information

For complete information about each graph function, see the online Help.

Modifying graph properties

When you define a graph in the Window or User Object painter, you specify its behavior and appearance. For example, you might define a graph as a column graph with a certain title, divide its Value axis into four major divisions, and so on. Each of these entries corresponds to a property of a graph. For example, all graphs have an enumerated attribute GraphType, which specifies the type of graph.

When dynamically changing the graph type
If you change the graph type, be sure to change other properties as needed to define the new graph properly.

You can change these graph properties at runtime by assigning values to the graph's properties in scripts. For example, to change the type of the graph gr_emp to Column, you could code:

```
gr_emp.GraphType = ColGraph!
```

To change the title of the graph at runtime, you could code:

```
gr_emp.Title = "New title"
```

How parts of a graph are represented

Graphs consist of parts: a title, a legend, and axes. Each of these parts has a set of display properties. These display properties are themselves stored as properties in a subobject (structure) of Graph called grDispAttr.

For example, graphs have a Title property, which specifies the title's text. Graphs also have a property TitleDispAttr, of type grDispAttr, which itself contains properties that specify all the characteristics of the title text, such as the font, size, whether the text is italicized, and so on.

Similarly, graphs have axes, each of which also has a set of properties. These properties are stored in a subobject (structure) of Graph called grAxis. For example, graphs have a property Values of type grAxis, which contains properties that specify the properties of the Value axis, such as whether to use autoscaling of values, the number of major and minor divisions, the axis label, and so on.

Here is a representation of the properties of a graph:

```
Graph
        int Height
        int Depth
        grGraphType GraphType
        boolean Border
        string Title
        …
grDispAttr TitleDispAttr, LegendDispAttr, PieDispAttr
        string FaceName
        int TextSize
        boolean Italic
        …
grAxis Values, Category, Series
        boolean AutoScale
        int MajorDivisions
        int MinorDivisions
        string Label
        …
```

Referencing parts of a graph

You use dot notation to reference these display properties. For example, one of the properties of a graph's title is whether the text is italicized or not. That information is stored in the boolean Italic property in the TitleDispAttr property of the graph.

For example, to italicize the title of graph gr_emp, code:

```
gr_emp.TitleDispAttr.Italic = TRUE
```

Similarly, to turn on autoscaling of a graph's Values axis, code:

```
gr_emp.Values.Autoscale = TRUE
```

To change the label text for the Value axis, code:

```
gr_emp.Values.Label = "New label"
```

To change the alignment of text in the Value axis's label text, code:

```
gr_emp.Values.LabelDispAttr.Alignment = Left!
```

For a complete list of graph properties, see the online Help for the graph control.

Accessing data properties

To access properties related to a graph's data during execution, you use PowerScript graph functions. The graph functions related to data fall into several categories:

- Functions that provide information about a graph's data
- Functions that save data from a graph
- Functions that change the color, fill patterns, and other visual properties of data

How to use the functions

To call functions for a graph in a graph control, use the following syntax:

graphControlName.FunctionName (*Arguments*)

For example, to get a count of the categories in the window graph gr_printer, code:

```
Ccount = gr_printer.CategoryCount()
```

Different syntax for graphs in DataWindows
The syntax for the same functions is more complex when the graph is in a DataWindow, like this:

DataWindowName.FunctionName ("*graphName*", *otherArguments…*)

For more information, see Chapter 12, "Manipulating Graphs in DataWindows."

Getting information about the data

The PowerScript functions in Table 7-3 allow you to get information about data in a graph at runtime.

Table 7-3: PowerScript functions for information at runtime

Function	Information provided
CategoryCount	The number of categories in a graph
CategoryName	The name of a category, given its number
DataCount	The number of data points in a series
FindCategory	The number of a category, given its name
FindSeries	The number of a series, given its name
GetData	The value of a data point, given its series and position (superseded by GetDataValue, which is more flexible)

Function	Information provided
GetDataPieExplode	The percentage of the pie's radius that the pie slice is to be moved away from the center (exploded)
GetDataStyle	The color, fill pattern, or other visual property of a specified data point
GetDataValue	The value of a data point, given its series and position
GetSeriesStyle	The color, fill pattern, or other visual property of a specified series
SeriesCount	The number of series in a graph
SeriesName	The name of a series, given its number

Saving graph data

The PowerScript functions in Table 7-4 allow you to save data from the graph.

Table 7-4: PowerScript functions for saving graph data

Function	Action
Clipboard	Copies a bitmap image of the specified graph to the clipboard
SaveAs	Saves the data in the underlying graph to the clipboard or to a file in one of a number of formats

Modifying colors, fill patterns, and other data

The PowerScript functions in Table 7-5 allow you to modify the appearance of data in a graph.

Table 7-5: PowerScript functions for changing appearance of data

Function	Action
ResetDataColors	Resets the color for a specific data point
SetDataPieExplode	Explodes a slice in a pie graph
SetDataStyle	Sets the color, fill pattern, or other visual property for a specific data point
SetSeriesStyle	Sets the color, fill pattern, or other visual property for a series

PART 3

Programming DataWindows and DataStores

This part describes techniques for using DataWindow objects and DataStores to implement data access features in the applications you develop with PocketBuilder.

CHAPTER 8

About DataWindow Technology

About this chapter

This chapter describes what DataWindow objects are and the ways you can use them in various application architectures and programming environments.

Contents

About DataWindow objects and controls

DataWindow technology is implemented in two parts:

- **A DataWindow object** The DataWindow object defines the data source and presentation style for the data.
- **A DataWindow control** The DataWindow control is a visual container for a DataWindow object. You write code that calls methods of the container to manipulate the DataWindow object.

You can also use a DataStore object as a nonvisual container for a DataWindow object. DataStores provide DataWindow functionality for retrieving and manipulating data without the onscreen display. For more information about DataStore objects, see Chapter 11, “Using DataStore Objects.”

DataWindow objects

A DataWindow object is an object that you use to retrieve, present, and manipulate data from a relational database or other data source (such as an Excel worksheet or dBASE file). You can specify whether the DataWindow object supports updating of data.

DataWindow objects have knowledge about the data they are retrieving. You can specify display formats, presentation styles, and other data properties to make the data meaningful to users.

You define DataWindow objects in the DataWindow painter.

Presentation styles and data sources

When you define a DataWindow object, you choose a presentation style and a data source.

Presentation styles

A presentation style defines a typical style of report and handles how rows are grouped on the page. You can customize the way the data is displayed in each presentation style. Table 8-1 lists the presentation styles available.

Table 8-1: DataWindow presentation styles

Presentation style	Description
Tabular	Data columns across the page and headers above each column. Several rows are viewable at once.
Freeform	Data columns going down the page with labels next to each column. One row displayed at a time.
Grid	Row-and-column format like a spreadsheet with grid lines. Users can move borders and columns.
Group	A tabular style with rows grouped under headings. Each group can have summary fields with computed statistics.
Graph	Graphical presentation of data.

For examples of the presentation styles, see the *User's Guide*.

Data sources

The data source specifies where the data in the DataWindow comes from and what data items are displayed. Data can come from tables in a database, or you can import data from a file or specify the data in code. For databases, the data specification is saved in a SQL statement. In all cases, the DataWindow object saves the names of the data items to display, as well as their data types.

Table 8-2: Data sources you can use for a DataWindow

Data source	Description
Quick Select	The data comes from one or more tables in a SQL database. The tables must be related through a foreign key. You need to choose only columns, selection criteria, and sorting.
SQL Select	You want more control over the select statement that is generated for the data source. You can specify grouping, computed columns, and so on.
Query	The data has already been selected and the SQL statement is saved in a query object that you have defined in the Query painter. When you define the DataWindow object, the query object is incorporated into the DataWindow and does not need to be present when you run the application.
Stored Procedure	The data is defined in a database stored procedure.
External	The data is not stored in a database, but is imported from a file (such as a tab-separated or dBASE file) or populated from code.

Basic process

Using a DataWindow involves two main steps:

1 Use the DataWindow wizard to create a DataWindow object.

In the wizard, you define the data source, presentation style, and some properties of the object, such as display formats, validation rules, sorting and filtering criteria, and graphs.

2 Use the DataWindow painter to design a DataWindow object.

In the painter, you define other properties of the object, such as display formats, validation rules, sorting and filtering criteria, and graphs.

3 Put a DataWindow control in a window or visual user object and associate a DataWindow object with it.

It is through this control that your application communicates with the DataWindow object you designed in the DataWindow painter. You write code to manipulate the DataWindow control and the DataWindow object it contains. A complete set of events and methods programmed in PowerScript provides control over all aspects of the DataWindow. Typically, your code retrieves and updates data, changes the appearance of the data, handles errors, and shares data between DataWindow controls.

DataWindow controls

The DataWindow control is a visual container for DataWindow objects in a PocketBuilder application. You can use it in a window to present an interactive display of data. The user can view and change data and send changes to the database.

The DataWindow supports data retrieval with retrieval arguments and data update. You can use edit styles, display formats, and validation rules for consistent data entry and display. The DataWindow provides many methods for manipulating the DataWindow, including Modify for changing DataWindow object properties.

You can share a result set between several DataWindow controls and update databases on Windows CE devices or on the desktop. Using Mobilink—a database synchronization tool that comes with Sybase SQL Anywhere Studio—you can synchronize data between a client database on a Windows CE device and a server database on the desktop.

Development environment

You add DataWindow controls to windows or visual user objects in the Window or Visual User Object painters. The DataWindow control is in a drop-down palette of controls on the PainterBars for these painters. After you add the control to the window or user object, you can associate a DataWindow object with it in the painter.

You write scripts that control the DataWindow's behavior and manipulate the data it retrieves. The DataWindow object associated with the control determines what data is retrieved and how it is displayed.

You can use the Browser to examine the properties, events, and methods of DataWindow controls on the System tab page. If you have a library open that contains DataWindow objects, you can examine the internal properties of the DataWindow object on the Browser's DataWindow tab page.

Database connections

The DataWindow uses an SQL Anywhere ODBC database driver or an UltraLite database driver for database connectivity. Users can connect to a data source on their Windows CE devices and make updates to those sources.

To make a connection, you can use the internal Transaction object of the DataWindow, or you can make the connection with a separate transaction object.

PocketBuilder provides a default Transaction object, SQLCA; you can define additional Transaction objects if you need to make additional connections. When you connect with a separate Transaction object, you can control when SQL COMMIT and ROLLBACK statements occur, and you can use the same connection for multiple controls.

For more information about using a Transaction object with a DataWindow, see Chapter 9, "Using DataWindow Objects." For more information about Transaction objects, see Chapter 16, "Using Transaction Objects."

Coding

You write scripts in the Window or User Object painter to connect to the database, retrieve data, process user input, and update data.

To take advantage of object inheritance, you can define a standard visual user object inherited from a DataWindow control and add your own customizations. You can reuse the customized DataWindow control in multiple applications.

You can also create a customized version of a DataStore object. You create DataStore objects—the nonvisual version of a DataWindow control— in scripts. For more information, see Chapter 11, "Using DataStore Objects."

Libraries and applications

You store DataWindow objects in PocketBuilder libraries (PKL files) during development. When you build your application, you can include the DataWindow objects in the application executable or in PocketBuilder dynamic libraries (PKD files).

For more information about designing DataWindow objects and building a PocketBuilder application, see the *User's Guide*.

CHAPTER 9

Using DataWindow Objects

About this chapter

This chapter describes how to use DataWindow objects in an application.

Contents

Before you begin

This chapter assumes that you know how to build DataWindow objects in the DataWindow painter, as described in the *User's Guide*.

About using DataWindow objects

Building DataWindow objects

Before you can use a DataWindow object in an application, you need to build it. PocketBuilder has separate painters for database management, DataWindow definition, and library management.

You define and edit a DataWindow object in the DataWindow painter. You specify its data source and presentation style, then enhance the object by specifying display formats, edit styles, and more.

Managing DataWindow objects

Several painters let you manage and package your DataWindow objects for use in applications.

In particular, you can maintain DataWindow objects in one or more libraries (PKL files). When you are ready to use your DataWindow objects in applications, you can package them in more compact runtime libraries (PKD files).

Using DataWindow objects

After you build a DataWindow object in the DataWindow painter, you can use it to display and process information from the appropriate data source. The sections that follow explore the details of how to do this.

Putting a DataWindow object into a control

The DataWindow control is a container for DataWindow objects in an application. It provides properties, methods, and events for manipulating the data and appearance of the DataWindow object. The DataWindow control is part of the user interface of your application.

You also use DataWindow objects in the nonvisual DataStore and in drop-down DataWindows. For more information about DataStores, see Chapter 11, "Using DataStore Objects." For more information about drop-down DataWindows, see the *User's Guide*.

This section has information about:

- Names for DataWindow controls and DataWindow objects
- Procedures for working with DataWindow controls at design time:
  - Inserting a DataWindow control
  - Specifying a DataWindow object
  - Editing the DataWindow object in the control
- Specifying the DataWindow object at runtime

Names for DataWindow controls and DataWindow objects

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 or form, it gets a default name. You should change the name to be something meaningful for your application.

It is useful to give the name of the control a prefix of `dw_`. For example, if the DataWindow control lists customers, you might want to name it `dw_customer`.

Using the name
In code, always refer to a DataWindow by the name of the *control* (such as dw_customer). Do not refer to the DataWindow *object* that is in the control.

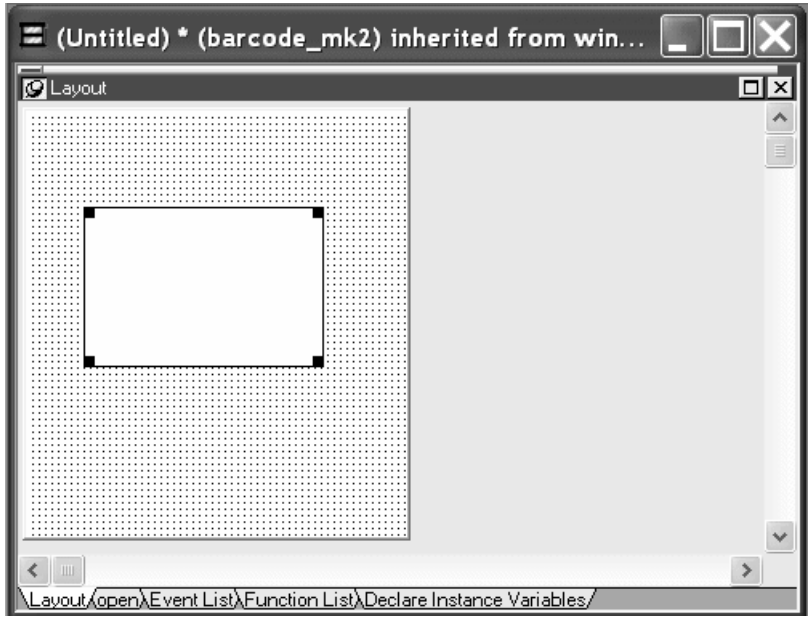
The DataWindow object name To avoid confusion, you should use different prefixes for DataWindow objects and DataWindow controls. The prefix d_ is commonly used for DataWindow objects. For example, if the name of the DataWindow control is dw_customer, you might want to name the corresponding DataWindow object d_customer.

Working with the DataWindow control in PocketBuilder

Inserting a DataWindow control

To use the DataWindow object in an application, you add a DataWindow control to a window, then associate that control with the DataWindow object.

Figure 9-1: DataWindow control before association with an object



❖ **To place a DataWindow control in a window or custom visual user object:**

1 Open the window or user object that will contain the DataWindow control.

2 Select Insert>Control>DataWindow from the menu bar.

3 Click where you want the control to display.

PocketBuilder places an empty DataWindow control in the window.

4 (Optional) Resize the DataWindow control by selecting it and dragging one of the handles, or changing its position properties on the Other page of the Properties view.

Specifying a DataWindow object

After placing the DataWindow control, you associate a DataWindow object with the control.

❖ **To associate a DataWindow object with the control:**

1 In the DataWindow Properties view, click the Browse button for the DataObject property.

2 Select the DataWindow object that you want to place in the control and click OK.

The name of the DataWindow object displays in the DataObject box in the DataWindow Properties view.

3 (Optional) Change the properties of the DataWindow control as needed.

Allowing users to move DataWindow controls
If you want users to be able to move a DataWindow control at runtime, give it a title and select the Title Bar check box. Then users can move the control by dragging the title bar.

Defining reusable 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.

To be able to define these behaviors once and reuse them in each window, you create a standard visual user object based on the DataWindow control. Define the user object's properties 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 do not need to specify it again.

For more information about creating and using user objects, see the *User's Guide*.

Editing the DataWindow object in the control

Once you have associated a DataWindow object with a DataWindow control in a window, you can go directly to the DataWindow painter to edit the associated DataWindow object.

❖ **To edit an associated DataWindow object:**

- Select Modify DataWindow from the DataWindow control's pop-up menu.

  PocketBuilder opens the associated DataWindow object in the DataWindow painter.

Specifying the DataWindow object at runtime

Changing the DataWindow at runtime

When you associate a DataWindow object with a control in a window, you are setting the initial value of the DataWindow control's DataObject property. At runtime, this tells your application to create an instance of the DataWindow object specified in the control's DataObject property and use it in the control.

At runtime, you change the DataWindow object associated with a DataWindow control by setting the DataObject property to one of the DataWindow objects built into the application.

Creating a DataWindow object at runtime
You can also create a new DataWindow object at runtime and associate it with a control. For more information, see Chapter 10, "Dynamically Changing DataWindow Objects."

To display the DataWindow object d_emp_hist from the library *emp.pkl* in the DataWindow control dw_emp, you can code:

```
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 as part of the window definition.

When you change the DataWindow object at runtime, you may need to call setTrans or setTransObject again. For more information, see "Setting the transaction object for the DataWindow control" on page 123.

Preventing redrawing

Use the SetRedraw method to turn off redrawing in order to avoid flicker and reduce redrawing time when you are making several changes to the properties of an object or control. Dynamically changing the DataWindow object at runtime implicitly turns redrawing on. To turn redrawing off again, call the SetRedraw method every time you change the DataWindow object:

```
dw_emp.DataObject = "d_emp_hist"
dw_emp.SetRedraw(FALSE)
```

Accessing the database

Before you can display data in a DataWindow control, you must get the data stored in the data source into that control. The most common way to get the data is to access a database.

An application must perform several steps to access a database:

1 Set the appropriate values for the transaction object.

2 Connect to the database.

3 Set the transaction object for the DataWindow control.

4 Retrieve and update data.

5 Disconnect from the database.

This section provides instructions for setting the transaction object for a DataWindow control and for using the DataWindow object to retrieve and update data.

To learn more about setting values for the transaction object, connecting to the database, and disconnecting from the database, see Chapter 16, “Using Transaction Objects.”

Setting the transaction object for the DataWindow control

There are two ways to handle database connections and transactions for the DataWindow control. You can use:

- Internal transaction management
- Transaction management with a separate transaction object

The two methods provide different levels of control over database transactions.

If you change the DataWindow object
If you change the DataWindow object associated with a DataWindow control at runtime, you need to call the SetTrans or SetTransObject method again.

Internal transaction management

When the DataWindow control uses internal transaction management, it handles connecting, disconnecting, commits, and rollbacks. It 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 method is executed), the DataWindow issues an internal CONNECT statement, does the appropriate data access, then issues an internal DISCONNECT.

When to use it

If the number of available connections at your site is limited, you might want to use internal transaction management because connections are not held open.

Internal transaction management is appropriate in simple situations when you are doing pure retrievals (such as in reporting) and do not need to hold database locks—when application control over committing or rolling back transactions is not an issue.

Do *not* use internal transaction management when:

- Your application requires the best possible performance

  Internal transaction management is slow and uses considerable system resources because it must connect and disconnect for every database access.

- You want control over when a transaction is committed or rolled back

  Because internal transaction management must disconnect after a database access, any changes are always committed immediately.

How it works

To use internal transaction management, you specify connection values for a transaction object, which could be the automatically instantiated SQLCA. Then you call the SetTrans method, which copies the values from a specified transaction object to the DataWindow control's internal transaction object.

```
SQLCA.DBMS = ProfileString("myapp.ini", &
       "database", "ODBC", " ")
... // Set more connection parameters
dw_employee.SetTrans(SQLCA)
dw_employee.Retrieve( )
```

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.

For more information about transaction objects, see Chapter 16, "Using Transaction Objects."

Transaction management with a separate transaction object

When you use a separate transaction object, you control the duration of the database transaction. Your scripts explicitly connect to and disconnect from the database. If the transaction object's AutoCommit property is set to false, you also program when an update is committed or rolled back.

Typically, a script for data retrieval or update involves these statements:

Connect
SetTransObject
Retrieve or Update
Commit or Rollback
Disconnect

In PocketBuilder, you use embedded SQL for connecting and committing.

The transaction object also stores error messages returned from the database in its properties. You can use the error information to determine whether to commit or roll back database changes.

When to use it

When the DataWindow control uses a separate transaction object, you have more control of the database processing and are responsible for managing the database transaction.

There are several reasons to use a separate transaction object:

- You have several DataWindow controls that connect to the same database and you want to make one database connection for all of them, saving the overhead of multiple connections
- You want to control transaction processing
- You require the improved performance provided by keeping database connections open

How it works

The SetTransObject method associates a transaction object with the DataWindow control. PocketBuilder has a default transaction object called SQLCA that is automatically instantiated. You can set its connection properties, connect, and assign it to the DataWindow control.

The following statement uses SetTransObject to associate the DataWindow control dw_emp with the default transaction object (SQLCA):

```
// Set connection parameters in the transaction object
SQLCA.DBMS = "ODBC"
SQLCA.database = ...
CONNECT USING SQLCA;
dw_emp.SetTransObject(SQLCA)
dw_emp.Retrieve( )
```

Instead of or in addition to using the predefined SQLCA transaction object, you can define your own transaction object in a script. This is necessary if your application needs to connect to more than one database at the same time.

The following statement uses SetTransObject to associate dw_customer with a programmer-created transaction object (trans_customer):

```
transaction trans_customer
trans_customer = CREATE transaction
// Set connection parameters in the transaction object
trans_customer.DBMS = "ODBC"
trans_customer.database = ...
CONNECT USING trans_customer;
dw_customer.SetTransObject(trans_customer)
dw_customer.Retrieve( )
```

For more information about SetTrans and SetTransObject methods, see the online Help.

Retrieving and updating data

You call the following two methods to access a database through a DataWindow control:

Retrieve
Update

Basic data retrieval

After you have set the transaction object for your DataWindow control, you can use the Retrieve method to retrieve data from the database and insert it into that control:

```
dw_emp.Retrieve( )
```

Using retrieval arguments

About retrieval arguments

Retrieval arguments qualify the SELECT statement associated with the DataWindow object, reducing the rows retrieved according to some criteria. For example, in the following SELECT statement, Salary is a retrieval argument defined in the DataWindow painter:

```
SELECT Name, emp.sal FROM Employee
       WHERE emp.sal > :Salary
```

When you call the Retrieve method, you supply a value for Salary. In PocketBuilder, the code looks like this:

```
dw_emp.Retrieve( 50000 )
```

When coding Retrieve with arguments, specify them in the order in which they are defined in the DataWindow object. Your Retrieve method can provide more arguments than a particular DataWindow object expects. Any extra arguments are ignored. This allows you to write a generic Retrieve that works with several different DataWindow objects. You can specify any number of retrieval arguments.

Omitting retrieval arguments

If your DataWindow object takes retrieval arguments but you do not pass them in the Retrieve method, the DataWindow control prompts the user for them when Retrieve is called.

Updating data

After users have made changes to data in a DataWindow control, you can use the Update method to save those changes in the database.

In PocketBuilder, the code looks like this:

```
dw_emp.Update()
```

Update sends to the database all inserts, changes, and deletions made in the DataWindow control since the last Update method call. When you are using an external transaction object, you can then commit (or roll back) those database updates with SQL statements.

For more specifics on how a DataWindow control updates the database (that is, which SQL statements are sent in which situations), see "Updating the database" on page 141.

Examples

The following example shows code that connects, retrieves, updates, commits or rolls back, and disconnects from the database.

Although the example shows all database operations in a single script or function, most applications separate these operations. For example, an application could connect to the database in the application Open event, retrieve and update data in one or more window scripts, and disconnect from the database in the application Close event.

The following statements retrieve and update data using the transaction object EmpSQL and 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 from the database
DISCONNECT USING EmpSQL;
```

Handling retrieval or update errors

A production application should include error tests after each database operation. For more about checking for errors, see “Handling DataWindow errors” on page 136.

Importing data from an external source

If the data for a DataWindow is not coming from a database (that is, the data source was defined as External in the DataWindow wizard), you can use these methods to import data into the DataWindow control:

ImportClipboard
ImportFile
ImportString

You can also get data into the DataWindow by using the SetItem method or by using a DataWindow expression.

For more information on the SetItem method and DataWindow expressions, see “Manipulating data in a DataWindow control” next.

Manipulating data in a DataWindow control

To handle user requests to add, modify, and delete data in a DataWindow, you can write code to process that data, but first you need to understand how DataWindow controls manage data.

How a DataWindow control manages 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:

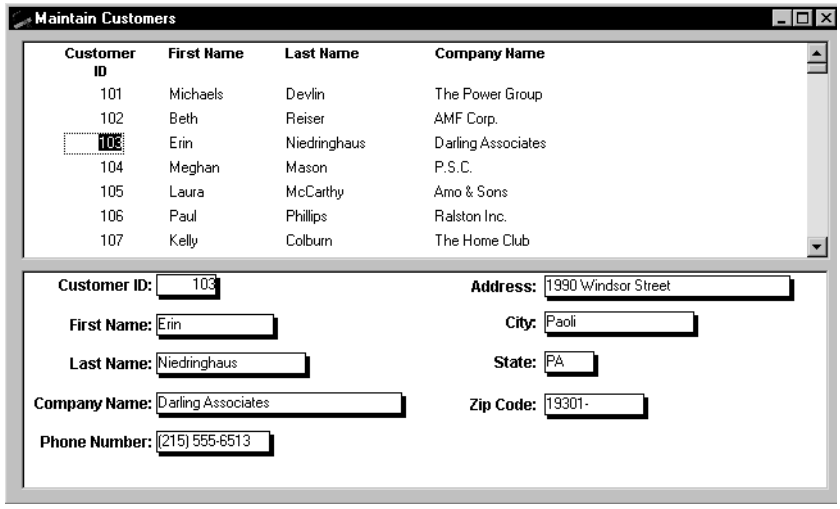
Table 9-1: DataWindow buffers

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 through code

About the edit control

As the user moves around the DataWindow control, the DataWindow places an edit control over the current cell (row and column):

Figure 9-2: Editing text in a DataWindow control



The contents of the edit control are 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 stylus, or pressing Up arrow or Down arrow from the soft input panel or other keyboard), the DataWindow processes the data and either accepts or rejects it, depending on whether it meets the requirements specified for the column.

If the data is accepted, the text is moved to the current row and column in the DataWindow Primary buffer. The data in the Primary buffer for a particular column is referred to as an item.

Events for changing text and items

When data is changed in the edit control, several events occur.

Table 9-2: Events triggered by changing text and items

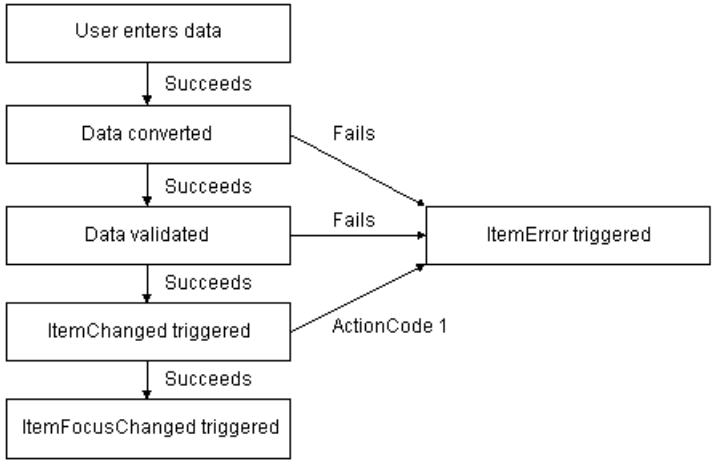
Event	Description
EditChanged	Occurs for each keystroke the user types in the edit control
ItemChanged	Occurs when a cell has been modified and loses focus
ItemError	Occurs when new data fails the validation rules for the column
ItemFocusChanged	Occurs when the current item in the control changes

How text is processed in the edit control

When the data in a column in a DataWindow has been changed and the column loses focus (for example, because the user tabs to the next column), the following sequence of events occurs:

1. The DataWindow control converts the text into the correct datatype for the column. For example, if the user is in a numeric column, the DataWindow control 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 datatype, the DataWindow control 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. If you set an action/return code of 1 in the ItemChanged event, the DataWindow control rejects the data and does not allow the focus to change. In this case, the ItemError event is triggered.
4. If the ItemChanged event accepts the data, the ItemFocusChanged event is triggered next and the data is stored as an item in a buffer.

Figure 9-3: How text is processed in edit controls



Action/return codes for events

You can affect the outcome of events by specifying numeric values in the event's program code. For example, step 3 above describes how you can force data to be rejected by using a RETURN statement with a code of 1 in the ItemChanged event.

For information about codes for individual events, see the *DataWindow Reference* in the online Help.

Accessing and manipulating the text in the edit control

Using methods

The following methods allow you to access the text in the edit control:

- GetText – obtains the text in the edit control.
- SetText – sets the text in the edit control.

In event code

In addition to these methods, the following events provide access to the text in the edit control:

EditChanged
ItemChanged
ItemError

Use the Data parameter, which is passed into the event, to access the text of the edit control. In your code for these events, you can test the text value and perform special processing depending on that value.

For an example, see "Coding the ItemChanged event" next.

Manipulating the text

When you want to further manipulate the contents of the edit control within your DataWindow control, you can use any of these methods:

Clear	Position	SelectedStart
Copy	ReplaceText	SelectedText
Cut	Scroll	SelectText
LineCount	SelectedLength	TextLine
Paste	SelectedLine	

For more information about these methods, see the *DataWindow Reference* in the online Help.

Coding 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 code for the ItemChanged event to do some additional processing. For example, you could perform some tests, set a code to reject the data, have the column regain focus, and trigger the ItemError event.

Example

The following sample code for the ItemChanged event for a DataWindow control called dw_Employee sets the return code in dw_Employee to reject data that is less than the employee's age, which is specified in a SingleLineEdit text box control in the window.

```
int a, age
age = Integer(sle_age.text)
a = Integer(data)

// Set the return code to 1 in the ItemChanged
// event to tell PocketBuilder to reject the data
// and not change the focus.
IF a < age THEN RETURN 1
```

Coding the ItemError event

The ItemError event is triggered if there is a problem with the data. By default, it rejects the data value and displays a message box. You can write code for the ItemError event to do some other processing. For example, you can set a code to accept the data value, or reject the data value but allow focus to change.

For more information about the events of the DataWindow control, see the *DataWindow Reference* in the online Help.

Accessing the items in a DataWindow

You can access data values in a DataWindow by using methods or DataWindow data expressions. Both methods allow you to access data in any buffer and to get original or current values.

The method you use depends on how much data you are accessing and whether you know the names of the DataWindow columns when the script is compiled.

Using methods

There are several methods for manipulating data in a DataWindow control.

GetItem methods You call GetItem methods to obtain the data that has been accepted into a specific row and column. You can also use them to check the data in a specific buffer before you update the database. You must use the method appropriate for the column's datatype.

These methods obtain the data in a specified row and column in a specified buffer: GetItemDate, GetItemDateTime, GetItemDecimal, GetItemNumber, GetItemString, GetItemTime.

For example, the following statement assigns the value from the empname column of the first row to the variable *ls_Name*:

```
ls_Name = dw_1.GetItemString (1, "empname")
```

SetItem method This method sets the value of a specified row and column: SetItem.

This statement sets the value of the empname column in the first row to the string "Waters":

```
dw_1.SetItem(1, "empname", "Waters")
```

For more information about the methods listed above, see the online Help.

Using expressions

DataWindow data expressions refer to single items, columns, blocks of data, selected data, or the whole DataWindow. You use dot notation to construct data expressions in PocketBuilder.

The Object property of the DataWindow control lets you specify expressions that refer directly to the data of the DataWindow object in the control. This direct data manipulation allows you to access small and large amounts of data in a single statement, without calling methods:

```
dw_1.Object.jobtitle[3] = "Programmer"
```

The next statement sets the value of the first column in the first row in the DataWindow to Smith:

```
dw_1.Object.Data[1,1] = "Smith"
```

For complete instructions on how to construct DataWindow data expressions, see the *DataWindow Reference* in the online Help.

Using other DataWindow methods

There are many more methods you can use to perform activities in DataWindow controls. The more common ones are listed in Table 9-3.

Table 9-3: Common methods in DataWindow controls

Method	Purpose
AcceptText	Applies the contents of the edit control to the current item in the DataWindow control.
DeleteRow	Removes the specified row from the DataWindow control, placing it in the Delete buffer; does not delete the row from the database.
Filter	Displays rows in the DataWindow control based on the current filter.
GetRow	Returns the current row number.
InsertRow	Inserts a new row.
Reset	Clears all rows in the DataWindow control.
Retrieve	Retrieves rows from the database.
RowsCopy, 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 in the DataWindow control.

You can see a complete list of DataWindow methods in the PocketBuilder Browser.

For complete information on DataWindow methods, see the *DataWindow Reference* in the online Help.

Accessing the properties of a DataWindow object

About DataWindow object properties

DataWindow object properties store the information that controls the behavior of a DataWindow object. They are not properties of the DataWindow control, but of the DataWindow object displayed in the control. The DataWindow object is itself made up of individual controls—column, text, graph, and drawing controls—that have DataWindow object properties.

You establish initial values for DataWindow object properties in the DataWindow painter. You can also get and set property values at runtime in your code.

You can access the properties of a DataWindow object by using the Describe and Modify methods or DataWindow property expressions. Which you use depends on the type of error checking you want to provide and on whether you know the names of the controls within the DataWindow object and properties you want to access when the script is compiled.

For lists and descriptions of DataWindow object properties, see the *DataWindow Reference* in the online Help.

Using methods to access object properties

You can use the following methods to work with the properties of a DataWindow object:

- Describe – reports the values of properties of a DataWindow object and controls within the DataWindow object.
- Modify – modifies a DataWindow object by specifying a list of instructions that change the DataWindow object's definition.

For example, the following statements assign the value of the Border property for the empname column to a string variable:

```
string ls_border
ls_border = dw_1.Describe("empname.Border")
```

The following statement changes the value of the Border property for the empname column to 1:

```
dw_emp.Modify("empname.Border=1")
```

About dynamic DataWindow objects
Using Describe and Modify, you can provide an interface through which application users can alter the DataWindow object at runtime. For example, you can change the appearance of a DataWindow object or allow an application user to create ad hoc reports.

For more information, see Chapter 10, "Dynamically Changing DataWindow Objects."

Using expressions

DataWindow property expressions provide access to properties with fewer nested strings. In PocketBuilder, you can handle problems with incorrect object and property names in the Error event.

Use the Object property and dot notation. For example:

```
integer li_border
li_border = Integer(dw_1.Object.empname.Border)
dw_1.Object.empname.Border = 1
```

Handling DataWindow errors

There are several types of errors that can occur during DataWindow processing:

- Data items that are invalid (discussed in "Manipulating data in a DataWindow control" on page 128)
- Failures when retrieving or updating data
- Attempts to access invalid or nonexistent properties or data

This section explains how to handle the last two types of errors.

Retrieve and Update errors and the DBError event

Retrieve and update testing

When using the Retrieve or Update method in a DataWindow control, you should test the method's return code to see whether the activity succeeded.

Table 9-4: Return codes for the Retrieve and Update methods

Method	Return code	Meaning
Retrieve	>=1	Retrieval succeeded; returns the number of rows retrieved
	-1	Retrieval failed; DBError event triggered
	0	No data retrieved
Update	1	Update succeeded
	-1	Update failed; DBError event triggered

Do not test the SQLCode attribute
After issuing a SQL statement (such as CONNECT, COMMIT, or DISCONNECT) or the equivalent method of the transaction object, you should always test the success/failure code (the SQLCode attribute in the transaction object). However, you should not use this type of error checking following a retrieval or update made in a DataWindow.

For more information about error handling after a SQL statement, see Chapter 16, "Using Transaction Objects."

Example

If you want to commit changes to the database only if an update succeeds, you can code:

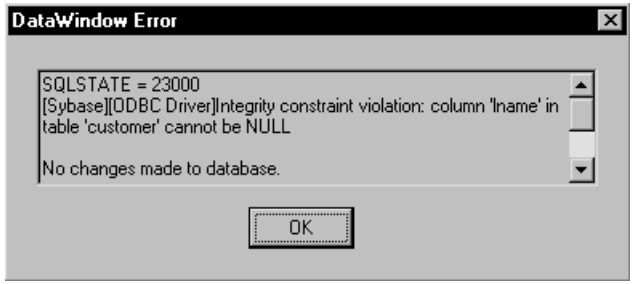
```
IF dw_emp.Update() > 0 THEN
       COMMIT USING EmpSQL;
ELSE
       ROLLBACK USING EmpSQL;
END IF
```

Using the DBError event

The DataWindow control triggers its DBError event whenever there is an error following a retrieval or update; that is, if the Retrieve or Update methods return –1. For example, if you try to insert a row that does not 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, the DataWindow control displays a message box describing the error message from the DBMS.

Figure 9-4: Sample error message displayed from DBError event



In many cases you may want to code your own processing in the DBError event and suppress the default message box. Here are some tips for doing this:

Table 9-5: Tips for processing messages from DBError event

To	Do this
Get the DBMS's error code	Use the SQLDBCode argument of the DBError event
Get the DBMS's message text	Use the SQLErrText argument of the DBError event
Suppress the default message box	Specify an action/return code of 1

About DataWindow action/return codes
Some events for DataWindow controls have codes that you can set to override the default action that occurs when the event is triggered. The codes and their meaning depend on the event. You set the code with a RETURN statement.

Example

Here is a sample script for the DBError event:

```
// Database error -195 means that some of the
// required values are missing
IF sqldbcode = -195 THEN
       MessageBox("Missing Information",  &
           "You have not supplied values for all " &
           +"the required fields.")
END IF
// Return code suppresses default message box
RETURN 1
```

At runtime, the user would see the message box after the error.

Figure 9-5: Example of a user-defined message for the DBError event

Errors in property and data expressions and the Error event

A DataWindow control's Error event is triggered whenever an error occurs in a data or property expression at runtime. These expressions that refer to data and properties of a DataWindow object may be valid under some runtime conditions but not others. The Error event allows you to respond with error recovery logic when an expression is not valid.

PocketBuilder syntax checking

When you use a data or property expression, the PowerScript compiler checks the syntax only as far as the Object property. Everything following the Object property is evaluated at runtime. For example, in the following expression, the column name emp_name and the property Visible are not checked until runtime:

```
dw_1.Object.emp_name.Visible = "0"
```

If the emp_name column did not exist in the DataWindow, or if you had misspelled the property name, the compiler would not detect the error. However, at runtime, PocketBuilder would trigger the DataWindow control's Error event.

Using a Try-Catch block

The Error event is triggered even if you have surrounded an error-producing data or property expression in a try-catch block. The catch statement is executed after the Error event is triggered, but only if you do not code the Error event or do not change the default Error event action from ExceptionFail!. The following example shows a property expression in a try-catch block:

```
TRY
   dw_1.Object.emp_name.Visible = "0"
CATCH (dwruntimeerror dw_e)
   MessageBox ("DWRuntimeError", dw_e.text)
END TRY
```

Determining the cause of the error

The Error event has several arguments that provide information about the error condition. You can check the values of the arguments to determine the cause of the error. For example, you can obtain the internal error number and error text, the name of the object whose script caused the error, and the full text of the script where the error occurred. The information provided by the Error event's arguments can be helpful in debugging expressions that are not checked by the compiler.

If you catch a DWRuntimeError error, you can use the properties of that class instead of the Error event arguments to provide information about the error condition. The following table displays the correspondences between the Error event arguments and the DWRuntimeError properties.

Table 9-6: Correspondence between Error event arguments and DWRuntimeError properties

Error event argument	DWRuntimeError property
errornumber	number
errorline	line
errortext	text
errorwindowmenu	objectname
errorobject	class
errorscript	routinename

Controlling the outcome of the event

When the Error event is triggered, you can have the application ignore the error and continue processing, substitute a different return value, or escalate the error by triggering the SystemError event. In the Error event, you can set two arguments passed by reference to control the outcome of the event.

Table 9-7: Setting arguments in the Error event

Argument	Description
Action	A value you specify to control the application's course of action as a result of the error. Values are: ExceptionIgnore! ExceptionSubstituteReturnValue! ExceptionFail! (default action)
ReturnValue	A value whose datatype matches the expected value that the DataWindow would have returned. This value is used when the value of action is ExceptionSubstituteReturnValue!.

For a complete description of the arguments of the Error event, see the online Help.

When to substitute a return value
The ExceptionSubstituteReturnValue! action allows you to substitute a return value when the last element of an expression causes an error. Do not use ExceptionSubstituteReturnValue! to substitute a return value when an element in the middle of an expression causes an error.

The ExceptionSubstituteReturnValue! action is most useful for handling errors in data expressions.

Updating the database

After users have made changes to data in a DataWindow control, you can use the Update method 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 or Retrieve method was executed.

How the DataWindow control updates the database

For database updates, the DataWindow control determines what 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.

Table 9-8: DataWindow item status for rows and columns

Status Name	Numeric value	Applies to
New!	2	Rows
NewModified!	3	Rows
NotModified!	0	Rows and columns
DataModified!	1	Rows and columns

The named values are values of the enumerated datatype dwItemStatus. You must use the named values, which end in an exclamation point.

How status is set

When data is retrieved When data is retrieved into a DataWindow, all rows and columns initially have a 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 method, the column status for that column changes to DataModified!. Once the status for any column in a retrieved row changes to DataModified!, 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 method, 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!.

When a DataWindow column has a default value, the column's status does not change to DataModified! until the user makes at least one actual change to a column in that row.

When Update is called

For rows in the Primary and Filter buffers When the Update method is called, the DataWindow control generates SQL INSERT and UPDATE statements for rows in the Primary and/or Filter buffers based upon the following row statuses:

Table 9-9: Row status after INSERT and UPDATE statements

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 updatable column list maintained by the DataWindow object

  For more information about setting the update characteristics of the DataWindow object, see the *User's Guide*.

- The column has a column status of DataModified!

The DataWindow control 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 null values in that column.

For rows in the Delete buffer The DataWindow control generates SQL DELETE statements for any rows that were moved into the Delete buffer using the DeleteRow method. However, if a row has a row status of New! or NewModified! before DeleteRow is called, 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 behavior from taking place. For example, you might copy a row from one DataWindow to another; and after the user modifies the row, you might want to issue an UPDATE statement instead of an INSERT statement.

You use the SetItemStatus method to programmatically change a DataWindow's row or column status information. Use the GetItemStatus method to determine the status of a specific row or column.

Changing column status

You use SetItemStatus to change the column status from DataModified! to NotModified!, or vice versa.

Change column status when you change row status
Changing the row status changes the status of all columns in that row to NotModified!, so if the Update method is called, no SQL update is produced. You must change the status of columns to be updated after you change the row status.

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.

Table 9-10: Effects of changing from one row status to another

Original status	*Specified status*			
	New!	**NewModified!**	**DataModified!**	**NotModified!**
New!	-	Yes	Yes	No
NewModified!	No	-	Yes	New!
DataModified!	NewModified!	Yes	-	Yes
NotModified!	Yes	Yes	Yes	-

In the preceding table, *Yes* 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* 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!. Change the column's status to DataModified! to ensure that an update results in a SQL Update.

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!.

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 a production report, you:

- Determine the type of report you want to produce
- Build a DataWindow object to display data for the report
- Place the DataWindow object in a DataWindow control on a window
- Write code to perform the processing required to populate the DataWindow control and print the contents as a report

Planning and building the DataWindow object

To design the report, you create a DataWindow object. You select the data source and presentation style and then:

- 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 object 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 might 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 format with a small amount of data before you print a large report.

Printing the report

After you build the DataWindow object and fill in print specifications, you can place it in a DataWindow control on a window or form, as described in "Putting a DataWindow object into a control" on page 118.

To allow users to print the report, your application needs code that performs the printing logic. For example, you can place a button on the window or form, then write code that is run when the user clicks the button.

To print the contents of a single DataWindow control or DataStore, call the Print method. For example, this statement prints the report in the DataWindow control dw_Sales:

```
dw_Sales.Print(TRUE)
```

For information about the Print method, see the *DataWindow Reference*.

Separate DataWindow controls in a single print job

If the window has multiple DataWindow controls, you can use multiple PrintDataWindow method calls in a script to print the contents of all the DataWindow controls in one print job.

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)
```

For information about PocketBuilder system functions for printing, see the *PowerScript Reference.*

CHAPTER 10

Dynamically Changing DataWindow Objects

About this chapter

This chapter describes how to modify and create DataWindow objects at runtime.

Contents

About dynamic DataWindow processing

Basics

DataWindow objects and all entities in them (such as columns, text, graphs, and pictures) each have a set of properties. You can look at and change the values of these properties at runtime using DataWindow methods or property expressions. You can also create DataWindow objects at runtime.

A DataWindow object that is modified or created at runtime is called a dynamic DataWindow object.

What you can do

Using this dynamic capability, 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.

Modifying a DataWindow object

At runtime, you can modify the appearance and behavior of a DataWindow object by doing one of the following:

- Changing the values of its properties
- Adding or deleting controls from the DataWindow object

Changing property values

You can use the Modify method or a property expression to set property values. This lets you change settings that you ordinarily specify during development in the DataWindow painter.

Before changing a property, 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 properties of a DataWindow object or a control in a DataWindow object, use the Describe method or a property expression.

Using expressions in property values

With some DataWindow properties, you can assign a value through an expression that the DataWindow evaluates at runtime, instead of having to assign a value directly. For example, the following statement displays a salary in red if it is less than $12,000, and in black otherwise:

```
dw_1.Modify("salary.Color &
       = '0 ~t if(salary <12000,255,0)' ")
```

For more information

The syntax is different for expressions in code versus expressions specified in the DataWindow painter. For the correct syntax and information about which properties can be assigned expressions, see the *DataWindow Reference* in the online Help.

Adding and deleting controls within the DataWindow object

You can also use the Modify method to:

- Create new objects in a DataWindow object

  This lets you add DataWindow controls (such as text, bitmaps, and graphic controls) dynamically to the DataWindow object.

  For how to get a good idea of the correct Create syntax, see “Specifying the DataWindow object syntax” on page 149.

- Destroy controls in a DataWindow object

  This lets you dynamically remove controls you no longer need.

Tool for easier coding of DataWindow syntax

Included with PocketBuilder is DW Syntax, a tool that makes it easy to build the correct syntax for property expressions, Describe, Modify, and SyntaxFromSQL statements. You click buttons to specify which properties of a DataWindow you want to use, and DW Syntax automatically builds the appropriate syntax, which you can copy and paste into your application code.

To access DW Syntax, select File>New and select the Tool tab.

Viewing DataWindow object properties in the Browser

You can use the Browser to get a list of DataWindow properties: on the DataWindow page, select a DataWindow object in the left pane and Properties in the right pane. To see the properties for a control in a DataWindow object, double-click the DataWindow object name, then select the control.

Creating a DataWindow object

This section describes how to create a DataWindow object by calling the Create method in an application.

DataWindow painter
You should use the techniques described here for creating a DataWindow from syntax only if you cannot accomplish what you need to in the DataWindow painter. The usual way of creating DataWindow objects is to use the DataWindow painter.

To learn about creating DataWindow objects in the DataWindow painter, see the *User's Guide*.

You use the Create method to create a DataWindow object dynamically at runtime. Create generates a DataWindow object using source code that you specify. It replaces the DataWindow object currently in the specified DataWindow control with the new DataWindow object.

Resetting the transaction object
The Create method destroys the association between the DataWindow control and the transaction object. As a result, you need to reset the control's transaction object by calling the SetTransObject or SetTrans method after you call Create.

To learn how to associate a DataWindow control with a transaction object, see Chapter 9, “Using DataWindow Objects.”

Specifying the DataWindow object syntax

There are several ways to specify or generate the syntax required for the Create method:

- Use the SyntaxFromSQL method of the transaction object
- Use the DataWindow.Syntax property of the DataWindow object

- Create the syntax yourself

Using SyntaxFromSQL You are likely to 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.

In PocketBuilder, SyntaxFromSQL is a method of the transaction object. The transaction object must be connected when you call the method.

SyntaxFromSQL has three required arguments:

- A string containing the SELECT statement for the DataWindow object
- A string identifying the presentation style and other settings
- The name of a string you want to fill with any error messages that might be returned

SyntaxFromSQL returns the complete syntax for a DataWindow object that is built using the specified SELECT statement.

Using the DataWindow.Syntax property You can obtain the source code of an existing DataWindow object to use as a model or for making minor changes to the syntax. Many values in the source code syntax correspond to properties of the DataWindow object.

This example gets the syntax of the DataWindow object in the DataWindow control, dw_1, and displays it in the text box control, textb_dw_syntax:

```
var dwSyntax
dwSyntax = dw_1.Describe("datawindow.syntax")
textb_dw_syntax.value = dwSyntax
```

Creating the syntax yourself You need to create the syntax yourself to use some of the advanced dynamic DataWindow features, such as creating a group break.

The DataWindow source code syntax that you need to supply to the Create method can be very complex. 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.

For more information on Create and Describe methods, as well as DataWindow object properties and syntax, see the *DataWindow Reference* in the online Help.

Providing query ability to users

When you call the Retrieve method for a DataWindow control, the rows specified in the DataWindow object's SELECT statement are retrieved. You can give users the ability to further specify which rows are retrieved at runtime by putting the DataWindow into query mode. To do that, you use the Modify method or a property expression (the examples here use Modify).

Limitations
You cannot use query mode in a DataWindow object 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 three figures show what happens when query mode is used. First, data is retrieved into the DataWindow. There are 36 rows in this example.

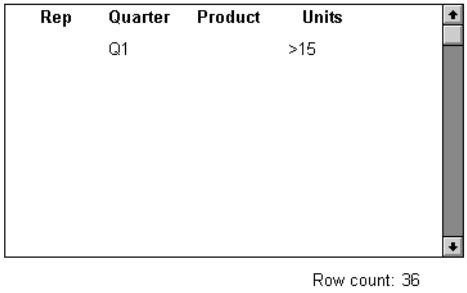
Figure 10-1: Example of data retrieved from a database table

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

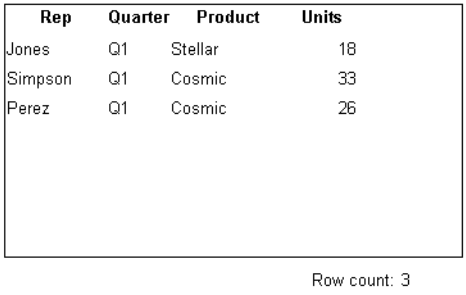
Next, query mode is turned on. The retrieved data disappears and users are presented with empty rows where they can specify selection criteria. Here the user wants to retrieve rows where Quarter = Q1 and Units > 15.

Figure 10-2: Example of a DataWindow in Query mode



Next, Retrieve is called and query mode is turned off. The DataWindow control adds the criteria to the SELECT statement, retrieves the three rows that meet the criteria, and displays them to the user.

Figure 10-3: Example of a DataWindow with results from query



You can turn query mode back on, allow the user to revise the selection criteria, and retrieve again.

Using query mode

❖ **To provide query mode to users at runtime:**

1 Turn query mode on by coding:

```
dw_1.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, just as you do when using Quick Select to define a DataWindow object's data source.

   Criteria entered in one row are joined together with the AND logical operator; criteria in different rows are joined together with the OR logical operator. Valid operators are =, <>, <, >, <=, >=, LIKE, IN, AND, and OR.

   For more information about Quick Select, see the *User's Guide*.

3 Call AcceptText and Retrieve, then turn off query mode to display the newly retrieved rows:

```
dw_1.AcceptText()
dw_1.Modify("datawindow.querymode=no")
dw_1.Retrieve()
```

   The DataWindow control adds the newly defined selection criteria to the WHERE clause of the SELECT statement, then retrieves and displays the specified rows.

Revised SELECT statement
You can look at the revised SELECT statement that is sent to the DBMS when data is retrieved with criteria. To do so, look at the *sqlsyntax* argument in the SQLPreview event of the DataWindow control.

How the criteria affect the SELECT statement

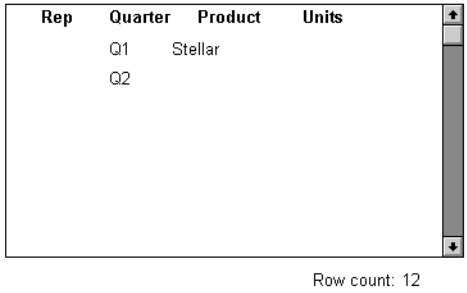
Criteria specified by the user are added to the SELECT statement that originally defined the DataWindow object.

For example, suppose the original SELECT statement for the printer table was:

```
SELECT printer.rep, printer.quarter, printer.product,
printer.units
FROM printer
WHERE printer.units < 70
```

Figure 10-4 displays a DataWindow with user-entered criteria for the Q1 quarter for Stellar printers, and for the Q2 quarter for all printer products.

Figure 10-4: Example of a DataWindow with a new user query



The SELECT statement generated from this user query 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, Use the QueryClear property.

```
dw_1.Modify("datawindow.queryclear=yes")
```

Sorting in query mode You can allow users to sort rows in a DataWindow while specifying criteria in query mode using the QuerySort property. The following statement makes the first row in the DataWindow dedicated to sort criteria (just as in Quick Select in the DataWindow wizard).

```
dw_1.Modify("datawindow.querysort=yes")
```

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, use the Criteria.Override_Edit property for each column:

```
dw_1.Modify("mycolumn.criteria.override_edit=yes")
```

You can also specify this in the DataWindow painter by checking Override Edit on the General property page for the column. 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:

```
dw_1.Modify("mycolumn.criteria.required=yes")
```

You can also specify this in the DataWindow painter by checking Equality Required on the General property page for the column. Doing this ensures that the user specifies criteria for the column and that the criteria for the column use the = operator rather than other operators, such as < or >=.

CHAPTER 11

Using DataStore Objects

About this chapter

This chapter describes how to use DataStore 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 the *User's Guide*.

About DataStores

A DataStore is a nonvisual DataWindow control. DataStores act just like DataWindow controls except that they do not have many of the visual characteristics associated with DataWindow controls. Like a DataWindow control, a DataStore has a DataWindow object associated with it.

When to use a DataStore

DataStores are useful when you need to access data but do not need the visual presentation of a DataWindow control. DataStores allow you to:

- Perform background processing against the database without having to hide DataWindow controls in a window

  Suppose that the DataWindow object displayed in a DataWindow control is suitable for online display but not for saving to a file. In this case, you could define a second DataWindow object for saving that has the same result set description and assign this object to a DataStore. You could then share data between the DataStore and the DataWindow control. Whenever the user asked to save the data in the window, you could save the contents of the DataStore.

- Hold data used to show multiple views of the same information

  When a window shows multiple views of the same information, you can use a DataStore to hold the result set. By sharing data between a DataStore and one or more DataWindow controls, you can provide different views of the same information without retrieving the data more than once.

- Manipulate table rows without using embedded SQL statements

  In places where an application calls for row manipulation without the need for display, you can use DataStores instead of embedded SQL statements to handle the database processing. DataStores typically perform faster at runtime than embedded SQL statements. Also, because the SQL is stored with the DataWindow object when you use a DataStore, you can easily reuse the SQL.

DataStore methods

Most of the methods and events available for DataWindows are also available for DataStores. However, some of the methods that handle online interaction with the user are not available. For example, DataStores support the Retrieve, Update, InsertRow, and DeleteRow methods, but not GetClickedRow and SetRowFocusIndicator.

Prompting for information

When you are working with DataStores, you cannot prompt the user for more information by using functionality that causes a dialog box to display. Here are some examples of ways to manage information entry with DataStores:

SetSort and SetFilter You can use the SetSort and SetFilter methods to specify sort and filter criteria for a DataStore object, just as you would with a DataWindow control. However, when you are working with a DataWindow control, if you pass a null value to either SetSort or SetFilter, the DataWindow prompts the user to enter information.

When you are working with a DataStore, you must supply a valid value with the method call. You must also supply a valid value when you share data between a DataStore and a DataWindow control; you can pass a null value to the DataWindow control, but not the DataStore.

Prompt for Criteria You can define your DataWindow objects so that the user is prompted for retrieval criteria before the DataWindow retrieves data. This feature works with DataWindow controls only. It is not supported with DataStores.

SaveAs If you are working with a DataStore, you must supply the *filename* argument when you use the SaveAs method. With a DataWindow object, you can pass an empty string for the *filename* argument so that the user is prompted for a file name to save to.

Prompt for Printing For DataWindow controls, you can specify that a print setup dialog box display at execution time, either by checking the Prompt Before Printing check box on the DataWindow object's Print Specifications property page, or by setting the DataWindow object's Print.Prompt property in a script. This is not supported with DataStores.

Retrieval arguments If you call the Retrieve method for a DataWindow control that has a DataWindow object that expects an argument, but do not specify the argument in the method call, the DataWindow prompts the user for a retrieval argument. This behavior is not supported with DataStores.

DataStores have some visual methods

Many of the methods and events that pertain to the visual presentation of the data in a DataWindow do not apply to DataStores. However, because you can print the contents of a DataStore and also import data into a DataStore, DataStores have some visually oriented events and methods. For example, DataStores support the SetBorderStyle and SetSeriesStyle methods so that you can control the presentation of the data at print time. Similarly, DataStores support the ItemError event, because data imported from a string or file that does not pass the validation rules for a column triggers this event.

For a complete list of the methods and events for the DataStore object and information about each method, see the *DataWindow Reference* in the online Help.

DataStores require no visual overhead

Unlike DataWindow controls, DataStores do not require any visual overhead in a window. Using a DataStore is therefore more efficient than hiding a DataWindow control in a window.

Working with a DataStore

To use a DataStore, you first need to create an instance of the DataStore object in a script and assign the DataWindow object to the DataStore. Then, if the DataStore is intended to retrieve data, you need to set the transaction object for the DataStore. Once these setup steps have been performed, you can retrieve data into the DataStore, share data with another DataStore or DataWindow control, or perform other processing.

Examples

The following script uses a DataStore to retrieve data from the database. First it instantiates the DataStore object and assigns a DataWindow object to the DataStore. Then it sets the transaction object and retrieves data into the DataStore:

```
datastore lds_datastore
lds_datastore = CREATE datastore
lds_datastore.DataObject = "d_cust_list"
lds_datastore.SetTransObject (SQLCA)
lds_datastore.Retrieve()
/* Perform some processing on the data... */
```

Using a custom DataStore object

This section describes how to extend a DataStore in PocketBuilder by creating a user object.

You may want to use a custom version of the DataStore object that performs specialized processing. To define a custom DataStore, you use the User Object painter. There you specify the DataWindow object for the DataStore, and you can optionally write scripts for events or define your own methods, user events, and instance variables.

Using a custom DataStore involves two procedures:

1 In the User Object painter, define and save a standard class user object inherited from the built-in DataStore object.

2 Use the custom DataStore in your PocketBuilder application.

Once you have defined a custom DataStore in the User Object painter, you can write code that uses the user object to perform the processing you want.

For instructions on using the User Object painter in PocketBuilder, see the *User's Guide*.

❖ **To define the standard class user object:**

1 Select Standard Class User Object on the PB Objects page in the New dialog box.

2 Select "datastore" as the built-in system type that you want your user object to inherit from, and click OK.

The User Object painter workspace displays so that you can define the custom object.

3 Specify the name of the DataWindow object in the DataObject box in the Properties view and click OK.

4 Customize the DataStore by scripting the events for the object, or by defining methods, user events, and instance variables.

5 Save the object.

❖ **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 Script view and select the event for which you want to write the script.

3 Write code that uses the user object to do the necessary processing.

Here is a simple code example that shows how to use a custom DataStore to retrieve data from the database. First it instantiates the custom DataStore object, then it sets the transaction object and retrieves data into the DataStore:

```
uo_cust_dstore lds_cust_dstore
lds_cust_dstore = CREATE uo_cust_dstore
lds_cust_dstore.SetTransObject (SQLCA)
lds_cust_dstore.Retrieve()
/* Perform some processing on the data... */
```

Notice that this script does not assign the DataWindow object to the DataStore. This is because the DataWindow object is specified in the user object definition.

Changing the DataWindow object at runtime
When you associate a DataWindow object with a DataStore in the User Object painter, you are setting the initial value of the DataStore's DataObject property. At runtime, you can change the DataWindow object for the DataStore by changing the value of the DataObject property.

4 Compile the script and save your changes.

Accessing and manipulating data in a DataStore

To access data using a DataStore, you need to read the data from the data source into the DataStore.

If the data source is a database

If the data for the DataStore 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. The steps you perform to communicate with the database are the same steps you use for a DataWindow control.

For more information about communicating with the database, see "Accessing the database" on page 122.

If the data source is not a database

If the data for the DataWindow object does not come from a database (that is, the data source was defined as External in the DataWindow painter), you can use the following methods to import data into the DataStore:

ImportClipboard
ImportFile
ImportString

You can put data in the DataStore by using a DataWindow data expression, or by using the SetItem method. You can use the same property and data expressions as for a DataWindow control.

For more information on accessing data in a DataStore, see the *DataWindow Reference* in the online Help.

About the DataStore buffers

Like a DataWindow control, a DataStore uses three buffers to manage data. The buffers are described in Table 11-1.

Table 11-1: DataStore buffers

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

The DataStore object has an Edit control. However, the Edit control for a DataStore behaves in a slightly different manner from the Edit control for a DataWindow. The Edit control for a DataWindow keeps track of text entered by the user in the current cell (row and column); the Edit control for a DataStore is used to manage data imported from an external source. The text in the Edit control for a DataStore cannot be changed directly by the user. It must be manipulated programmatically.

Programming with DataStores

There are many methods for manipulating DataStore objects. Table 11-2 lists some of the more commonly used.

Table 11-2: Common methods in DataStore objects

Method	Purpose
DeleteRow	Deletes the specified row from the DataStore.
Filter	Filters rows in the DataStore based on the current filter criteria.
InsertRow	Inserts a new row.
Print	Sends the contents of the DataStore to the current printer.
Reset	Clears all rows in the DataStore.
Retrieve	Retrieves rows from the database.
RowsCopy	Copies rows from one DataStore to another DataStore or DataWindow control.
RowsMove	Moves rows from one DataStore to another DataStore or DataWindow control.
ShareData	Shares data among different DataStores or DataWindow controls. See “Sharing information” on page 164.
Sort	Sorts the rows of the DataStore based on the current sort criteria.
Update	Sends to the database all inserts, changes, and deletions that have been made since the last Update.

For information about DataStore methods, see the *DataWindow Reference* in the online Help.

Dynamic DataWindow objects The methods in Table 11-2 manipulate data in the DataStore but do not change the definition of the underlying DataWindow object. In addition, you can use the Modify and Describe methods to access and manipulate the definition of a DataWindow object. Using these methods, you can change the DataWindow object at runtime. For example, you can change the appearance of a DataWindow or allow your user to create ad hoc reports.

If you assign a DataWindow object to a DataStore dynamically, you must make sure that the DataWindow object is available in a PKD file or is listed in a PKR file used to create the executable.

For more information, see Chapter 10, “Dynamically Changing DataWindow Objects.”

Using DataStore properties and events This chapter mentions only a few of the properties and events that you can use to manipulate DataStores. For more information about DataStore properties and events, see the *DataWindow Reference* in the online Help.

Sharing information

The ShareData method allows you to share a result set between two different DataStores or DataWindow controls. When you share information, you remove the need to retrieve the same data multiple times.

The ShareData method shares data retrieved by one DataWindow control or DataStore (called the primary DataWindow) with another DataWindow control or DataStore (the secondary DataWindow).

Result set descriptions must match

When you share data, the result set descriptions for the DataWindow objects must be the same. However, the SELECT statements can be different. For example, you could use the ShareData method to share data between DataWindow objects that have the following SELECT statements, because the result set descriptions are the same:

```
SELECT dept_id from dept
SELECT dept_id from dept where dept_id = 200
SELECT dept_id from employee
```

You can also share data between two DataWindow objects where the source of one is a database and the source of the other is external. As long as the lists of columns and their datatypes match, you can share the data.

What is shared?

When you use the ShareData method, the following information is shared:

Primary buffer
Delete buffer
Filter buffer
Sort order

ShareData does not share the formatting characteristics of the DataWindow objects. That means you can use ShareData to apply different presentations to the same result set.

When you alter the result set

If you perform an operation that affects the result set for either the primary or the secondary DataWindow, the change affects both of the objects sharing the data. Operations that alter the buffers or the sort order of the secondary DataWindows are rerouted to the primary DataWindow. For example, if you call the Update method for the secondary DataWindow, the update operation is applied to the primary DataWindow also.

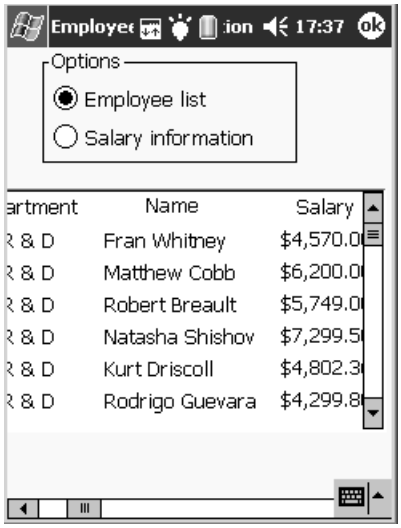
Turning off sharing data

To turn off the sharing of data, you use the ShareDataOff method. When you call ShareDataOff for a primary DataWindow, any secondary DataWindows are disassociated and no longer contain data. When you call ShareDataOff for a secondary DataWindow, that DataWindow no longer contains data, but the primary DataWindow and other secondary DataWindows are not affected.

In most cases you do not need to turn off sharing, because the sharing of data is turned off automatically when a window is closed and any DataWindow controls (or DataStores) associated with the window are destroyed.

Example: printing data from a DataStore

Suppose you have a window called w_employees that allows users to retrieve, update, and print employee data retrieved from the database:



The DataWindow object displayed in the DataWindow control is suitable for online display but not for printing. In this case, you could define a second DataWindow object for printing that has the same result set description as the object used for display and assign the second object to a DataStore. You could then share data between the DataStore and the DataWindow control. Whenever the user asked to print the data in the window, you could print the contents of the DataStore.

Required third-party software
You must install the FieldSoftware PrinterCE SDK before you can use print methods in PocketBuilder applications deployed to a device or emulator. An evaluation version of this software is available from the FieldSoftware Web site at http://www.fieldsoftware.com.

When the window or form opens

The code you write begins by establishing the hand pointer as the current row indicator for the dw_employees DataWindow control. Then the script sets the transaction object for dw_employees and issues a Retrieve method to retrieve some data. After retrieving data, the script creates a DataStore using the instance variable or data member ids_datastore, and assigns the DataWindow object d_employees to the DataStore. The final statement of the script shares the result set for the dw_employees DataWindow control with the DataStore.

This code is for the window's Open event:

```
dw_employees.SetRowFocusIndicator(Hand!)
dw_employees.SetTransObject(SQLCA)
dw_employees.Retrieve()

ids_datastore = CREATE datastore
ids_datastore.DataObject = "d_employees"
dw_employees.ShareData(ids_datastore)
```

Code for the Update button

Code for the cb_update button applies the update operation to the dw_employees DataWindow control.

This code is for the Update button's Clicked event:

```
IF dw_employees.Update() = 1 THEN
       COMMIT using SQLCA;
       MessageBox("Save","Save succeeded")
ELSE
       ROLLBACK using SQLCA;
       MessageBox("Save","Save failed")
END IF
```

Code for the Print button

The Clicked event of the cb_print button prints the contents of ids_datastore. Because the DataWindow object for the DataStore is d_employees, the printed output uses the presentation specified for this object.

This code is for the Print button's Clicked event:

```
ids_datastore.Print()
```

When the window or form closes

When the window closes, the DataStore gets destroyed.

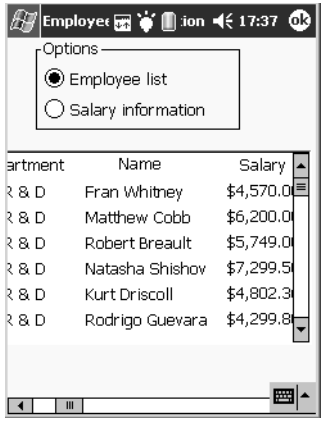
This code is for the window's Close event:

```
destroy ids_datastore
```

Example: using two DataStores to process data

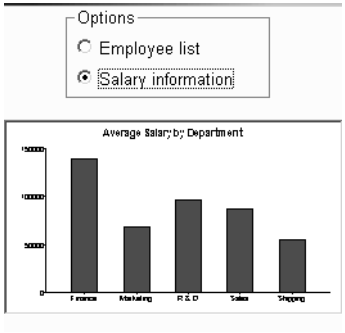
Suppose you have a window called w_multi_view that shows multiple views of the same result set. When the Employee List radio button is selected, the window shows a list of employees retrieved from the database.

Figure 11-1: Displaying a list of employees in a DataWindow



When the Employee Salary Information radio button is selected, the window displays a graph that shows employee salary information by department.

Figure 11-2: Displaying a graph of employee salary information



This window has one DataWindow control called dw_display. It uses two DataStores to process data retrieved from the database. The first DataStore (ids_emp_list) shares its result set with the second DataStore (ids_emp_graph). The DataWindow objects associated with the two DataStores have the same result set description.

When the window or form opens

When the window opens, the application sets the pointer to the hourglass shape. Then the code creates the two DataStores and sets the DataWindow objects for the DataStores. Next the code sets the transaction object for ids_emp_list and issues a Retrieve method to retrieve some data.

After retrieving data, the code shares the result set for ids_emp_list with ids_emp_graph. The next-to-last statement triggers the Clicked event for the Employee List radio button. The final statement resets the pointer.

This code is for the window's Open event:

```
SetPointer(HourGlass!)
ids_emp_list = Create DataStore
ids_emp_graph = Create DataStore

ids_emp_list.DataObject = "d_emp_list"
ids_emp_graph.DataObject = "d_emp_graph"

ids_emp_list.SetTransObject(sqlca)
ids_emp_list.Retrieve()
ids_emp_list.ShareData(ids_emp_graph)
rb_emp_list.EVENT Clicked()
SetPointer(Arrow!)
```

Code for the Employee List radio button

The code for the Employee List radio button (called rb_emp_list) sets the DataWindow object for the DataWindow control to be the same as the DataWindow object for ids_emp_list. Then the script displays the data by sharing the result set for the ids_emp_list DataStore with the DataWindow control.

This code is for the Employee List radio button's Clicked event:

```
dw_display.DataObject = ids_emp_list.DataObject
ids_emp_list.ShareData(dw_display)
```

Code for the Employee Salary Information radio button

The code for the Employee Salary Information radio button (called rb_graph) is similar to the code for the List radio button. It sets the DataWindow object for the DataWindow control to be the same as the DataWindow object for ids_emp_graph. Then it displays the data by sharing the result set for the ids_emp_graph DataStore with the DataWindow control.

This code is for the Employee Salary Information radio button's Clicked event:

```
dw_display.DataObject = ids_emp_graph.DataObject
ids_emp_graph.ShareData(dw_display)
```

When the window or form closes

When the window closes, the DataStores get destroyed.

This code is for the window's Close event:

```
Destroy ids_emp_list
Destroy ids_emp_graph
```

Use garbage collection
Do not destroy the objects if they might still be in use by another process. Rely on garbage collection instead.

CHAPTER 12

Manipulating Graphs in DataWindows

About this chapter

This chapter describes how to write code that allows you to access and change a graph in your application at runtime.

Contents

Using graphs

It is common for developers to design DataWindow objects that include one or more graphs. When users need to understand and analyze data quickly, a bar, line, or pie graph can often be the most effective format to display.

To learn about designing graphs, see the *User's Guide*.

Working with graphs in your code

The following sections describe how you can access (and optionally modify) a graph by addressing its properties in code at runtime. There are two kinds of graph properties:

- **Properties of the graph definition itself** These properties are initially set in the DataWindow painter when you create a graph. They include a graph's type, title, axis labels, whether axes have major divisions, and so on.
- **Properties of the data** These properties are relevant only at runtime, when data has been loaded into the graph. They include the number of series in a graph (series are created at runtime), colors of bars or columns for a series, whether the series is an overlay, text that identifies the categories (categories are created at runtime), and so on.

Using graphs in other controls
Although you will probably use graphs most often in DataWindow objects, you can also add graph controls to windows or visual user objects. Additional PowerScript functions and events are available for use with graph controls.

For more information, see Chapter 7, “Manipulating Graphs in Windows.”

Modifying graph properties

When you define a graph in the DataWindow painter, you specify its behavior and appearance. For example, you might define a graph as a column graph with a certain title, divide its Value axis into four major divisions, and so on. Each of these entries corresponds to a property of a graph. For example, all graphs have a property GraphType, which specifies the type of graph.

When dynamically changing the graph type
If you change the graph type, be sure also to change the other properties as needed to define the new graph properly.

You can change these graph properties at runtime by assigning values to the graph’s properties in code.

Property expressions

You can modify properties using property expressions. For example, to change the type of the graph gr_emp to Column, you could code:

```
dw_empinfo.Object.gr_emp.GraphType = ColGraph!
```

To change the title of the graph at runtime, you could code:

```
dw_empinfo.Object.gr_emp.Title = "New title"
```

Modify method

In any environment, you can use the Modify method to reference parts of a graph.

For example, to change the title of graph gr_emp in DataWindow control dw_empinfo, you could code:

```
dw_empinfo.Modify("gr_emp.Title = 'New title'")
```

For a complete list of graph properties, see the *DataWindow Reference* in the online Help.

How parts of a graph are represented

Graphs consist of parts: a title, a legend, and axes. Each of these parts has a set of display properties. These display properties are themselves stored as properties in a subobject (structure) of Graph called *grDispAttr*.

For example, graphs have a Title property, which specifies the text for the title. Graphs also have a property TitleDispAttr, of type grDispAttr, which itself contains properties that specify all the characteristics of the title text, such as the font, size, whether the text is italicized, and so on.

Similarly, graphs have axes, each of which also has a set of properties. These properties are stored in a subobject (structure) of Graph called *grAxis*. For example, graphs have a property Values, of type grAxis, which specifies the properties of the Value axis, such as whether to use auto scaling of values, the number of major and minor divisions, the axis label, and so on.

Here is a representation of the properties of a graph:

```
Graph
        int Height
        int Depth
        grGraphType GraphType
        boolean Border
        string Title
        …
grDispAttr TitleDispAttr, LegendDispAttr, PieDispAttr
        string FaceName
        int TextSize
        boolean Italic
        …
grAxis Values, Category, Series
        boolean AutoScale
        int MajorDivisions
        int MinorDivisions
        string Label
        …
```

Referencing parts of a graph

You use dot notation or the Describe and Modify methods to reference the display properties of the various parts of a graph. For example, one of the properties of a graph's title is whether the text is italicized or not. That information is stored in the boolean Italic property in the TitleDispAttr property of the graph.

This example changes the label text for the Value axis of graph gr_emp in the DataWindow control dw_empinfo:

```
dw_empinfo.Object.gr_emp.Values.Label="New label"
```

You can use the Browser to examine the properties of a DataWindow object that contains a graph. For descriptions of graph properties, see the *DataWindow Reference* in the online Help.

Accessing data properties

To access properties related to a graph's data at runtime, you use DataWindow methods for graphs. There are three categories of these methods related to data:

- Methods that provide information about a graph's data
- Methods that save data from a graph
- Methods that change the color, fill patterns, and other visual properties of data

How to use the methods

To call the methods for a graph in a DataWindow control, use the following syntax:

DataWindowName.methodName ("*graphName*", *otherArguments…*)

For example, there is a method CategoryCount, which returns the number of categories in a graph. So to get the category count in the graph gr_printer (which is in the DataWindow control dw_sales), write:

```
Ccount = dw_sales.CategoryCount("gr_printer")
```

Getting information about the data

There are quite a few methods for getting information about data in a graph in a DataWindow control at runtime. For all methods, you provide the name of the graph within the DataWindow as the first argument. You can provide your own name for graph controls when you insert them in the DataWindow painter. If the presentation style is Graph, you do not need to name the graph.

These methods get information about the data and its display. For several of them, an argument is passed by reference to hold the requested information:

Table 12-1: Common methods for graph DataWindows

Method	Information provided
CategoryCount	The number of categories in a graph
CategoryName	The name of a category, given its number
DataCount	The number of data points in a series
FindCategory	The number of a category, given its name
FindSeries	The number of a series, given its name
GetData	The value of a data point, given its series and position (superseded by GetDataValue, which is more flexible)
GetDataPieExplode	The percentage of the pie's radius that the pie slice is to be moved away from the center (exploded)
GetDataStyle	The color, fill pattern, or other visual property of a specified data point
GetDataValue	The value of a data point, given its series and position
GetSeriesStyle	The color, fill pattern, or other visual property of a specified series
SeriesCount	The number of series in a graph
SeriesName	The name of a series, given its number

Saving graph data

The following methods allow you to save data from the graph:

Table 12-2: Methods for saving data from a graph

Method	Action
Clipboard	Copies a bitmap image of the specified graph to the clipboard.
SaveAs	Saves the data in the underlying graph to the clipboard or to a file in one of a number of formats.

Modifying colors, fill patterns, and other data

The methods in Table 12-3 allow you to modify the appearance of data in a graph.

Table 12-3: Methods for modifying the appearance of data

Method	Action
ResetDataColors	Resets the color for a specific data point
SetDataStyle	Sets the color, fill pattern, or other visual property for a specific data point
SetSeriesStyle	Sets the color, fill pattern, or other visual property for a series

Using graph methods

You call the data-access methods after a graph has been created and populated with data. Some graphs, such as graphs that display data for a page or group of data, are destroyed and re-created internally as the user pages through the data. Any changes you made to the display of a graph, such as changing the color of a series, are lost when the graph is re-created.

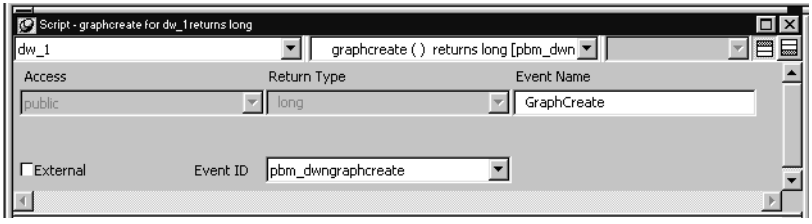
Event for graph creation

To be sure that data-access methods are called whenever a graph has been created and populated with data, you can call the methods in the code for an event that is triggered when a graph is created. The graph-creation event is triggered by the DataWindow control after it has created a graph and populated it with data, but before it has displayed the graph.

By accessing the data in the graph in this event, you are assured that you are accessing the current data and that the data displays the way you want it.

PocketBuilder provides an event ID, pbm_dwngraphcreate, that you can assign to a user event for a DataWindow control.

Figure 12-1: Selecting a user event ID for graph creation



❖ **To access data properties of a graph in a DataWindow control:**

1. Place the DataWindow control in a window or user object and associate it with the DataWindow object containing the graph.

   Next you create a user event for the DataWindow control that is triggered whenever a graph in the control is created or changed.

2. Select Insert>Event from the menu bar.

   The Script view displays and includes prototype fields for adding a new event.

3. Select the DataWindow control in the first drop-down list of the prototype window.

   If the second drop-down list also changes to display an existing DataWindow event prototype, scroll to the top of the list to select New Event or select Insert>Event once again from the menu bar.

4. Name the user event you are creating.

   For example, you might call it GraphCreate.

5. Select pbm_dwngraphcreate for the event ID.

6. Click OK to save the new user event.

7. Write a script for the new GraphCreate event that accesses the data in the graph.

   Calling data access methods in the GraphCreate event assures you that the data access happens each time the graph has been created or changed in the DataWindow.

Examples

The following statement sets to black the foreground (fill) color of the Q1 series in the graph gr_quarter, which is in the DataWindow control dw_report. The statement is in the GraphCreate event, which is associated with the event ID pbm_dwngraphcreate in PocketBuilder:

```
dw_report.SetSeriesStyle("gr_quarter", "Q1", &
       foreground!, 0)
```

The following statement changes the foreground (fill) color to red of the second data point in the Stellar series in the graph gr_sale in a window. The statement can be in a script for any event:

```
int SeriesNum
// Get the number of the series.
SeriesNum = gr_sale.FindSeries("Stellar")

// Change color of second data point to red
gr_sale.SetDataStyle(SeriesNum, 2, foreground!, 255)
```

For more information

For complete information about the data-access graph methods, see the *DataWindow Reference* in the online Help. For more about user events, see the *User's Guide*.

PART 4

Connecting to a Database

This part describes how to connect to a database in the development environment and in an application. It also describes how to use MobiLink synchronization in an application and how to set database parameters and preferences.

CHAPTER 13

Database Connectivity in PocketBuilder

About this chapter

This chapter describes how to access data in PocketBuilder and how to create and manage database profiles.

Contents

Accessing data in PocketBuilder

When you build an application in PocketBuilder, you can access data in a SQL Anywhere database or UltraLite database in the Database painter and the DataWindow painter. You can also write scripts that access these databases in the Application, Window, Menu, and User Object painters.

Proxy tables
SQL Anywhere allows you to create proxy tables that can map to other databases, including databases managed by other database management systems. For information on how to create proxy tables, see the SQL Anywhere documentation. For information on identifying identity columns in the underlying database tables referenced by proxy tables, see the technical note “Techniques for Working with Identity Columns in ASA Proxy Tables” on the Sybase Web site at http://www.sybase.com/detail?id=1035056.

Connecting to a database

To work in the Database painter or build a DataWindow object in the DataWindow painter, you need to create a database connection profile. This chapter describes how to create and manage database profiles and how to use them to connect to a database in PocketBuilder.

The connection profile has different options depending on which database interface you use. Chapter 14, "Using database interfaces," describes the database interfaces provided with PocketBuilder.

In scripts, you use Transaction objects to connect to the database. Chapter 16, "Using Transaction Objects," describes how you connect to a database in a script.

Troubleshooting connections

You can use two tools to trace your database connection in the development environment: the PocketBuilder Database Trace tool and, for ODBC connections, the ODBC Driver Manager Trace tool. Chapter 15, "Troubleshooting Your Connection," describes how to use the trace tools.

Synchronizing data

For many mobile applications, you do not have a constant network connection, so having access to a local database offline provides a way to have the application work with or without a network connection. You can use SQL Anywhere or UltraLite as the local database and synchronize with your back-end database using MobiLink synchronization technology. MobiLink allows the local database to synchronize directly to many enterprise databases including SQL Anywhere itself, Sybase Adaptive Server® Enterprise, Microsoft SQL Server, IBM DB2, and Oracle.

Chapter 17, "Using MobiLink Synchronization," provides an overview of MobiLink synchronization and describes how you can launch synchronization from PocketBuilder applications.

Setting parameters and preferences

Chapter 18, "Setting Additional Connection Parameters," describes how to set database parameters and database preferences in PocketBuilder. Reference information for database parameters and database preferences is in the *Connection Reference*.

About database profiles

PocketBuilder connects to a database when you:

- Open a painter that accesses the database
- Compile or save a PocketBuilder script containing embedded SQL statements (such as a CONNECT statement)

- Run an application that accesses the database
- Invoke a DataWindow control function that accesses the database while executing an application

PocketBuilder uses a database profile to connect to the database you used last when you open a painter that accesses the database. It determines which database you used last and how to connect to it by looking at database profile settings in the registry.

A database profile is a named set of parameters stored in your system registry. Each profile defines a connection to a particular database in the PocketBuilder development environment. You must create a database profile for each data connection. You can:

- Select a database profile to establish or change database connections. You can easily connect to another database at any time during a PocketBuilder session. This is particularly useful if you often switch between different database connections.
- Edit a database profile to modify or supply additional connection parameters.
- Delete a database profile if you no longer need to access that data.
- Import and export profiles.

Because database profiles are created when you define your data and are stored in the registry, they have the following benefits:

- They are always available to you in the development environment
- Connection parameters supplied in a database profile are saved until you edit or delete the database profile

You can also use a database profile to generate the correct syntax to use in a script for a database connection.

Creating database profiles

You work with two dialog boxes when you create a database profile in PocketBuilder: the Database Profiles dialog box and the Database Profile Setup dialog box.

Using the Database painter to create database profiles
You can also create database profiles from the Database painter's Objects view.

Database Profiles dialog box

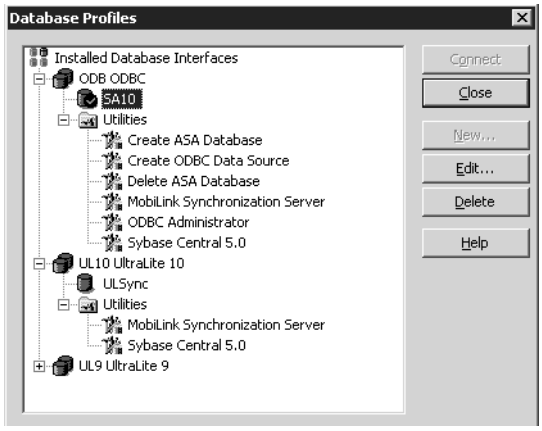
The Database Profiles dialog box uses an easy-to-navigate tree control format to display your defined database profiles. You can create, edit, and delete database profiles from this dialog box.

When you run the PocketBuilder Setup program, the program creates a Vendors key under the PocketBuilder section of the system registry in *HKEY_LOCAL_MACHINE\SOFTWARE*, and adds the following registry string values: ODB for the ODBC interface, UL 10 for the UltraLite 10.x interface, and UL9 for the UltraLite 9.x interface.

Most PocketBuilder examples use a standalone database called *asademo.db* that is installed with Adaptive Server Anywhere 9. The PocketBuilder setup program also installs the ASADemo_10 database in the *Code Examples\SA10DemoData* directory. This database is a migrated version of *asademo.db* that you can use with SQL Anywhere 10. Table and column names in this database are different than the names in the SQL Anywhere 10 Demo database that installs with SQL Anywhere 10.

The Database Profiles dialog box in Figure 13-1 displays profiles for a SQL Anywhere database (SA10) and an UltraLite database (ULSync). The check mark on the SA10 profile indicates that it is currently connected. The figure also shows utilities you can use with these interfaces.

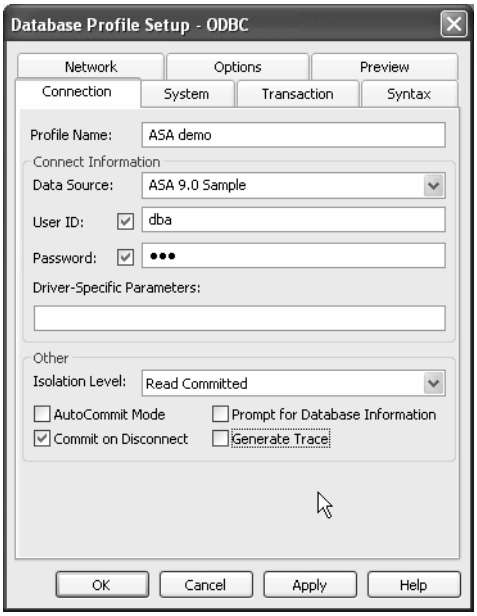
Figure 13-1: The Database Profiles dialog box



Database Profile Setup dialog box

Each interface has a Database Profile Setup dialog box where you can set interface-specific connection parameters. Figure 13-2 shows the dialog box for ODBC.

Figure 13-2: The Database Profile Setup dialog box



The Database Profile Setup dialog box groups similar connection parameters on each tab page and lets you set their values easily by using check boxes, drop-down lists, and text boxes. Basic (required) connection parameters are on the Connection tab page, and additional connection options, including additional database parameters and SQLCA properties, are on the other tab pages.

As you complete the Database Profile Setup dialog box in PocketBuilder, the correct PowerScript connection syntax for each selected option is generated on the Preview tab. You can copy the syntax you want from the Preview tab into a PocketBuilder application script.

Supplying information in the dialog box

You do not need to supply values for all boxes in the Database Profile Setup dialog box. If you supply the profile name and click OK, PocketBuilder displays a series of dialog boxes to prompt you for additional information when you connect to the database. For example, if you type in only the profile name for an ODBC profile, when you attempt to connect, you are prompted for a file or machine data source name.

For more information about creating SQL Anywhere data source files, see "About SQL Anywhere data sources" on page 200.

Creating a database profile

To create a new database profile, complete the Database Profile Setup dialog box.

❖ **To create a database profile:**

1 Click the Database Profile button in the PowerBar.

2 Highlight the database interface you want to use and click New.

Client software and interface must be installed
To display the Database Profile Setup dialog box, you must have the required client software and database interface properly installed and configured.

3 On the Connection tab page, type a name for the profile and supply values for any other basic parameters your interface requires for connection.

4 On the other tab pages, supply values for any additional connection options, such as database parameters and SQLCA properties.

For information about the values you can supply for your connection, click Help in the Database Profile Setup dialog box.

5 (Optional) Click the Preview tab if you want to see the PowerScript connection syntax that PocketBuilder generates for each selected option.

You can copy the PowerScript connection syntax from the Preview tab directly into a PocketBuilder application script. For more information, see "Using the Preview tab to connect in a PocketBuilder application" on page 239.

6 Click OK to save your changes and close the Database Profile Setup dialog box. (To save your changes on a particular tab page *without* closing the dialog box, click Apply.)

The Database Profiles dialog box displays, with the new profile name highlighted under the appropriate interface. The database profile values are saved in the system registry in the key *HKEY_CURRENT_USER\Software\Sybase\PocketBuilder\2.0\DatabaseProfiles\Pocket PB*.

Specifying passwords in database profiles

Your password does *not* display when you specify it in the Database Profile Setup dialog box. However, when PocketBuilder stores the values for this profile in the registry, the actual password *does* display, in encrypted form, in the DatabasePassword or LogPassword field.

Suppressing display in the profile registry entry

To suppress password display in the profile registry entry, do the following when you create a database profile.

❖ **To suppress password display in the profile registry entry:**

1 Select the Prompt For Database Information check box on the Connection tab in the Database Profile Setup dialog box.

This tells PocketBuilder to prompt for any missing information when you select this profile to connect to the database.

2 Leave the Password box blank. Instead, specify the password in the dialog box that displays to prompt you for additional information when you connect to the database.

When you specify the password in response to a prompt instead of in the Database Profile Setup dialog box, the password does not display in the registry entry for this profile.

Connecting to a database

To establish or change a database connection in PocketBuilder in the Database Profiles dialog box, select the database profile for the database you want to access.

❖ **To connect to a database using the Database Profiles dialog box:**

1 Click the Database Profile button in the PowerBar, or select Tools>Database Profile from the PowerBar.

2 Click the plus sign (+) to the left of the database interface you want to use or double-click its name.

The list expands to display the database profiles defined for the interface.

3 Select the name of the database profile you want to access and click Connect.

PocketBuilder connects to the specified database and closes the dialog box.

Using the Database painter to select a database profile
You can also select the database profile for the database you want to access from the Database painter's Objects view and select Connect from its pop-up menu. However, this method uses more system resources than using the Database Profiles dialog box.

What happens when you connect

When you connect to a database by selecting its database profile, PocketBuilder writes the profile name to the *MRUProfile* (for "most recently used" profile) string value in the following registry key: *HKEY_CURRENT_USER\Software\Sybase\PocketBuilder\2.0\DatabaseProfiles\Pocket PB.*

PocketBuilder also adds the profile to a list of recently-used profiles for the current workspace under the *Workspace* subkey. These profiles display in the Recent Connections list in the File menu for the workspace.

Each time you connect to a different database, PocketBuilder overwrites the MRUProfile value in the registry with the name for the new database connection.

When you open a painter that accesses the database, you are connected to the database you used last. PocketBuilder determines which database this is by reading the registry.

Importing and exporting database profiles

Each database interface provides an Import Profile(s) and an Export Profile(s) option. You can use the Import option to import a previously defined profile for use with an installed database interface. Conversely, you can use the Export option to export a defined profile for use by another user.

The ability to import and export profiles provides a way to move profiles easily between developers. It also means you no longer have to maintain a shared file to maintain profiles, which is important if you cannot rely on connecting to a network to share a file.

When you migrate to a new version of PocketBuilder, you can export the profiles from the previous version and import them into the new version.

❖ **To import a profile:**

1 Highlight a database interface and select Import Profile(s) from the pop-up menu. (In the Database painter, select Import Profile(s) from the File or pop-up menu.)

2 From the Select Profile File dialog box, select the file whose profiles you want to import and click Save.

3 Select the profile(s) you want to import from the Import Profile(s) dialog box and click OK.

The profiles are copied into your registry. If a profile with the same name already exists, you are asked if you want to overwrite it.

❖ **To export a profile:**

1 Highlight a database interface and select Export Profile(s) from the pop-up menu. (In the Database painter, select Export Profile(s) from the File or pop-up menu.)

2 Select the profile(s) you want to export from the Export Profile(s) dialog box and click OK.

The Export Profile(s) dialog box lists all profiles defined in your registry.

3 From the Select Profile File dialog box, select a directory and a file in which to save the exported profile(s) and click Save.

The exported profiles can be saved to a new or existing file. If saved to an existing file, the profile(s) are added to the existing profiles. If a profile with the same name already exists, you are asked if you want to overwrite it.

Maintaining database profiles

You can edit a database profile to change one or more of its connection parameters, and you can delete it when you no longer need to access its data. You can also change a profile using either the Database Profile dialog box or the Database painter.

When you edit a database profile, PocketBuilder updates the database profile entry in the registry.

When you delete a database profile that connects to an ODBC data source, PocketBuilder deletes the database profile entry in the registry, but it does *not* delete the corresponding data source definition from the ODBC.INI registry key. This lets you recreate the database profile later if necessary without having to redefine the data source.

Sharing database profiles

When you work in PocketBuilder, you can share database profiles among users. This section describes what you need to know to set up, use, and maintain shared database profiles in PocketBuilder.

About shared database profiles

You can share database profiles in the PocketBuilder development environment by specifying the location of a file containing the profiles you want to share. You specify this location in the Database Preferences dialog box in the Database painter.

To share database profiles among all PocketBuilder users at your site, store a profile file on a network file server accessible to all users. This shared profile file contains two profiles:

```
[DBMS_PROFILES]
Profiles=Sales, Orders
[Profile Sales]
DBMS=ODBC
DBParm=ConnectString='DSN=Sales;UID=dba;PWD=sql'
Prompt=FALSE
AutoCommit=FALSE
[Profile Orders]
DBMS=ODBC
DBParm=ConnectString='DSN=Orders;UID=;PWD=sql'
Prompt=FALSE
AutoCommit=FALSE
```

When you share database profiles, PocketBuilder displays shared database profiles from the file you specify as well as those from your registry.

Shared database profiles are read-only. You can select a shared profile to connect to a database, but you *cannot* edit, save, or delete profiles that are shared. You can, however, make changes to a shared profile and save it on your computer, as described in "Making a local copy of a shared database profile" on page 194.

Setting up shared database profiles

To set up shared database profiles in PocketBuilder, you specify the location of the file containing shared profiles in the Database painter's Database Preferences dialog box.

❖ **To set up shared database profiles:**

1 In the Database painter, select Design>Options from the menu bar.

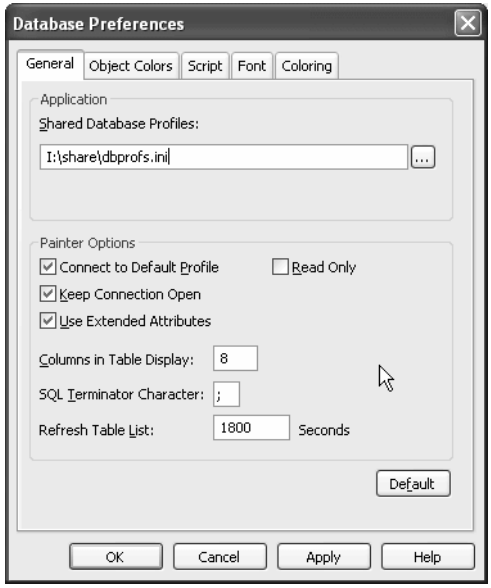
The Database Preferences dialog box displays. If necessary, click the General tab to display the General property page.

2 In the Shared Database Profiles box, specify the location of the file containing the database profiles you want to share. Do this in either of the following ways:

- Type the location (path name) in the Shared Database Profiles box
- Click the Browse button to navigate to the file location and display it in the Shared Database Profiles box

In the following example, *I:\SHARE\dbprofs.ini* is the name and location of the file containing the database profiles to be shared:

Figure 13-3: Specifying shared database profiles

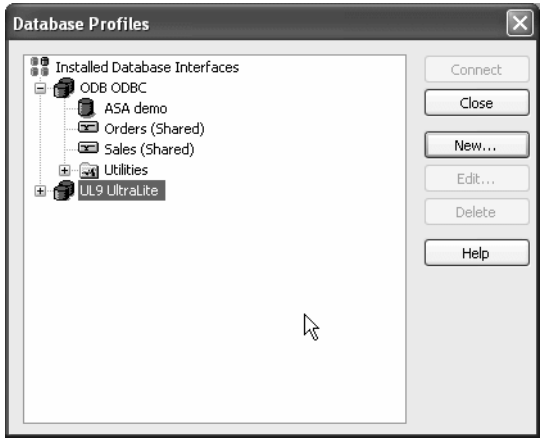


3 Click OK to apply the Shared Database Profiles setting to the current connection and all future connections and close the Database Preferences dialog box.

   PocketBuilder saves your Shared Database Profiles setting in the registry.

You select a shared database profile to connect to a database the same way you select a local profile. The Database Profiles dialog box lists both shared and local profiles. Shared profiles are denoted by a network icon and the word *(Shared)*.

Figure 13-4: Shared database profiles in the Database Profiles dialog box



Making a local copy of a shared database profile

Because shared database profiles can be accessed by multiple users running PocketBuilder, you cannot make changes to these profiles. However, you can modify and save a copy of a shared database profile *for your own use*.

Select the profile in the Database Profiles dialog box, click Edit, and modify the profile. When you click OK to save your changes, PocketBuilder pops up a response window asking if you want to save a copy of the profile to your local machine. When you click OK, PocketBuilder saves the modified copy in your computer's registry. Both profiles display in the Database Profiles dialog box.

Maintaining shared database profiles

If you maintain the database profiles for PocketBuilder at your site, you might need to update shared database profiles from time to time and make these changes available to your users.

Because shared database profiles can be accessed by multiple users running PocketBuilder, it is *not* a good idea to make changes to the profiles over a network. Instead, you should make any changes locally and then provide the updated profiles to your users.

❖ **To maintain shared database profiles at your site:**

1. Make and save required changes to the shared profiles on your own computer. These changes are saved in your registry.
2. Export the updated profile entries from your registry to the existing file containing shared profiles.

   For instructions, see "Importing and exporting database profiles" next.

3 If they have not already done so, have users specify the location of the new profiles file in the Database Preferences dialog box so that they can access the updated shared profiles on their computers.

CHAPTER 14

Using database interfaces

About this chapter

This chapter describes the database interfaces provided by PocketBuilder.

Contents

About database interfaces

There are two ways to access data in the PocketBuilder development environment:

- Through a standard database interface
- Through a native database interface

Standard database interfaces

A standard database interface communicates with a database through a standard-compliant driver or data provider. The standard-compliant driver or data provider translates the abstract function calls defined by the standard's API into calls that are understood by a specific database. To use a standard interface, you need to install the standard's API and a suitable driver or data provider. Then, install the standard database interface you want to use to access your DBMS by selecting the interface in the PocketBuilder Setup program.

PocketBuilder currently supports the Open Database Connectivity (ODBC) standard driver.

Native database interfaces

A native database interface communicates with a database through a direct connection, using that database's native API and a PocketBuilder database interface DLL for that specific database.

PocketBuilder currently provides a native interface to UltraLite 9.x databases through the file *pkul920.dll*. This file is installed by default when you install PocketBuilder.

Loading database interface libraries

PocketBuilder loads the libraries used by a database interface when it connects to the database. PocketBuilder does *not* automatically free the database interface libraries when it disconnects.

Although memory use is somewhat increased by this technique (since the loaded database interface libraries continue to be held in memory), the technique improves performance and eliminates problems associated with the freeing and subsequent reloading of libraries experienced by some database connections.

If you want PocketBuilder to free database interface libraries on disconnecting from the database, you can change its default behavior:

To change the default behavior for	Do this
Connections in the development environment	Select the Free Database Driver Libraries On Disconnect check box on the General tab of the System Options dialog box.
Runtime connections	Set the FreeDBLibraries property of the Application object to TRUE on the General tab of the Properties view in the Application painter or in a script.

Working with the ODBC database interface

PocketBuilder is closely integrated with Sybase SQL Anywhere, which uses the ODBC database interface. This section contains the following topics:

- Connecting to a SQL Anywhere database on Windows CE
- About SQL Anywhere data sources
- Defining the SQL Anywhere data source
- Defining multiple data sources for the same data
- How PocketBuilder accesses the data source
- Support for Transact-SQL special timestamp columns
- The PKODB20 initialization file
- Preparing remote databases
- Starting SQL Anywhere on a device

Connecting to a SQL Anywhere database on Windows CE

In the development environment, the ODBC driver manager provides an interface between the PocketBuilder ODB interface (*pkodb20.dll*) and the Adaptive Server Anywhere 9 ODBC driver (*dbodbc9.dll*) or the SQL Anywhere 10 ODBC driver (*dbodbc10.dll*). The driver manager can handle three types of data source name (DSN) files: system DSNs, user DSNs, and file DSNs. You can create a database connection profile using any of these DSN types.

In applications that you deploy to Windows CE, you must use a file DSN because there is no ODBC driver manager.

Using file DSNs

Windows CE does not provide an ODBC driver manager or an ODBC Administrator. On Windows CE, SQL Anywhere uses ODBC data sources stored in ANSI format files. A file DSN has the same name as the data source, with the extension *.dsn*, and is usually stored at the root level of the device.

Windows CE also searches for data source files in the following locations:

- The directory from which the ODBC driver (*dbodbc9.dll* or *dbodbc10*) was loaded. This is usually the *Windows* directory.
- The directory specified in the Location key of the Adaptive Server Anywhere or SQL Anywhere section of the registry. This is usually the same as the SQL Anywhere installation directory. The default installation directory is: *\Program Files\Sybase\ASA* or *\Program Files\SQLAny10*.

You can specify either the DSN or the FILEDSN keyword to use file data source definitions in a script. On Windows CE, DSN and FILEDSN are synonyms.

The data source typically specifies the location of the database and the database engine. For more information about defining data sources, see "About SQL Anywhere data sources" on page 200.

Using a different SQL Anywhere ODBC driver

The default ODBC driver supplied with Adaptive Server Anywhere version 9 is *dbodbc9.dll*. You can specify a different ODBC driver by including "`driver=dbodbc`***`X`***`.dll`" in the ConnectString parameter in the DBParm value or in the file data source, where *X* is the version number of the SQL Anywhere ODBC driver.

For example, to use a SQL Anywhere 10 ODBC driver on the Windows CE device, you can set the following DBParm value in a script:

```
SQLCA.DBPARM="ConnectString='DSN=myDSN;driver=dbodbc10.dll;UID=dba;PWD=sql'"
CONNECT using SQLCA;
```

SQLCA is the default connection object.

If you are running your PocketBuilder applications from the desktop, use the actual name of the driver (for example, "SQL Anywhere 10") in the ConnectString parameter rather than the name of the DLL. Otherwise, the driver you select in the connection string might be ignored.

For more information about setting SQLCA parameters, see "Assigning values to the Transaction object" on page 237.

The DSN you assign must exist in the root directory on the Windows CE device or emulator, or in the *\Windows* directory or the directory from which the server was started. You can include the driver name in the DSN instead of the DBParm by adding an assignment for the driver in the DSN file:

```
[ODBC]
enginename=asademo
databasename=asademo
databasefile=\Program Files\Sybase\ASA\asademo.db
start=\Program Files\Sybase\ASA\dbsrv9.exe
Driver=dbodbc9.dll
```

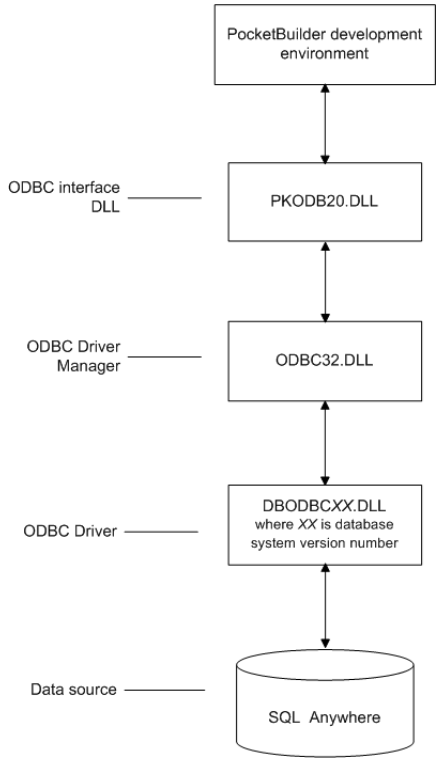
About SQL Anywhere data sources

SQL Anywhere includes two database servers—a personal database server that can be used on a single desktop, and a network database server that supports communications across a network. The network database server, *dbsrvXX.exe* is always used on Windows CE systems, where *XX* is the version of the SQL Anywhere database system. (The personal database server uses the *dbengXX.exe* engine.)

Basic software components for SQL Anywhere

Figure 14-1 show the basic software components required to connect to a SQL Anywhere data source in PocketBuilder.

Figure 14-1: Components of a SQL Anywhere connection



The PocketBuilder ODB interface (*pkodb20.dll*) calls ODBC functions to submit SQL statements, to catalog requests, and to retrieve results from a data source.

The Microsoft ODBC driver manager (*odbc32.dll*) installs, loads, and unloads drivers for an application.

No ODBC driver manager on Windows CE
There is no ODBC driver manager on Windows CE. For more information, see “Connecting to a SQL Anywhere database on Windows CE” on page 199.

The SQL Anywhere ODBC driver (*dbodbc9.dll* or *dbodbc10.dll*) processes ODBC function calls, submits SQL requests to a particular data source, and returns results to an application. The driver also translates an application’s request so that it conforms to the SQL syntax supported by the SQL Anywhere database.

The SQL Anywhere data source stores and manages data for an application.

Preparing to use the data source

Before you define and connect to a SQL Anywhere data source in PocketBuilder, follow these steps to prepare the data source.

❖ **To prepare a SQL Anywhere data source:**

1 Make sure the database file for the SQL Anywhere data source exists.

You can create a new database using SQL Anywhere outside of PocketBuilder, or by launching the Create ASA Database utility from the Utilities folder in the Database Profiles dialog box. For more information, see the chapter on managing databases in the *User's Guide*.

You can also convert an enterprise database for use with PocketBuilder. For more information, see "Preparing remote databases" on page 211.

2 Make sure you have the log file associated with the SQL Anywhere database so that you can fully recover the database if it becomes corrupted.

If the log file for the SQL Anywhere database does not exist, the SQL Anywhere database engine creates it. However, if you are copying or moving a database from another computer or directory, you should copy or move the log file with it.

Defining the SQL Anywhere data source

You can define user, system, or file data sources. If you want to use the data source on a Pocket PC, define a file data source. Windows CE supports only file data sources.

When you create a local SQL Anywhere database in the Database painter, PocketBuilder automatically creates a data source definition and database profile for you. Therefore, you do not need to define a user data source for use on the desktop, but you do need to define a file data source if you want to deploy the database to a remote device.

Defining file data sources

You can use the ODBC Data Source Administrator utility to create a file data source, but in general it is easier to use a text editor. The file should be saved with ANSI encoding.

The following example shows the format of the file. The database name and engine name are usually the same as the name of the database file without the *.db* extension. In the example, the database file is *MyRemoteDB.db*, and on the device it will be installed in the same directory as the application, *\Program Files\AcmeTools*:

```
[ODBC]
uid=dba
pwd=sql
enginename=MyRemoteDB
databasename=MyRemoteDB
databasefile=\Program Files\AcmeTools\MyRemoteDB.db
start=\Program Files\Sybase\ASA\dbsrv9.exe
```

Defining system and user data sources

Use the following procedure to define a system or user data source for use on a desktop or server:

❖ **To define a data source for the SQL Anywhere driver:**

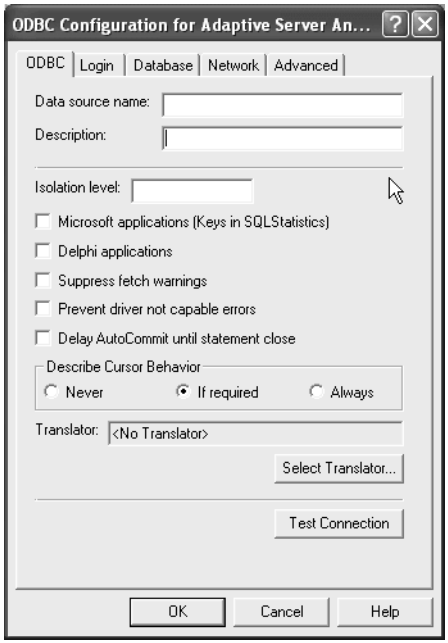
1. Launch the ODBC Data Source Administrator utility. From the User or System DSN tab page, click the Add button.

   The Create New Data Source dialog box displays.

2. Select a driver for the version of SQL Anywhere or Adaptive Server Anywhere that your database uses and click Finish.

   The ODBC configuration dialog box displays for the database version you selected.

Figure 14-2: The ODBC Configuration dialog box



3 You must supply the following values:

- Data source name on the ODBC page
- User ID and password on the Login page
- Database file on the Database page

Use the Help button to get information about boxes in the dialog box.

Using the Browse button

When you use the Browse button to supply the Database File name (for example, the SQL Anywhere sample database demo.db), this name also creates entries in both the Data Source Name and Database Name boxes. This might change values you previously supplied in these boxes.

If you want to specify a different name for the data source or database, you can edit one or both of these boxes *after* using the Browse button.

4 Click OK to save the data source definition.

Specifying a Start Line value

When the SQL Anywhere ODBC driver cannot find a running database server using the PATH variable and Database Name setting, it uses the commands specified in the Start Line field to start the database server.

If you are creating a DSN for use on a Pocket PC device, specify the name of the network database server and its location on the device, for example: *\Program Files\SQLAnyPath\dbsrv\XX.exe*, where *SQLAnyPath* is typically *SQLAny10* for SQL Anywhere 10 and *Sybase\ASA* for Adaptive Server Anywhere 9, and *XX* is the version of the database system.

If you are creating a DSN for use in the development environment only, you can specify the name and location of the personal database server. The default location is *C:\Program Files\Sybase\SQL Anywhere XX\win32\dbengXX.exe*.

Preventing the SQL Anywhere log screen from displaying

You can add a -q or -qw switch to the start line to prevent the SQL Anywhere log screen from displaying when you connect to the database. The start line in a file DSN for use on a Pocket PC device would look like this:

```
start=\Program Files\SQLAny10\dbsrv10.exe -qw
```

The start line for Adaptive Server Anywhere 9 might look like this:

```
start=\Program Files\Sybase\ASA\dbsrv9.exe -q
```

Because the connection might take a few moments, you might want to call the SetPointer function to display the Windows CE version of the hourglass icon when using the -q or -qw switch. On Windows CE, you need to explicitly call the SetPointer function to reset the default pointer when the script completes.

For more information on completing the ODBC Configuration For SQL Anywhere dialog box, see the *SQL Anywhere Server Database Administration* book in the SQL Anywhere documentation set.

Defining multiple data sources for the same data

When you define an ODBC data source in PocketBuilder, each data source name must be unique. You can, however, define multiple data sources that access the same data, as long as the data sources have unique names.

For example, assume that your data source is a SQL Anywhere database, *C:\SQLAny\SALES.DB*. Depending on your application, you might want to specify different sets of connection parameters for accessing the database, such as different passwords and user IDs.

To do this, you can define two ODBC data sources named Sales1 and Sales2 that specify the same database (*C:\SQLAny\SALES.DB*) but use different user IDs and passwords. When you connect to the data source using a profile created for either of these data sources, you are using different connection parameters to access the same data.

How PocketBuilder accesses the data source

When you access an ODBC data source in the PocketBuilder development environment, there are several initialization files and registry entries on your computer that work with the ODBC interface and driver to make the connection.

PKODB20 initialization file

The *PKODB20* initialization file maintains access to extended functionality in the back-end DBMS, for which ODBC does not provide an API call. Examples of extended functionality are SQL syntax or DBMS-specific function calls.

In most cases, you do not need to edit the PKODB20 initialization file. In certain situations, however, you may need to add functions to the PKODB20 initialization file for your DBMS.

For instructions, see “The PKODB20 initialization file” on page 209.

ODBCINST registry entries

The ODBCINST initialization information is located in the *HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI* registry key. When you install an ODBC-compliant driver, *ODBCINST.INI* is automatically updated with a description of the driver.

This description includes:

- The DBMS or data source associated with the driver
- The drive and directory of the driver and setup DLLs (for some data sources, the driver and setup DLLs are the same)
- Other driver-specific connection parameters

You do *not* need to edit the registry key directly to modify connection information.The key is automatically updated when you install the driver.

ODBC registry entries

ODBC initialization information is located in the *HKEY_CURRENT_USER\SOFTWARE\ODBC\ODBC.INI* registry key. When you define a data source, the driver writes the values you specify in the ODBC setup dialog box to the *ODBC.INI* registry key.

The *ODBC.INI* key contains subkeys named for each defined data source. Each subkey contains the values specified for that data source in the ODBC setup dialog box. The values include the following:

- Database file
- Driver
- Optional description
- Connection parameters

Do *not* edit the *ODBC* subkey directly to modify connection information. Instead, use a tool designed to define ODBC data sources and the ODBC configuration automatically, such as the ODBC Data Source Administrator.

Database profiles registry entry

Database profiles for all data sources are stored in the registry in *HKEY_CURRENT_USER\SOFTWARE\Sybase\PocketBuilder\2.0\DatabaseProfiles*.

You should *not* need to edit the profiles directly to modify connection information. These files are updated automatically when PocketBuilder creates the database profile as part of the ODBC data source definition.

You can also edit the profile in the Database Profile Setup dialog box or complete the Database Preferences dialog box in PocketBuilder to specify other connection parameters stored in the registry. (For instructions, see Chapter 18, “Setting Additional Connection Parameters.”)

The following example shows a portion of a database profile for the SQLAny Demo data source:

```
DBMS=ODBC
Database=SQL Anywhere Demo DB
UserId=dba
DatabasePassword=
LogPassword=
ServerName=
LogId=
Lock=
```

```
DbParm=ConnectString='DSN=SQLAny Demo;UID=dba;PWD=sql'
Prompt=0
```

This registry entry example shows the two most important values in a database profile for an ODBC data source:

- **DBMS** The DBMS value (ODBC) indicates that you are using the ODBC interface to connect to the data source.
- **DBParm** The ConnectString DBParm parameter controls your ODBC data source connection. The connect string *must* specify the DSN (data source name) value, which tells ODBC which data source you want to access. When you select a database profile to connect to a data source, ODBC looks in the *ODBC.INI* registry key for a subkey that corresponds to the data source name in your profile. ODBC then uses the information in the subkey to load the required libraries to connect to the data source. The connect string can also contain the UID (user ID) and PWD (password) values needed to access the data source.

Support for Transact-SQL special timestamp columns

When you work with a SQL Anywhere table in the Database or DataWindow painter, the default behavior is to treat any column named timestamp as a SQL Anywhere Transact-SQL special timestamp column.

Creating special timestamp columns

You can create a Transact-SQL special timestamp column in a SQL Anywhere table.

❖ **To create a Transact-SQL special timestamp column in a SQL Anywhere table in PocketBuilder:**

1. Give the name timestamp to any column having a timestamp datatype that you want treated as a Transact-SQL special timestamp column. Do this in one of the following ways:
   - In the painter – select timestamp as the column name. (For instructions, see the *User's Guide*.)
   - In a SQL CREATE TABLE statement – follow the "CREATE TABLE example" next.

2 Specify `timestamp` as the default value for the column. Do this in one of the following ways:

- In the painter – select `timestamp` as the default value for the column. (For instructions, see the *User's Guide*.)
- In a SQL CREATE TABLE statement – follow the "CREATE TABLE example" next.

CREATE TABLE example

The following CREATE TABLE statement defines a SQL Anywhere table named timesheet containing three columns: employee_ID (integer datatype), hours (decimal datatype), and timestamp (timestamp datatype and timestamp default value):

```
CREATE TABLE timesheet (
   employee_ID INTEGER,
   hours DECIMAL,
   timestamp TIMESTAMP default timestamp )
```

Not using special timestamp columns

If you want to change the default behavior, you can specify that PocketBuilder *not* treat SQL Anywhere columns named timestamp as Transact-SQL special timestamp columns.

❖ **To specify that PocketBuilder *not* treat columns named *timestamp* as a Transact-SQL special timestamp column:**

- Edit the Adaptive Server Anywhere section of the PKODB20 initialization file to change the value of SQLSrvrTSName from `'Yes'` to `'No'`.

  After making changes in the initialization file, you must reconnect to the database to have them take effect. See "Adding functions to the PKODB20 initialization file" on page 210.

The PKODB20 initialization file

The name of the PKODB20 initialization file is *PKODB20.INI*.

Function of the PKODB20 initialization file

When you access data through the ODBC interface, PocketBuilder uses the PKODB20 initialization file to maintain access to extended functionality in the back-end DBMS, for which ODBC does not provide an API call. Examples of extended functionality are SQL syntax or function calls specific to a particular DBMS.

Editing the PKODB20 initialization file

In most cases, you do *not* need to modify the PKODB20 initialization file. Change the PKODB20 initialization file only if you are asked to do so by a Technical Support representative. Changes to this file can adversely affect PocketBuilder.

However, you *can* edit the PKODB20 initialization file if you need to add functions for your back-end DBMS.

If you modify the PKODB20 initialization file, first make a copy of the existing file, then keep a record of all changes you make. If you call Technical Support after modifying the PKODB20 initialization file, tell the representative that you changed the file and describe the changes you made.

Adding functions to the PKODB20 initialization file

The PKODB20 initialization file lists the functions for SQL Anywhere. If you need to add a function to the PKODB20 initialization file for use with SQL Anywhere, add the function to the ASA_FUNCTIONS section.

❖ **To add functions to an existing section in the PKODB20 initialization file:**

1 Open the PKODB20 initialization file in one of the following ways:

- Use the File editor in PocketBuilder. (For instructions, see the *User's Guide*.)
- Use any text editor outside PocketBuilder.

2 Find the Functions section for SQL Anywhere (ASA_FUNCTIONS):

```
;*******************************************
;Functions
;*******************************************
[ASA_FUNCTIONS]
AggrFuncs=avg(x),avg(distinct x),count(x),
    count(distinct x),count(*),list(x),
    list(distinct x),max(x),max(distinct x),
    min(x),min(distinct x),sum(x),sum(distinct x)
Functions=abs(x),acos(x),asin(x),atan(x),
    atan2(x,y),ceiling(x),cos(x),cot(x),degrees(x),
    exp(x),floor(x),log(x),log10(x),
    mod(dividend,divisor),pi(*),power(x,y),
    radians(x),rand(),rand(x),
    remainder(dividend,divisor),round(x,y),
    sign(x),sin(x),sqrt(x),tan(x),
    "truncate"(x,y),ascii(x),byte_length(x),
    byte_substr(x,y,z),char(x),char_length(x),
    charindex(x,y),difference(x,y)insertstr(x,y,z),
    lcase(x),left(x,y),length(x), locate(x,y,z),
    lower(x),ltrim(x),patindex('x',y),repeat(x,y),
    replicate(x,y),right(x,y),rtrim(x),
    similar(x,y),soundex(x),space(x),str(x,y,z),
    string(x,...),stuff(w,x,y,z),substr(x,y,z),
    trim(x),ucase(x),upper(x),date(x),
    dateformat(x,y),datename(x,y),day(x),
    dayname(x),days(x),dow(x),hour(x),hours(x),
```

```
minute(x),minutes(x),minutes(x,y),month(x),
monthname(x),months(x),months(x,y),now(*),
quarter(x),second(x),seconds(x),seconds(x,y),
today(*),weeks(x),weeks(x,y),year(x),years(x),
years(x,y),ymd(x,y,z),dateadd(x,y,z),
datediff(x,y,z),datename(x,y),datepart(x,y),
getdate(),cast(x as y),convert(x,y,z),
hextoint(x),inttohex(x),
connection_property(x,...),datalength(x),
db_id(x),db_name(x),db_property(x),
next_connection(x),next_database(x),
property(x),property_name(x),
property_number(x),property_description(x),
argn(x,y,...),coalesce(x,...),
estimate(x,y,z),estimate_source(x,y,z),
experience_estimate(x,y,z),ifnull(x,y,z),
index_estimate(x,y,z),isnull(x,...),
number(*),plan(x),traceback(*)
```

3 To add a new function, type a comma followed by the function name at the end of the appropriate function list, as follows:

- Aggregate functions – add aggregate functions to the end of the AggrFuncs list.
- All other functions – add all other functions to the end of the Functions list.

Case sensitivity
If the back-end DBMS you are using is case sensitive, be sure to use the required case when you add the function name.

4 Save your changes to the PKODB20 initialization file.

Preparing remote databases

When you prepare a SQL Anywhere database to be used as a remote database on a device, you usually define a subset of the enterprise database that is relevant to the needs of the mobile application. There are several options available for building a SQL Anywhere database from an enterprise database:

- Use Sybase Central
- Use PowerDesigner
- Use tools that come with your enterprise database

After you have built the SQL Anywhere database, you can copy it to the device using Microsoft ActiveSync. You also need to create a DSN file and copy it to the root directory on the device. For more information, see "About SQL Anywhere data sources" on page 200.

Sybase Central

Sybase Central is a utility that allows developers to build a SQL Anywhere database from another database management system. (Sybase Central can be installed from the SQL Anywhere installation program.) For example, these are the steps needed to start with an Oracle database and build a SQL Anywhere database and use it with PocketBuilder:

1. Open Sybase Central.
2. Connect to an Oracle database.
3. Migrate required tables to SQL Anywhere.
4. Use MobiLink scripting capabilities to generate initial scripts.
5. Use the PocketBuilder Database painter to view the database schema.
6. Develop an application using the SQL Anywhere database in PocketBuilder.
7. Deploy the application to the Windows CE device or emulator.

PowerDesigner

PowerDesigner Physical Data Model allows you to reverse-engineer enterprise databases and create corresponding SQL Anywhere databases. It also gives you the ability to manipulate the database schema. PowerDesigner Physical Data Model is provided with the full version of SQL Anywhere Studio. It is not provided with PocketBuilder.

Tools that come with your enterprise database

You can use the tools that come with your enterprise database to view the database schema. You can then create the SQL Anywhere database in the tool of your choice, such as Sybase Central, PowerDesigner, or a command-line tool.

For information about preparing remote databases for use with MobiLink, see Chapter 17, "Using MobiLink Synchronization."

Starting SQL Anywhere on a device

To start a SQL Anywhere database on a Pocket PC device or emulator, use the File Explorer (in the Programs folder) to navigate to the location where SQL Anywhere is installed, typically *Program Files\SQLAny10* or *Program Files\Sybase\ASA*. Tap dbsrv10 or dbsrv9, use the drop-down lists or the Soft Input Panel to complete the Server Startup Options dialog box, and tap OK.

Your application can start the database automatically by specifying the properties of a Transaction object and issuing a CONNECT statement. For more information, see Chapter 16, "Using Transaction Objects."

Working with the UltraLite database interface

This section describes how to use the native UltraLite database interface with PocketBuilder. UltraLite is a deployment technology for SQL Anywhere databases that allows applications on small devices to use full-featured SQL to accomplish data storage, retrieval, and manipulation.

UltraLite supports referential integrity, transaction processing, multi-table joins of all varieties, and most of the same datatypes, runtime functions, and SQL data manipulation features as SQL Anywhere. It provides an ultra-small footprint by generating a custom database engine for your application that includes only the features required by your application.

PocketBuilder supports UltraLite 9.x and UltraLite 10.x. The PocketBuilder UL9 (*pkul920.dll*) and UL10 interfaces (*pkul1020.dll*) use the UltraLite C++ Component API to implement communication with the database.

Supported UltraLite datatypes

The UltraLite interface supports the following UltraLite datatypes:

BINARY	S_BIG
BIT	S_LONG
CHAR	S_SHORT
DATE	TIME
DOUBLE	TIMESTAMP
LONGBINARY	TINY
LONGVARCHAR	U_BIG
NUMERIC	U_LONG
REAL	U_SHORT

Running utilities for UltraLite databases

Table 14-1 lists the database utilities you can access from the Database painter and the Database Profiles dialog box in PocketBuilder.

Table 14-1: Utilities you can use with UltraLite databases

Utility	Purpose
Create UltraLite Database	Provides a user interface to the *ulconv.exe* command to create UltraLite databases
MobiLink Synchronization Server	Allows you to launch a MobiLink server with options for automatic script generation, automatic addition of users, and generation of diagnostic log files
Sybase Central	Allows you to connect to a consolidated database to add publications, users, script versions, and synchronized tables
UltraLite Schema Painter	Starts the schema painter that lets you generate a USM file containing information about tables and publications to be synchronized

Defining the UltraLite database interface

To define a connection through the UltraLite database interface, you must create a database profile by supplying values for at least the basic connection parameters in the Database Profile Setup dialog box for UltraLite. You can then select this profile at any time to connect to the UltraLite database. For more information, see "Creating database profiles" on page 183.

Specifying the database file

Each UltraLite application has its own database, which is held in a file with the extension *.UDB*. Each database contains a schema that includes information about the database's tables, indexes, column names, datatypes, primary and foreign keys, and other metadata. The schema for an UltraLite database is stored in a compact form.

You can create an UltraLite database in the Database painter. For more information, see the chapter on managing databases in the *User's Guide*. You can also create an UltraLite database using a reference SQL Anywhere database.

When you browse to select a database file in the Database Profile Setup dialog box, PocketBuilder adds a DBF value to the ConnectString database parameter.

Connecting to an encrypted UltraLite database

You can choose to specify the encryption key for an encrypted database by typing it in the Additional ConnectionString Parameters box on the general page of the Database Profile Setup dialog box for UltraLite. Use the following syntax:

KEY=*encryption_key*

However, this practice is not recommended. If you leave the Additional ConnectionString Parameters box blank, you are prompted to enter the key when you connect.

In your applications, you can code the encryption key in the ConnectString value when you specify the DBParm property of the transaction object, but to ensure that only authorized users can access the database, provide a dialog box so that users can enter the key at runtime.

Setting Autocommit mode

In the development environment, PocketBuilder always behaves as if AutoCommit is set to true, whether you check the AutoCommit mode check box or not. The setting in the Database Profile Setup dialog box affects the behavior only when you run or debug the application. For more information, see the description of the AutoCommit database preference in the *Connection Reference*.

Adding multiple users

PocketBuilder allows you to add or edit multiple users of UltraLite databases. UltraLite permits a maximum of four user IDs per database. For UltraLite database profiles, you can use the pop-up menu for the new Users item in the Objects view of the Database painter to add or edit users, and to delete users.

When you select Add or Edit User from the User item pop-up menu for an UltraLite database, or when you select Delete User, you should already know which user IDs exist in the database. If you select either of these menu items, the Properties view displays. The Properties view has three text boxes (User, New Password, and Confirm New Password) when you select Add or Edit User; it has a single text box (UltraLite User to Delete) when you select Delete User.

After you save changes in the Database painter involving user maintenance for an UltraLite database, the Output view displays a message with the type of change you made.

Migrating a SQL Anywhere application to UltraLite

Applications that you create with a SQL Anywhere connection can be modified to use an UltraLite connection. This section describes the differences between the two databases that require you to make some changes in your applications before you can use them with an UltraLite database.

Table owners

UltraLite does not support the concept of a table owner, therefore all table names must be unique in the database, and any user who successfully signs on to the UltraLite database has full access and update rights to all tables.

UltraLite does not support SQL statements that include owner-qualified table names, but the *pkul920.dll* and *pkul1020.dll* interfaces automatically process fully-qualified SQL statements at runtime to remove the table owner qualification before passing the statement to the UltraLite SQL parser. However, this does have implications for DataWindow objects that you create while connected to the UltraLite driver. The SQL SELECT statement that is generated contains unqualified SQL. This can limit the portability of a DataWindow object if you want to reuse it in an application that connects to a SQL Anywhere database using ODBC.

Views

UltraLite does not support views. You must rework existing applications that reference views so that they reference base tables instead.

Cursors

If your application uses the PowerScript DECLARE Cursor SQL statement, you might need to modify the application for connections to an UltraLite database. The *pkul920.dll* and *pkul1020.dll* interfaces use the established connection to implement the cursor, which means that you cannot nest one cursor within another, and you cannot issue other SQL statements, other than FETCH Cursor statements, while the cursor is open.

As an alternative, you can instantiate a DataStore that retrieves the same result set that the cursor returns, and rewrite your application to scroll through the DataStore instead of performing a FETCH loop through the cursor.

Stored procedures

UltraLite does not support stored procedures. You must modify your application if it uses stored procedures in any of the following ways that are supported by SQL Anywhere but not by UltraLite:

- As the data source for a DataWindow object
- In DataWindow update procedures
- In PowerScript DECLARE Procedure SQL statements
- In declarations of stored procedures as remote procedure calls using the RPCFUNC {ALIAS FOR "*spname*" } syntax

Database access limits

If two or more PocketBuilder applications are running concurrently, only one application can successfully connect to a specific UltraLite database at a time. Whichever application connects first retains exclusive use of that UltraLite database until the application closes all connections to the database.

A single PocketBuilder application can obtain two or more simultaneous connections to the same UltraLite database using different transaction objects. A single PocketBuilder application can also connect to two or more UltraLite databases concurrently by instantiating multiple transaction objects.

CHAPTER 15

Troubleshooting Your Connection

About this chapter

This chapter describes how to troubleshoot your database connection in PocketBuilder by using the following tools:

- Database Trace
- ODBC Driver Manager Trace

Contents

About tracing database connections

In PocketBuilder

In the development environment, you can trace connections as you work in the Database painter or DataWindow painter, and when you run and test your applications. The trace information that is collected can help you troubleshoot your applications before you deploy them. You can use two tools to collect information: the PocketBuilder Database Trace tool and the ODBC Driver Manager Trace tool.

The rest of this chapter describes these tools.

In a deployed application

In your deployed application, SQL Anywhere provides several different ways to create log files of database activity:

- To save the server window output to a file, start the database engine with the -o option for dbsrv*XX*, or dbeng*XX*, where *XX* is the SQL Anywhere version number. For example:

```
dbeng9 -c 8m -o asademo.out -n asademo9
"D:\Program Files\Sybase\asademo.db"
```

- To save client error messages and debugging messages in a file, use the Logfile connection parameter when you specify the ConnectString. For example:

```
sqlca.dbparm="ConnectString='DSN=ASA 9.0 Sample;LOGFILE=D:\logs\asademo.txt'"
```

- To save a log when you run the MobiLink synchronization server (dbmlsrv9 or mlsrv10) or client (dbmlsync) to synchronize SQL Anywhere remote databases with a consolidated database, use the -o option to specify the name of the output file, the -v option to specify the level of message logging, and the -dl option to display all logging messages on the screen. For example:

```
dbmlsrv9 -c "dsn=consoldb" -o mlserver.mls -v+ -dl
dbmlsync -c "dsn=remotedb" -o dbmlsync.out -v+ -dl
```

For more information about these tools, see the SQL Anywhere documentation.

Using the Database Trace tool

This section describes how to use the Database Trace tool in the PocketBuilder development environment.

About the Database Trace tool

The Database Trace tool records the internal commands that PocketBuilder executes while accessing a database. PocketBuilder writes the output of Database Trace to a log file named *PKTRACE.LOG* (by default) or to a nondefault log file that you specify. For instructions, see “Specifying a nondefault Database Trace log” on page 224.

When you enable database tracing for the first time, PocketBuilder creates the log file on your computer. Tracing continues until you disconnect from the database.

How you can use the Database Trace information

You can use information from the Database Trace tool to understand what PocketBuilder is doing *internally* when you work with your database. Examining the information in the log file can help you:

- Understand how PocketBuilder interacts with your database
- Identify and resolve problems with your database connection
- Provide useful information to Technical Support if you call them for help with your database connection

If you are familiar with PocketBuilder and your DBMS, you can use the information in the log to help troubleshoot connection problems on your own.

If you are less experienced or need help, run the Database Trace tool *before* you call Technical Support. You can then report or send the results of the trace to the Technical Support representative who takes your call.

You can view the log file using the built-in PocketBuilder file editor or any other text editor. You can also add your own annotations as you examine the file.

Contents of the Database Trace log

The Database Trace tool records the following information in the log file when you trace a database connection:

- Parameters used to connect to the database
- Time to perform each database operation (in milliseconds)
- The internal commands executed to retrieve and display table and column information from your database. Examples include:
  - Preparing and executing SQL statements such as SELECT, INSERT, UPDATE, and DELETE
  - Getting column descriptions
  - Fetching table rows
  - Binding user-supplied values to columns (if your database supports bind variables)
  - Committing and rolling back database changes
- Disconnection from the database
- Shutdown of the database interface

Format of the Database Trace log

The specific content of the Database Trace log file depends on the database you are accessing and the operations you are performing. However, the log uses the following basic format to display output:

COMMAND : (*time*)

{*additional_information*}

COMMAND is the internal command that PocketBuilder executes to perform the database operation.

time is the number of milliseconds it takes PocketBuilder to perform the database operation. The precision used depends on your operating system's timing mechanism.

additional_information is additional information about the command. The information provided, if any, depends on the database operation.

For an example of Database Trace output, see "Sample Database Trace output" on page 225.

Starting the Database Trace tool

By default, the Database Trace tool is turned off in PocketBuilder. You can start it by editing a database profile or a script.

Turning tracing on and off
To turn tracing on or off, you must reconnect. Setting and resetting are not sufficient.

Running Database Trace when you test your application

In the script where you set the values of SQLCA parameters, add the word TRACE to the value for the DBMS parameter. For example, if you are using the default transaction object, type:

```
SQLCA.DBMS = "TRACE ODB"
```

As an alternative to setting the DBMS property directly in an application script, you can use the PowerScript ProfileString function to read values from a specified section of an external text file, such as an application-specific initialization file.

Use the following PowerScript syntax to specify the ProfileString function with the DBMS property:

SQLCA.*variable* = ProfileString(*file*, *section*, *variable*, *default_value*)

For example, the following statement in a PocketBuilder script reads the DBMS value from the [Database] section of the *APP.INI* file:

```
SQLCA.dbms =
   ProfileString("APP.INI","Database","DBMS","")
```

Running Database Trace when you work in painters

To trace connection activity when you work in the DataWindow and Database painters, you set a property in the connection profile you are using.

❖ **To start the Database Trace tool by editing a profile:**

1 Open the Database Profile Setup dialog box for the connection you want to trace.

2 On the Connection tab, select the Generate Trace check box and click OK or Apply.

On the Preview tab, the setting that starts Database Trace is DBMS:

```
SQLCA.DBMS = "TRACE ODB"
```

3 Click Connect in the Database Profiles dialog box to connect to the database.

A message box displays, stating that database tracing is enabled and indicating where PocketBuilder will write the output.

4 Click OK.

PocketBuilder connects to the database and starts tracing the connection.

Stopping the Database Trace tool

Once you start tracing a particular database connection, PocketBuilder continues sending trace output to the log until you do one of the following:

- Reconnect to the same database with tracing stopped
- Connect to another database for which you have not enabled tracing

If you added the word TRACE in a script, you can simply delete it. If you added tracing in a connection profile, you need to edit the profile.

❖ **To stop the Database Trace tool by editing a profile:**

1. In the Database Profile Setup dialog box for the database you are tracing, clear the Generate Trace check box on the Connection tab.
2. Click OK in the Database Profile Setup dialog box.

   The Database Profiles dialog box displays with the name of the edited profile highlighted.
3. Right-click on the connected database and select Re-connect from the drop-down menu in the Database Profiles dialog box.

   PocketBuilder connects to the database and stops tracing the connection.

Specifying a nondefault Database Trace log

You can specify a nondefault name and location for the log file when you use Database Trace. This lets you:

- Control where PocketBuilder writes the output of the Database Trace tool
- Give the log file a name and location that best meets the development needs at your site

❖ **To specify a nondefault Database Trace log file:**

1. Open the PocketBuilder initialization file for editing.

   You can use the File Editor (in PocketBuilder) or any text editor (outside PocketBuilder).
2. Create an entry named DBTraceFile in the [Database] section of the initialization file, using the following syntax to specify a nondefault log file:

   DBTraceFile=*log_file_pathname*

   For example:

   ```
   [Database]
   ...
   DBTraceFile=c:\logs\mydbtrce.log
   ```
3. Save your changes to the initialization file.

   The next time you use the Database Trace tool to trace a connection in the development environment, PocketBuilder writes the output to the log file you specified instead of to the default *PKTRACE.LOG* file.

Deleting or clearing the Database Trace log

Each time you connect to a database with tracing enabled, PocketBuilder appends the trace output of your connection to the existing log. As a result, the log file can become very large over time.

❖ **To keep the size of the log file manageable:**

- Do either of the following periodically:
  - Open the log file, clear its contents, and save the empty file.

    Provided that you use the default *PKTRACE.LOG* or the same nondefault file the next time you connect to a database with tracing enabled, PocketBuilder will write to this empty file.
  - Delete the log file.

    PocketBuilder will automatically create a new log file the next time you connect to a database with tracing enabled.

Sample Database Trace output

This section gives an example of Database Trace output that you might see in the log file.

The example traces a connection to an ODBC database named ASA Sample. The output was generated while running a PocketBuilder application that displays information about employees. The SELECT statement shown retrieves information from the Employee table to display the IDs, names, and birth dates of employees. (In the trace output, each command line is prefixed with a hexadecimal string enclosed in parentheses and followed by a colon. This prefix is omitted from the output shown here.)

```
/*-------------------------------------------------*/
/*                5/1/2004  09:00                  */
/*-------------------------------------------------*/
LOGIN: (40 MilliSeconds)
CONNECT TO trace odb:
DBPARM=ConnectString='DSN=ASA Sample;UID=dba;PWD=sql'
SERVER=asademo (0 MilliSeconds)
PREPARE: (0 MilliSeconds)
BEGIN TRANSACTION: (0 MilliSeconds)
PREPARE:
select  "employee"."emp_id" , "employee"."emp_fname" ,
"employee"."emp_lname" , "employee"."birth_date" from
"employee" (0 MilliSeconds)
```

```
BIND SELECT OUTPUT BUFFER (DataWindow):(0 MilliSeconds)
,len=44,type=FLOAT,pbt=3,dbt=0,ct=0,prec=0,scale=0
,len=22,type=CHAR,pbt=1,dbt=0,ct=0,prec=0,scale=0
,len=22,type=CHAR,pbt=1,dbt=0,ct=0,prec=0,scale=0
,len=30,type=DATE,pbt=7,dbt=0,ct=0,prec=0,scale=0
EXECUTE: (0 MilliSeconds)
FETCH NEXT: (0 MilliSeconds)
    COLUMN=01COLUMN=FranCOLUMN=WhitneyCOLUMN=06-05-
1958 255:00:00:000000
FETCH NEXT: (0 MilliSeconds)
    COLUMN=02COLUMN=MatthewCOLUMN=CobbCOLUMN=12-04-
1960 255:00:00:000000
// ...
// Additional FETCH NEXT statements omitted
// ...
FETCH NEXT: (0 MilliSeconds)
COMMIT: (10 MilliSeconds)
DISCONNECT: (10 MilliSeconds)
SHUTDOWN DATABASE INTERFACE: (0 MilliSeconds)
```

Using the ODBC Driver Manager Trace tool

This section describes how to use the ODBC Driver Manager Trace tool.

About ODBC Driver Manager Trace

You can use the ODBC Driver Manager Trace tool to trace a connection to any ODBC data source that you access in PocketBuilder through the ODBC interface.

ODBC Driver Manager Trace records information about ODBC API calls (such as SQLDriverConnect, SQLGetInfo, and SQLFetch) that PocketBuilder makes while connected to an ODBC data source. It writes this information to a default log file named *SQL.LOG* or to a log file that you specify. The default location of *SQL.LOG* is in your root directory.

You can display the contents of the ODBC Driver Manager Trace log file at any time during a PocketBuilder session, using the built-in file editor or any other text editor.

Which tool to use

The information from ODBC Driver Manager Trace, like that from Database Trace, can help you:

- Understand what PocketBuilder is doing *internally* while connected to an ODBC data source
- Identify and resolve problems with your ODBC connection
- Provide information useful to Technical Support if you call them for help with your database connection

Use ODBC Driver Manager Trace *instead* of the Database Trace tool if you want more detailed information about the ODBC API calls made by PocketBuilder.

Performance considerations
Turning on ODBC Driver Manager Trace can slow performance in PocketBuilder. Therefore, use ODBC Driver Manager Trace for debugging purposes only and keep it turned off when you are not debugging.

Starting ODBC Driver Manager Trace

By default, ODBC Driver Manager Trace is turned off in PocketBuilder. You can start it in order to trace your ODBC connection in two ways:

- Edit a script in an application to trace the connection when you test the application
- Edit a database profile to trace actions in the DataWindow or Database painters

In both cases, you set values of the ConnectOption DBParm parameter.

The ConnectOption DBParm parameter

ConnectOption includes several parameters, two of which control the operation of ODBC Driver Manager Trace for any ODBC-compatible driver you are using in PocketBuilder:

- SQL_OPT_TRACE starts or stops ODBC Driver Manager Trace. The values you can specify are:
  - SQL_OPT_TRACE_OFF (the default), which stops ODBC Driver Manager Trace

  - SQL_OPT_TRACE_ON, which starts ODBC Driver Manager Trace
- SQL_OPT_TRACEFILE specifies the name of the trace file where you want to send the output of ODBC Driver Manager Trace. PocketBuilder appends the output to the trace file you specify until you stop the trace.

  You can specify any file name for the trace file. By default, if tracing is on and you have not specified a trace file, PocketBuilder sends ODBC Driver Manager Trace output to a file named *SQL.LOG* in your root directory.

Starting ODBC Driver Manager Trace in a script

The most efficient way to specify the correct values for the ConnectOption DBParm in a script is to set the options in the Database Connection Profile dialog box, as described in "Starting ODBC Driver Manager Trace in a database profile" next, and then copy them from the Preview page to the script. This example shows the settings for a data source named Employee. Note that in this example, the first statement specifies that Database Trace is also on:

```
SQLCA.DBMS = "TRACE ODBC"
SQLCA.AutoCommit = False
SQLCA.DBParm =
"ConnectString='DSN=Employee;UID=;PWD=',
ConnectOption='SQL_OPT_TRACE,SQL_OPT_TRACE_ON;
SQL_OPT_TRACEFILE,C:\logs\odbctrc.log'"
```

For more information, see "Copying DBParm syntax from the Preview tab" on page 288.

As an alternative to setting the DBParm property in an application script, you can use the PowerScript ProfileString function to read DBParm values from a specified section of an external text file, such as an application-specific initialization file.

Use the following PowerScript syntax to specify the ProfileString function with the DBParm property:

SQLCA.dbParm = ProfileString(*file*, *section*, *variable*, *default_value*)

For example, the following statement in a PocketBuilder script reads the DBParm values from the [Database] section of the *APP.INI* file:

```
SQLCA.dbParm =
   ProfileString("APP.INI","Database","DBParm","")
```

Starting ODBC Driver Manager Trace in a database profile

To start ODBC Driver Manager Trace to trace your connections in the PocketBuilder development environment, edit the database profile for the connection you want to trace, as described in the following procedure.

❖ **To start ODBC Driver Manager Trace by editing the database profile:**

1. Open the Database Profile Setup dialog box for the ODBC connection you want to trace.
2. On the Options tab, select the Trace ODBC API Calls check box.
3. (Optional) Specify a log file where you want PocketBuilder to write the output of ODBC Driver Manager Trace.

   You can type the path name in the Trace File box, or click Browse to browse to an existing log file.

   By default, if the Trace ODBC API Calls check box is selected and no trace file is specified, PocketBuilder sends ODBC Driver Manager Trace output to the default *SQL.LOG* file.
4. Click OK.

   PocketBuilder saves your settings in the registry in the *HKEY_CURRENT_USER\Software\Sybase\PocketBuilder\2.0\Database Profiles\Pocket PB* key.

   The following example shows the DBParm string value from a database profile entry for an ODBC data source named Employee. The settings that start ODBC Driver Manager Trace are in the ConnectOption DBParm parameter:

   ```
   ConnectString='DSN=Employee;UID=;PWD=',
   ConnectOption='SQL_OPT_TRACE,SQL_OPT_TRACE_ON;
   SQL_OPT_TRACEFILE,C:\logs\odbctrc.log'
   ```
5. Right-click on the connected database and select Re-connect from the drop-down menu in the Database Profiles dialog box.

   PocketBuilder connects to the database, starts tracing the ODBC connection, and writes output to the log file you specified.

Stopping ODBC Driver Manager Trace

Once you start tracing an ODBC connection with ODBC Driver Manager Trace, PocketBuilder continues sending trace output to the log file until you stop tracing. After you stop tracing, you must reconnect to have the changes take effect.

To stop tracing in a script, you can delete the ConnectOption parameter if it contains only trace parameters, or delete the SQL_OPT_TRACE options if ConnectOption contains other parameters.

You can also change the value of the SQL_OPT_TRACE parameter to SSQL_OPT_TRACE_OFF, as shown in the following example. This makes it easier to turn tracing on again later:

```
SQLCA.DBMS = "TRACE ODBC"
SQLCA.AutoCommit = False
SQLCA.DBParm =
"ConnectString='DSN=Employee;UID=;PWD=',
ConnectOption='SQL_OPT_TRACE,SQL_OPT_TRACE_OFF;
SQL_OPT_TRACEFILE,C:\logs\odbctrc.log'"
```

To stop tracing in the development environment, you need to edit the database profile.

❖ **To stop ODBC Driver Manager Trace by editing the database profile:**

1. Open the Database Profile Setup dialog box for the connection you are tracing.
2. On the Options tab, clear the Trace ODBC API Calls check box.

   If you supplied the pathname of a log file in the Trace File box, you can leave it specified in case you want to restart tracing later.
3. Click OK in the Database Profile Setup dialog box.
4. Right-click on the connected database and select Re-Connect from the dropdown menu in the Database Profiles dialog box.

   PocketBuilder connects to the database and stops tracing the connection.

Sample ODBC Driver Manager Trace output

This section shows a partial example of output from ODBC Driver Manager Trace to give you an idea of the information it provides. The example is part of the trace on an ODBC connection to the ASA Demo DB.

For more about a particular ODBC API call, see your ODBC documentation.

```
PK20    1d9-1bb EXIT  SQLSetConnectOptionW with return code 0 (SQL_SUCCESS)
         HDBC           01643D08
         UWORD                 104 <SQL_OPT_TRACE>
         SQLULEN               1

PK20    1d9-1bb ENTER SQLSetConnectOptionW
         HDBC           01643D08
         UWORD                 105 <SQL_OPT_TRACEFILE>
         SQLULEN        17654398
```

```
PK20    1d9-1bb EXIT  SQLSetConnectOptionW with return code 0 (SQL_SUCCESS)
         HDBC           01643D08
         UWORD                105 <SQL_OPT_TRACEFILE>
         SQLULEN         17654398

PK20    1d9-1bb ENTER SQLDriverConnectW
         HDBC           01643D08
         HWND           01DB04FE
         WCHAR *        0x1F7F8B88 [    -3] "******\ 0"
         SWORD                -3
         WCHAR *        0x1F7F8B88
         SWORD                 8
         SWORD *        0x00000000
         UWORD                 1 <SQL_DRIVER_COMPLETE>

PK20    1d9-1bb EXIT  SQLDriverConnectW  with return code 0 (SQL_SUCCESS)
         HDBC           01643D08
         HWND           01DB04FE
         WCHAR *        0x1F7F8B88 [    -3] "******\ 0"
         SWORD                -3
         WCHAR *        0x1F7F8B88
         SWORD                 8
         SWORD *        0x00000000
         UWORD                 1 <SQL_DRIVER_COMPLETE>

PK20    1d9-1bb ENTER SQLGetInfoW
         HDBC           01643D08
         UWORD                 6 <SQL_DRIVER_NAME>
         PTR            0x001293CC
         SWORD               258
         SWORD *        0x001293CA

PK20    1d9-1bb EXIT  SQLGetInfoW  with return code 0 (SQL_SUCCESS)
         HDBC           01643D08
         UWORD                 6 <SQL_DRIVER_NAME>
         PTR            0x001293CC [    22] "DBODBC9.DLL"
         SWORD               258
         SWORD *        0x001293CA (22)
...
```

CHAPTER 16

Using Transaction Objects

About this chapter

This chapter describes Transaction objects and how to use them in PocketBuilder applications.

Contents

About Transaction objects

In a PocketBuilder database connection, a Transaction object is a special nonvisual object that functions as the communications area between a PocketBuilder application and the database. The Transaction object specifies the parameters that PocketBuilder uses to connect to a database. You must establish the Transaction object before you can access the database from your application.

Communicating with the database

In order for a PocketBuilder application to display and manipulate data, the application must communicate with the database in which the data resides.

❖ **To communicate with the database from your PocketBuilder application:**

1. Assign the appropriate values to the Transaction object.
2. Connect to the database.
3. Assign the Transaction object to a DataWindow control or DataStore.
4. Perform the database processing.
5. Disconnect from the database.

For information about setting the Transaction object for a DataWindow control and using the DataWindow to retrieve and update data, see "Setting the transaction object for the DataWindow control" on page 123.

Default Transaction object

When you start executing an application, PocketBuilder creates a global default Transaction object named SQLCA (SQL Communications Area). You can use this default Transaction object in your application or define additional Transaction objects if your application has multiple database connections.

Transaction object properties

Each Transaction object has 15 properties, of which:

- Ten are used to connect to the database (not all apply to the database interfaces supported in PocketBuilder).
- Five are used to receive status information from the database about the success or failure of each database operation. These error-checking properties all begin with SQL.

Description of Transaction object properties

Table 16-1 describes each Transaction object property. For each of the connection properties, the table also lists the equivalent field in the Database Profile Setup dialog box that you complete to create a database profile in the PocketBuilder development environment.

Table 16-1: Transaction object properties

Property	Datatype	Description	In a database profile
DBMS	String	The three- or four-letter DBMS identifier for your connection. For SQL Anywhere, this is ODB. For UltraLite 9.x, it is UL9. For UltraLite 10.x, it is UL10.	Defined when you select a database interface
UserID	String	The name or ID of the user who connects to the database. UserID is optional for ODBC. (Be careful specifying the UserID property; it overrides the connection's UserName property returned by the ODBC SQLGetInfo call.)	User ID
Lock	String	For DBMSs that support the use of lock values and isolation levels, the isolation level to use when you connect to the database. For information about the lock values you can set, see Lock in the PocketBuilder *Connection Reference*.	Isolation Level
LogID	String	The name or ID of the user who logs in to the database server. PocketBuilder uses the LogID and LogPass properties only if the ODBC driver does *not* support the SQL driver CONNECT call.	—

Property	Datatype	Description	In a database profile
LogPass	String	The password used to log in to the database server.	—
AutoCommit	Boolean	Specifies whether PocketBuilder issues SQL statements outside or inside the scope of a transaction. Values you can set are: • True – PocketBuilder issues SQL statements *outside* the scope of a transaction; that is, the statements are not part of a logical unit of work (LUW). If the SQL statement succeeds, the DBMS updates the database immediately as if a COMMIT statement had been issued. • False (Default) – PocketBuilder issues SQL statements *inside* the scope of a transaction. PocketBuilder issues a BEGIN TRANSACTION statement at the start of the connection. In addition, PocketBuilder issues another BEGIN TRANSACTION statement after each COMMIT or ROLLBACK statement is issued. When you connect to an UltraLite database in the development environment, all processing in painters takes place as if AutoCommit is set to true. For more information, see AutoCommit in the *Connection Reference*.	AutoCommit Mode
DBParm	String	Contains connection parameters that support particular DBMS features. For a description of each DBParm parameter that PocketBuilder supports, see the *Connection Reference*.	Various
SQLReturnData	String	Contains DBMS-specific information.	—
SQLCode	Long	The success or failure code of the most recent SQL operation. For details, see “Error handling after a SQL statement” on page 244.	—
SQLNRows	Long	The number of rows affected by the most recent SQL operation. The database vendor supplies this number, so the meaning may be different for each DBMS.	—
SQLDBCode	Long	The database vendor’s error code. For details, see “Error handling after a SQL statement” on page 244.	—
SQLErrText	String	The text of the database vendor’s error message corresponding to the error code. For details, see “Error handling after a SQL statement” on page 244.	—

Working with Transaction objects

PocketBuilder uses a basic concept of database transaction processing called logical unit of work (LUW). LUW is synonymous with transaction. A transaction is a set of one or more SQL statements that forms an LUW. Within a transaction, all SQL statements must succeed or fail as one logical entity.

There are four PowerScript transaction management statements:

- COMMIT
- CONNECT
- DISCONNECT
- ROLLBACK

Transaction basics

CONNECT and DISCONNECT

A successful CONNECT starts a transaction, and a DISCONNECT terminates the transaction. All SQL statements that execute between the CONNECT and the DISCONNECT occur within the transaction.

Before you issue a CONNECT statement, the Transaction object must exist and you must assign values to all Transaction object properties required to connect to your DBMS.

COMMIT and ROLLBACK

When a COMMIT executes, all changes to the database since the start of the current transaction (or since the last COMMIT or ROLLBACK) are made permanent, and a new transaction is started. When a ROLLBACK executes, all changes since the start of the current transaction are undone and a new transaction is started.

AutoCommit setting

You can issue a COMMIT or ROLLBACK only if the AutoCommit property of the Transaction object is set to False (the default) and you have not already started a transaction using embedded SQL.

For more about AutoCommit, see “Description of Transaction object properties” on page 234.

Automatic COMMIT when disconnected
When a transaction is disconnected, PocketBuilder issues a COMMIT statement.

The default Transaction object

SQLCA

Since most applications communicate with only one database, PocketBuilder provides a global default Transaction object called SQLCA (SQL Communications Area).

PocketBuilder creates the Transaction object before the application's Open event script executes. You can use PowerScript dot notation to reference the Transaction object in any script in your application.

You can create additional Transaction objects as you need them, such as when you are using multiple database connections at the same time, but in most cases, SQLCA is the only Transaction object you need.

Example

This simple example uses the default Transaction object SQLCA to connect to and disconnect from an ODBC data source named Sample:

```
// Set the default Transaction object properties.
SQLCA.DBMS="ODB"
SQLCA.DBParm="ConnectString='DSN=Sample'"
// Connect to the database.
CONNECT USING SQLCA;
IF SQLCA.SQLCode < 0 THEN &
   MessageBox("Connect Error", SQLCA.SQLErrText,&
   Exclamation!)
...
// Disconnect from the database.
DISCONNECT USING SQLCA;
IF SQLCA.SQLCode < 0 THEN &
   MessageBox("Disconnect Error", SQLCA.SQLErrText,&
      Exclamation!)
```

Semicolons are SQL statement terminators
When you use embedded SQL in a PocketBuilder script, all SQL statements must be terminated with a semicolon (;). You do *not* use a continuation character for multiline SQL statements.

Assigning values to the Transaction object

Before you can use a default (SQLCA) or nondefault (user-defined) Transaction object, you must assign values to the Transaction object connection properties. To assign the values, use PowerScript dot notation.

Example

The following PowerScript statements assign values to the properties of SQLCA required to connect to the SQL Anywhere demo database through the PocketBuilder ODB database interface:

```
SQLCA.DBMS = 'odb'
SQLCA.DBParm = "ConnectString='DSN=SQLAny 10 Demo;UID=dba;PWD=sql'"
```

The following PowerScript statements assign values to the properties of SQLCA required to connect to a Sybase UltraLite 9.x database through the PocketBuilder UL9 database interface:

```
SQLCA.DBMS = "UL9"
SQLCA.DBParm = "ConnectString='DBF=\UltraLite\uleq.udb;UID=dba;PWD=sql'"
```

Reading values from an external file

Using external files

You might want to set the Transaction object values from an external file. For example, you might want to retrieve values from your PocketBuilder initialization file when you are developing the application, or from an application-specific initialization file when you distribute the application.

ProfileString function

You can use the PowerScript ProfileString function to retrieve values from a text file that is structured into sections containing variable assignments, like a Windows INI file. The PocketBuilder initialization file is such a file, consisting of several sections including *PB*, *Application*, and *Database*:

```
[PB]
variables and their values
...
[Application]
variables and their values
...
[Database]
variables and their values
...
```

The ProfileString function has this syntax:

ProfileString (*file, section, key, default*)

Example

This script reads values from an initialization file to set the Transaction object to connect to a database. Conditional code sets the variable *startupfile* to an appropriate value:

```
SQLCA.DBMS = ProfileString(startupfile, "database",  &
   "dbms", "")
SQLCA.DBParm = ProfileString(startupfile, "database", &
   "dbparm", "")
```

Connecting to the database

Once you establish the connection parameters by assigning values to the Transaction object properties, you can connect to the database using the SQL CONNECT statement:

```
// Transaction object values have been set.
CONNECT;
```

Because CONNECT is a SQL statement, not a PowerScript statement, you need to terminate it with a semicolon.

If you are using a Transaction object other than SQLCA, you *must* include the USING TransactionObject clause in the SQL syntax:

CONNECT USING *TransactionObject*;

For example:

```
CONNECT USING MyTrans;
```

Using the Preview tab to connect in a PocketBuilder application

The Preview tab page in the Database Profile Setup dialog box makes it easy to generate accurate PowerScript connection syntax in the development environment for use in your PocketBuilder application script.

As you complete the Database Profile Setup dialog box, the correct PowerScript connection syntax for each selected option is generated on the Preview tab. PocketBuilder assigns the corresponding DBParm parameter or SQLCA property name to each option and inserts quotation marks, commas, semicolons, and other characters where needed. You can copy the syntax you want from the Preview tab directly into your script.

❖ **To use the Preview tab to connect in a PocketBuilder application:**

1 In the Database Profile Setup dialog box for your connection, supply values for basic options (on the Connection tab) and additional database parameters and SQLCA properties (on the other tabbed pages) as required.

For information about connection parameters and the values you should supply, click Help.

2 Click Apply to save your settings without closing the Database Profile Setup dialog box.

3 Click the Preview tab.

The correct PowerScript connection syntax for each selected option displays in the Database Connection Syntax box on the Preview tab.

4 Select one or more lines of text in the Database Connection Syntax box and click Copy.

PocketBuilder copies the selected text to the clipboard. You can then paste this syntax into your script, modifying the default Transaction object name (SQLCA) if necessary.

ODB or ODBC for DBMS value
An ODBC connection profile shows "ODBC" as the value for the DBMS parameter. Only the first three characters in this string are used, so ODB and ODBC both work correctly.

5 Click OK.

Disconnecting from the database

When your database processing is completed, you disconnect from the database using the SQL DISCONNECT statement:

```
DISCONNECT;
```

If you are using a Transaction object other than SQLCA, you *must* include the USING TransactionObject clause in the SQL syntax:

DISCONNECT USING *TransactionObject*;

For example:

```
DISCONNECT USING MyTrans;
```

Automatic COMMIT when disconnected
When a transaction is disconnected, PocketBuilder issues a COMMIT statement by default.

Defining Transaction objects for multiple database connections

Use one Transaction object per connection

To perform operations in multiple databases at the same time, you need to use multiple Transaction objects, one for each database connection. You must declare and create the additional Transaction objects before referencing them, and you must destroy these Transaction objects when they are no longer needed.

Caution
PocketBuilder creates and destroys SQLCA automatically. Do not attempt to create or destroy it.

Creating the nondefault Transaction object

To create a Transaction object other than SQLCA, you first declare a variable of type transaction:

transaction *TransactionObjectName*

You then instantiate the object:

TransactionObjectName = CREATE transaction

For example, to create a Transaction object named DBTrans, code:

```
transaction DBTrans
DBTrans = CREATE transaction
// You can now assign property values to DBTrans.
DBTrans.DBMS = "ODB"
...
```

Assigning property values

When you assign values to properties of a Transaction object that you declare and create in a PocketBuilder script, you *must* assign the values *one property at a time*, like this:

```
// This code produces correct results.
transaction SQLAnyTrans
SQLAnyTrans = CREATE TRANSACTION
SQLAnyTrans.DBMS = "Sybase"
SQLAnyTrans.Database = "Personnel"
```

You *cannot* assign values by setting the nondefault Transaction object equal to SQLCA, like this:

```
// This code produces incorrect results.
transaction  MyTrans
MyTrans = CREATE TRANSACTION
MyTrans = SQLCA // ERROR!
```

Specifying the Transaction object in SQL statements

When a database statement requires a Transaction object, PocketBuilder assumes the Transaction object is SQLCA unless you specify otherwise. These CONNECT statements are equivalent:

```
CONNECT;
CONNECT USING SQLCA;
```

However, when you use a Transaction object *other* than SQLCA, you *must* specify the Transaction object in the SQL statements in Table 16-2 with the USING TransactionObject clause.

Table 16-2: SQL statements that require USING TransactionObject

COMMIT	INSERT
CONNECT	PREPARE (dynamic SQL)
DELETE	ROLLBACK
DECLARE Cursor	SELECT
DECLARE Procedure	SELECTBLOB
DISCONNECT	UPDATEBLOB
EXECUTE (dynamic SQL)	UPDATE

❖ **To specify a user-defined Transaction object in SQL statements:**

- Add the following clause to the end of any of the SQL statements in the preceding list:

  USING *TransactionObject*

  For example, this statement uses a Transaction object named MyTrans to connect to the database:

  ```
  CONNECT USING MyTrans;
  ```

Always code the Transaction object

Although specifying the USING TransactionObject clause in SQL statements is optional when you use SQLCA and required when you define your own Transaction object, it is good practice to code it for any Transaction object, including SQLCA. This avoids confusion and ensures that you supply USING TransactionObject when it is required.

Example

The following statements use the default Transaction object (SQLCA) to communicate with an UltraLite database and a nondefault Transaction object named SQLAnyTrans to communicate with a SQL Anywhere database:

```
// Set the default Transaction object properties.
SQLCA.DBMS = "UL9"
SQLCA.DBParm = "ConnectString='DBF=\Test\test.ucb'"
// Connect to the UltraLite database.
CONNECT USING SQLCA;

// Declare a SQL Anywhere Transaction object.
transaction SQLAnyTrans
// Create the SQL Anywhere Transaction object.
SQLAnyTrans = CREATE TRANSACTION
// Set the SQL Anywhere Transaction object properties.
SQLAnyTrans.DBMS = "ODB"
SQLAnyTrans.DBParm = "ConnectString='DSN=Work'"
// Connect to the SQL Anywhere database.
CONNECT USING SQLAnyTrans;

// Insert a row into the first database.
INSERT INTO CUSTOMER
VALUES ( 'CUST789', 'BOSTON' )
USING SQLCA;
// Insert a row into the second database.
INSERT INTO EMPLOYEE
VALUES ( 'Peter Smith', 'New York' )
USING SQLAnyTrans;

// Disconnect from the first database
DISCONNECT USING SQLCA;
// Disconnect from the second database.
DISCONNECT USING SQLAnyTrans;
// Destroy the SQL Anywhere Transaction object.
DESTROY SQLAnyTrans
```

Using error checking

An actual script would include error checking after the CONNECT, INSERT, and DISCONNECT statements. For details, see “Error handling after a SQL statement” next.

Error handling after a SQL statement

When to check for errors

You should always test the success or failure code (the SQLCode property of the Transaction object) after issuing one of the following statements in a script:

- Transaction management statement (such as CONNECT, COMMIT, and DISCONNECT)
- Embedded or dynamic SQL

Not in a DataWindow
Do *not* do this type of error checking following a retrieval or update made in a DataWindow. For information about handling errors in DataWindow objects, see "Handling DataWindow errors" on page 136.

SQLCode return values

Table 16-3 shows the SQLCode return values.

Table 16-3: SQLCode return values

Value	Meaning
0	Success
100	Fetched row not found
-1	Error (the statement failed) Use SQLErrText or SQLDBCode to obtain the details.

Using SQLErrText and SQLDBCode

The string SQLErrText in the Transaction object contains the database vendor-supplied error message. The long named SQLDBCode in the Transaction object contains the database vendor-supplied status code. You can reference these variables in your script.

Example To display a message box containing the DBMS error number and message if the connection fails, code the following:

```
CONNECT USING SQLCA;
IF SQLCA.SQLCode = -1 THEN
   MessageBox("SQL error " + String(SQLCA.SQLDBCode),&
   SQLCA.SQLErrText )
END IF
```

Using Transaction objects to call stored procedures

SQLCA is a built-in global variable of type transaction that is used in all PocketBuilder applications. In your application, you can define a specialized version of SQLCA that performs certain processing or calculations on your data.

Not supported in UltraLite
Stored procedures are not supported in UltraLite databases.

You might already have defined remote stored procedures to perform these operations. You can use the remote procedure call (RPC) technique to define a customized version of the Transaction object that calls these database stored procedures in your application.

Result sets
You *cannot* use the RPC technique to access result sets returned by stored procedures. If the stored procedure returns one or more result sets, PocketBuilder ignores the values and returns the output parameters and return value. If your stored procedure returns a result set, you can use the embedded SQL DECLARE Procedure statement to call it.

For information about the DECLARE Procedure statement, see the chapter on SQL statements in the *PowerScript Reference* or the online Help.

Overview of the RPC procedure

To call database stored procedures from within your PocketBuilder application, you can use the remote procedure call technique and PowerScript dot notation (*object*.*function*) to define a customized version of the Transaction object that calls the stored procedures.

❖ **To call database stored procedures in your application:**

1. From the Objects tab in the New dialog box, define a standard class user object inherited from the built-in Transaction object.
2. In the Script view in the User Object painter, use the RPCFUNC keyword to declare the stored procedure as an external function or subroutine for the user object.
3. Save the user object.
4. In the Application painter, specify the user object you defined as the default global variable type for SQLCA.
5. Code your PocketBuilder application to use the user object.

For instructions on using the User Object and Application painters and the Script view in PocketBuilder, see the *User's Guide*.

Understanding the example

u_trans_database user object The following sections give step-by-step instructions for using a Transaction object to call stored procedures in your application. The example shows how to define and use a standard class user object named u_trans_database.

The u_trans_database user object is a descendant of (inherited from) the built-in Transaction object SQLCA. A descendant is an object that inherits functionality (properties, variables, functions, and event scripts) from an ancestor object. A descendent object is also called a subclass.

The example uses a simple stored procedure that takes a salary as an input and returns the value of the salary after a 5% raise:

```
CREATE FUNCTION DBA."sp_raise" ( salary double )
RETURNS double
BEGIN
    DECLARE salary double;
    SET salary = salary * 1.05;
    RETURN salary;
END
```

Step 1: define the standard class user object

❖ **To define the standard class user object:**

1. Start PocketBuilder.
2. Connect to the database.
3. Click the New button in the PowerBar, or select File>New from the menu bar.
4. On the PB Object tab in the New dialog box, select the Standard Class icon, and click OK to define a new standard class user object.

   The Select Standard Class Type dialog box displays, listing all the standard class types provided in PocketBuilder.
5. Select *transaction* as the built-in system type that you want your user object to inherit from, and click OK.

   The User Object painter workspace displays so that you can assign properties (instance variables) and functions to your user object.

Step 2: declare the stored procedure as an external function

FUNCTION or SUBROUTINE declaration

You can declare a non-result-set database stored procedure as an external function or external subroutine in a PocketBuilder application. If the stored procedure has a return value, declare it as a function (using the FUNCTION keyword). If the stored procedure returns nothing or returns VOID, declare it as a subroutine (using the SUBROUTINE keyword).

RPCFUNC and ALIAS FOR keywords

You *must* use the RPCFUNC keyword in the function or subroutine declaration to indicate that this is a remote procedure call (RPC) for a database stored procedure rather than for an external function in a dynamic library. Optionally, you can use the ALIAS FOR "*spname*" expression to supply the name of the stored procedure as it appears in the database if this name differs from the one you want to use in your script.

For complete information about the syntax for declaring stored procedures as remote procedure calls, see the chapter on calling functions and events in the *PowerScript Reference*.

❖ **To declare stored procedures as external functions for the user object:**

1. In the Script view in the User Object painter, select [Declare] from the first list and Local External Functions from the second list.
2. Place your cursor in the Declare Local External Functions view. From the pop-up menu or the Edit menu, select Paste Special>SQL>Remote Stored Procedures.

   PocketBuilder 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.

Figure 16-1: Remote Stored Procedures dialog box

3 Select the names of one or more stored procedures that you want to declare as functions for the user object, and click OK.

PocketBuilder retrieves the stored procedure declarations from the database and pastes each declaration into the view.

Here is the declaration that displays (on one line) when you select sp_raise:

```
function double sp_raise(double salary) RPCFUNC
ALIAS FOR "~"dba~".~"sp_raise~""
```

4 Edit the stored procedure declaration as needed for your application.

Use either of the following syntax formats to declare the database remote procedure call (RPC) as an external function or external subroutine:

FUNCTION *rtndatatype functionname* ({ { REF } *datatype1 arg1*, ..., { REF } *datatypen argn* }) RPCFUNC { ALIAS FOR "*spname*" }

SUBROUTINE *functionname* ({ { REF } *datatype1 arg1* , ..., { REF } *datatypen argn* }) RPCFUNC { ALIAS FOR "*spname*" }

For details about the syntax, see the *PowerScript Reference* or the online Help.

Step 3: save the user object

❖ **To save the user object:**

1 In the User Object painter, click the Save button, or select File>Save from the menu bar.

The Save User Object dialog box displays.

2 Specify the name of the user object, comments that describe its purpose, and the library in which to save the user object, and click OK.

PocketBuilder saves the user object with the name you specified in the selected library.

Step 4: specify the default global variable type for SQLCA

This procedure assumes that your application uses the default Transaction object SQLCA, but you can also declare and create an instance of your own Transaction object and then write code that calls the user object as a property of your Transaction object. For instructions, see the chapter on working with user objects in the *User's Guide*.

In the Application painter, you must specify the user object you defined as the default global variable type for SQLCA. When you execute your application, this tells PocketBuilder to use your standard class user object instead of the built-in system Transaction object.

❖ **To specify the default global variable type for SQLCA:**

1 Click the Open button in the PowerBar, or select File>Open from the menu bar.

2 In the Open dialog box, select Applications from the Object Type drop-down list. Choose the application where you want to use your new user object and click OK.

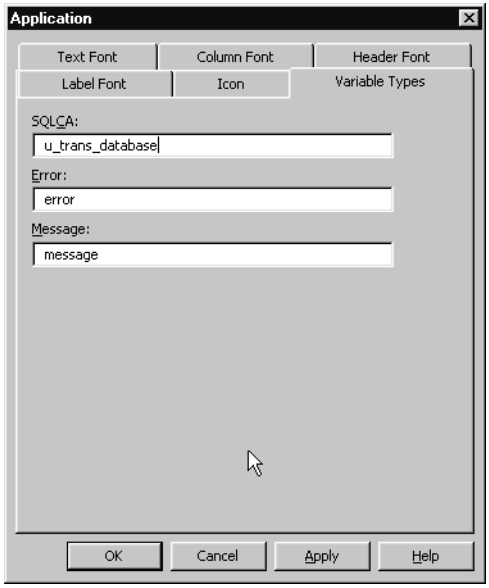
The Application painter workspace displays.

3 Select the General tab in the Properties view. Click the Additional Properties button.

4 In the Additional Properties dialog box, click the Variable Types tab to display the Variable Types property page.

5 In the SQLCA box, specify the name of the standard class user object you defined in Steps 1 through 3.

Figure 16-2: Specifying a user-defined Transaction object as the default global variable for SQLCA



6 Click OK or Apply.

When you execute your application, PocketBuilder will use the specified standard class user object instead of the built-in system object type it inherits from.

Step 5: code your application to use the user object

What you have done so far In the previous steps, you defined a 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 PocketBuilder application access to the properties and functions encapsulated in the user object.

What you do now You now need to write code that uses the user object to perform the necessary processing.

In your application script, you can use PowerScript dot notation to call the stored procedure functions you defined for the user object, just as you do when using SQLCA for all other PocketBuilder objects. The dot notation syntax is:

object.*function* (*arguments*)

For example, you can call the sp_raise stored procedure with code similar to the following:

```
double ld_result
double ld_inputsalary
ld_result = sqlca.sp_raise( ld_inputsalary )
```

❖ **To code your application to use the user object:**

1 Open the object or control for which you want to write the script.

2 Select the event for which you want to write the script.

3 Write code that uses the user object to do the necessary processing for your application.

4 Compile the script to save your changes.

Supported DBMS features when calling stored procedures

When you define and use a custom Transaction object to call remote stored procedures in your application, the features supported depend on the DBMS to which your application connects. Stored procedures are not supported in UltraLite databases.

Result sets

You *cannot* use the remote procedure call technique to access result sets returned by stored procedures. If the stored procedure returns one or more result sets, PocketBuilder ignores the values and returns the output parameters and return value.

If your stored procedure returns a result set, you can use the embedded SQL DECLARE Procedure statement to call it. For information about the DECLARE Procedure statement, see the chapter on SQL statements in the online Help.

If your application connects to an SQL Anywhere database, you can use the following features:

- IN, OUT, and IN OUT parameters, as shown in Table 16-4.

Table 16-4: SQL Anywhere IN, OUT, and IN OUT parameters

Parameter	What happens
IN	An IN variable is passed by value and indicates a value being passed to the procedure.
OUT	An OUT variable is passed by reference and indicates that the procedure can modify the PowerScript variable that was passed. Use the PowerScript REF keyword for this parameter type.
IN OUT	An IN OUT variable is passed by reference and indicates that the procedure can reference the passed value and can modify the PowerScript variable. Use the PowerScript REF keyword for this parameter type.

- Blobs as parameters. You can use blobs that are up to 32,512 bytes long.

CHAPTER 17

Using MobiLink Synchronization

About this chapter

This chapter provides an introduction to MobiLink synchronization. It also describes PocketBuilder wizards and utilities that help you control database synchronization from a PocketBuilder application, how to prepare to use the wizards, and how to use the objects created by the wizards.

Contents

About MobiLink synchronization

MobiLink is a session-based synchronization system that allows one- or two-way synchronization between a central data source, typically a consolidated database, and many remote databases. Administration and resource requirements at the remote database sites are minimal, making MobiLink well suited to a variety of mobile applications.

Where to find additional information

Detailed information about MobiLink synchronization is provided in the *MobiLink Getting Started*, the *MobiLink - Client Administration*, and the *Mobilink - Server Administration* books. These books are available online on the SQL Anywhere Product Manuals Web site at http://www.ianywhere.com/developer/product_manuals/sqlanywhere/.

If you are already familiar with MobiLink, go to "Working with PocketBuilder synchronization objects" on page 260 to learn about PocketBuilder integration with MobiLink.

This section introduces some MobiLink terms and concepts.

Data movement and synchronization

Data movement occurs when shared data is distributed over multiple databases on multiple nodes and changes to data in one database are applied to the corresponding data in other databases. Data can be moved using replication or synchronization.

Data replication moves all transactions from one database to another, whereas data synchronization moves only the net result of transactions. Both techniques get their information by scanning transaction log files, but synchronization uses log file segments instead of the full log file, making data movement much faster and more efficient.

With synchronization, data is available locally and can be modified without a connection to a server. MobiLink synchronization uses a loose consistency model, which means that all changes are synchronized with each site over time in a consistent manner, but different sites might have different copies of data at any instant. Only successful transactions are synchronized.

Consolidated and remote databases

The consolidated database, which can be any ODBC-compliant database, such as SQL Anywhere, Adaptive Server Enterprise, Oracle, IBM DB2 UDB, or Microsoft SQL Server, holds the master copy of all the data. Optionally, in MobiLink 10, you can store all or part of your central data in a data source such as an application server, spreadsheet, Web server, Web service, or text file.

For information on using a central data source other than a consolidated database, see the chapter on direct row handling in the *MobiLink Server Administration* book.

The remote database contains a subset of the consolidated data. In PocketBuilder, the remote database is a SQL Anywhere or UltraLite database.

The MobiLink synchronization server

The MobiLink synchronization server, dbmlsrv9 or mlsrv10, manages the synchronization process and provides the interface between remote databases and the consolidated database server. All communication between the MobiLink synchronization server and the consolidated database occurs through an ODBC connection.The consolidated database and synchronization server often reside on the same machine, but that is not a requirement.

The MobiLink server must be running before a synchronization process is launched.

As you build and test PocketBuilder applications, you can start the server from the Utilities folder in the Objects view in the Database painter. For more information, see the chapter on managing databases in the PocketBuilder *User's Guide*.

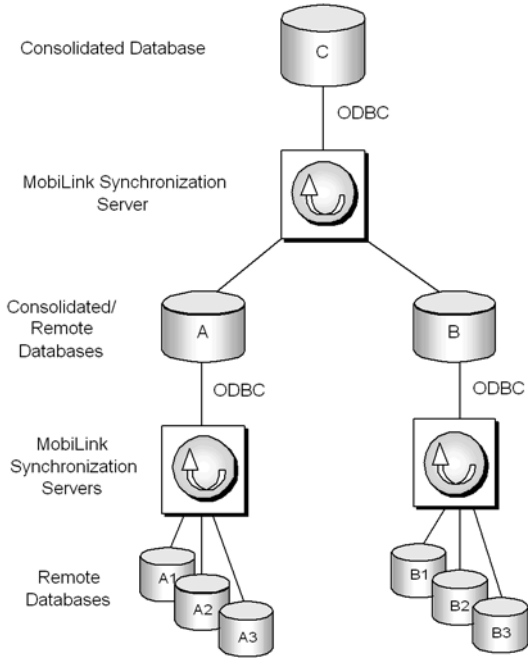
For information about starting the server from the command line, see "mlsrv10" in the index of the SQL Anywhere Studio online books.

MobiLink hierarchy

MobiLink typically uses a hierarchical configuration. The nodes in the hierarchy can reside on servers, desktop computers, and handheld or embedded devices. A simple hierarchy might consist of a consolidated database on a server and multiple remote databases on mobile devices. A more complex hierarchy might contain multiple levels in which some sites act as both remote and consolidated databases. Any consolidated database that also acts as a remote database must be a SQL Anywhere database.

For example, suppose remote sites A1, A2, and A3 synchronize with a consolidated database A on a local server, and remote sites B1, B2, and B3 synchronize with a consolidated database B on another local server. A and B in turn act as remote sites and synchronize with a consolidated database C on a master server. C can be any ODBC-compliant database, but A and B must both be SQL Anywhere databases.

Figure 17-1: MobiLink hierarchy



Synchronization scripts

MobiLink synchronization is an event-driven process. When a MobiLink client initiates a synchronization, a number of synchronization events occur inside the MobiLink server. When an event occurs, MobiLink looks for a script to match the synchronization event. If you want the MobiLink server to take an action, you must provide a script for the event.

You can write synchronization scripts for connection-level events and for events for each table in the remote database. You save these scripts on the consolidated database.

You can write scripts using SQL, Java, or .NET. For more information about event scripts and writing them in the MobiLink Synchronization plug-in in Sybase Central, see “Preparing consolidated databases” on page 270.

The MobiLink synchronization SQL Anywhere client

SQL Anywhere clients at remote sites initiate synchronization by running a command-line utility called dbmlsync. This utility synchronizes one or more subscriptions in a remote database with the MobiLink synchronization server. Subscriptions are described in “Publications, articles, and users” on page 257. For more information about the dbmlsync utility and its options, see “dbmlsync utility” in the index of the SQL Anywhere online books.

In PocketBuilder, synchronization objects that you create with the MobiLink Synchronization for ASA wizard manage the dbmlsync process. For more information, see "Working with PocketBuilder synchronization objects" on page 260.

A different PocketBuilder wizard, the MobiLink Sync User and Subscription Maintenance wizard, lets you provide application users with the ability to create user names and subscriptions in a remote database. For more information about this wizard, see the chapter on managing the database in the PocketBuilder *User's Guide*.

The MobiLink synchronization UltraLite client

One of the major differences in synchronizing an UltraLite database instead of a SQL Anywhere database is that there are no subscriptions in an UltraLite database. By default, all tables in an UltraLite remote database are updated during synchronization, although you can create publications in the database to restrict the updates to specific tables. Another difference is that the MobiLink synchronization call is made directly on the connection object to the remote UltraLite database, rather than through an outside call to a separate synchronization utility.

MobiLink synchronization requires matching publications to be defined in both the consolidated and remote databases, and synchronization scripts to be defined on the consolidated databases. "Preparing to use the wizard for remote UltraLite databases" on page 268 describes the steps you must take to prepare the databases for synchronization using utilities you can run from the Database painter. You can prepare the databases before or after running the UltraLite Synchronization wizard that helps you integrate MobiLink synchronization with your PocketBuilder applications.

For more information on the UltraLite Synchronization wizard, see the Managing Databases chapter in the *User's Guide*.

Publications, articles, and users

A publication is a database object on the remote database that identifies tables and columns to be synchronized. Each publication can contain one or more articles. An article is a database object that represents a whole table, or a subset of the columns and rows in a table.

In MobiLink 9, a MobiLink user is a database object that uniquely identifies a remote database, and is also used to authenticate a person who synchronizes. There is one MobiLink user name for each remote database in the MobiLink system.

In MobiLink 10, a MobiLink user is used only to authenticate a person who synchronizes. User names do not need to be unique. Instead, an identifier called a remote ID uniquely identifies a remote database. The remote ID is stored in the remote database. MobiLink generates a remote ID the first time a remote database synchronizes to a central data source, or any time it encounters a NULL value for the remote ID. The remote ID is created automatically as a GUID, but you can set it to any string that has meaning to you.

The remote ID makes it easier for the same MobiLink user to synchronize different sets of data in different remote databases. For SQL Anywhere 10 remote databases, the MobiLink server tracks synchronization progress by remote ID and subscription. In UltraLite 10 remote databases, the remote ID is also useful for allowing multiple MobiLink users to synchronize the same remote database. For these databases, the MobiLink server tracks synchronization progress by remote ID and publication.

Every script that accepts the MobiLink user name as a parameter now also accepts a remote_id parameter. The remote_id parameter is available only if you use named parameters.

In both versions 9 and 10, MobiLink users are created on the remote database and registered on the consolidated database. You can register users with the mluser utility, the mlsrv10 -zu option, or in other ways. Once registered, MobiLink user names are stored in the ml_user system table on the consolidated database.

Subscriptions

A subscription associates a user with one or more publications. It specifies the synchronization protocol (such as TCP/IP, HTTP, or HTTPS), address (such as *myserver.acmetools.com*), and additional optional connection and extended options.

All of these objects are created in the *remote* database, although subscriptions are not used with UltraLite databases. In Sybase Central, you create publications, users, and subscriptions using the SQL Anywhere plug-in, not the MobiLink Synchronization plug-in. For more information, see “Creating remote databases” on page 277.

The MobiLink Sync User and Subscription Maintenance wizard creates objects that allow a PocketBuilder application user to create MobiLink users and subscriptions in a remote database. For more information, see the chapter on managing the database in the PocketBuilder *User's Guide*.

Create Synchronization Model wizard

The MobiLink 10 plug-in for Sybase Central provides the Create Synchronization Model wizard to help you set up synchronization between a remote database and a consolidated database. You can use the wizard to create scripts, publications, users, and so on. You can also use the wizard to create the remote database and all its objects based on the schema of a consolidated database. You start the wizard by selecting the Tools>MobiLink 10>Setup MobiLink Synchronization menu item or by clicking "Create a Synchronization Model" in the Sybase Central Task view.

The synchronization process

For remote SQL Anywhere databases Dbmlsync connects to the remote database using TCP/IP, HTTP, or HTTPS, and prepares a stream of data (the upload stream) to be uploaded to the consolidated database. Dbmlsync uses information contained in the transaction log of the remote database to build the upload stream. The upload stream contains the MobiLink user name and password, the version of synchronization scripts to use, the last synchronization timestamp, the schema of tables and columns in the publication, and the net result of all inserts, updates, and deletes since the last synchronization.

After building the upload stream, dbmlsync uses information stored in the specified publication and subscription to connect to the MobiLink synchronization server and exchange data.

For remote UltraLite databases The Synchronize call on the connection object to the remote database uses the selected communication stream (TCP/IP, HTTP, or HTTPS) and prepares a stream of data to be uploaded to the consolidated database. Information passed in a structure object is used to connect to the MobiLink synchronization server and exchange data.

For all remote databases When the MobiLink synchronization server receives data, it updates the consolidated database, then builds a download stream that contains all relevant changes and sends it back to the remote site. At the end of each successful synchronization, the consolidated and remote databases are consistent. Either a whole transaction is synchronized, or none of it is synchronized. This ensures transactional integrity at each database.

Working with PocketBuilder synchronization objects

When you run the MobiLink Synchronization for ASA or the UltraLite Synchronization wizard from the Database page in the New dialog box, the wizard generates objects that let you initiate and control MobiLink synchronization requests from a PocketBuilder application. These objects let you obtain feedback during the synchronization process, code PowerScript events at specific points during synchronization, and cancel the process programmatically.

To get started, create a new workspace and a target application. You do not need to create a SQL database connection, but you do need to create a project.

Using SalesDB as a sample remote SQL Anywhere database

Before you use the MobiLink Synchronization for ASA wizard to generate objects for an application, you need to set up a remote database and add at least one publication, user, and subscription to it, and create a PocketBuilder database profile for the remote database. To test the synchronization objects on the Pocket PC device and emulator, you need to set up a consolidated database.

You can create your own databases, as described in "Preparing consolidated databases" on page 270 and "Creating remote databases" on page 277, or use the databases provided for the SalesDB sample application.

You can set up the SalesDB consolidated and remote databases using the *MakeDB.cmd* file located in the *Code Examples\SalesDB\db* directory in your PocketBuilder installation. The remote database already has a publication (salesapi), user (tutorial), and subscription.

Instructions for running the command file and setting up a database profile are in the *SalesDB.html* file in the *SalesDB* directory. There is also a tutorial version of the application in the *Tutorial* directory.

Adding synchronization capabilities to your application

To test the synchronization objects generated by the MobiLink Synchronization for ASA wizard or the UltraLite Synchronization wizard, complete the following steps:

1 Run the wizard.

2 Call synchronization objects from your application.

3 Deploy the application and database files.

4 Start the MobiLink server.

5 Run the application.

Run the wizard

For remote SQL Anywhere databases The wizard prompts you for a database profile, a file DSN, and a publication. You can use the SalesDB values if you set up the databases and profile. Continue through the wizard, selecting default values, and click Finish to generate the synchronization objects.

For remote UltraLite databases The wizard prompts you for a publication and a script version. For testing purposes, you can leave the script version blank and, for UltraLite 9, select the Send Column Names check box. In UltraLite 10, the column names are used by the MobiLink server for direct row handling. You need to set this option only when using the row handling API to refer to columns by name rather than by index.

Continue through the wizard, selecting default values, and click Finish to generate the synchronization objects.

For all remote databases For information on the objects generated by the wizards for remote SQL Anywhere or UltraLite databases, see the chapter on managing databases in the *User's Guide*. For help in wizard screens, place the mouse pointer in any wizard field and press F1.

Call synchronization objects from your application

In a Menu object for your application, add two submenu items to the File menu, called Synchronize and Sync Options. Alternatively, you can use these names to label two buttons on an application window.

For remote SQL Anywhere databases Add the following code to the Clicked event of the Synchronize menu item or button (*appname* is the name of your application):

```
gf_appname_sync("tutorial", "")
```

Add the following code to the Clicked event of the Sync Options menu item or button:

```
gf_appname_options_sync()
```

For remote UltraLite databases Add the following code to the Clicked event of the Synchronize menu item or button (*appname* is the name of your application):

```
gf_appname_ulsync("dba", "sql", sqlca)
```

Add the following code to the Clicked event of the Sync Options menu item or button:

```
gf_appname_options_ulsync(sqlca)
```

Deploy the application and database files

Use the Project painter to deploy the application to the Pocket PC or emulator. For remote SQL Anywhere databases, you need to copy the file DSN for the remote database to the root directory of the device or emulator, and copy the remote database and its transaction log file to the directory on the device or emulator specified in the DSN.

For more information about copying files to devices or emulators, see the *Installation Guide*.

Start the MobiLink server

Select MobiLink Synchronization Server from the Utilities folder in the Database painter. In the dialog box, select the Automatic Addition of Users check box. This ensures that the MobiLink names created in remote databases are registered for synchronization. Click OK to start the server.

Testing purposes only
Typically, you would not select the Automatic Addition of Users option for applications that you create for production.

Run the application

Run the application on the device or emulator and select the File>Synchronize and File>Sync Options menu items or buttons to test their operation.

Using the synchronization objects in your application

Before you use objects generated by one of the MobiLink synchronization wizards, you should examine them in the PocketBuilder painters to understand how they interact. Many of the function and event scripts contain comments that describe their purpose.

All the source code is provided so that you have total control of how your application manages synchronization. You can use the objects as they are, modify them, or use them as templates for your own objects.

Instance variables in the user object

MobiLink Synchronization for SQL Anywhere The nvo_*appname*_sync user object contains instance variables that represent specific dbmlsync arguments, including the publication name, the MobiLink server host name and port, the DSN used on the desktop, and the file DSN created for deployment to the Pocket PC.

UltraLite Synchronization The nvo_*appname*_ulsync user object contains instance variables that represent arguments passed to a structure object in the Synchronize call. Publication names, a script version, and the MobiLink server host name and port number are included in instance variables of the nonvisual user object.

For all synchronization wizards When you run the wizard, the values that you specify for these instance variables are set as default values in the script for the constructor event of the user object. They are also set in the Windows registry on the development computer in *HKEY_CURRENT_USER\Software\Sybase\PocketBuilder\2.0\appname\MobiLink*, where *appname* is the name of your application.

At runtime, the constructor event script gets the values of the instance variables from the Windows CE registry on the device. If they cannot be obtained from the registry, or if you override the registry settings, the default value supplied in the script is used instead and is written to the registry.

You can change the default values in the event script, and you can let the user change the registry values at runtime by providing a menu item that opens the w_*appname*_sync_options or the w_*appname*_ulsync_options window.

The user object's uf_runsync and uf_runsync_with_window functions use the instance variables as arguments when they launch a dbmlsync process or Synchronize call.

Launching synchronization

To enable the user to launch a synchronization process, code a button or menu event script to call the gf_*appname*_sync (SQL Anywhere) or gf_*appname*_ulsync (UltraLite) global function. This function creates an instance of the nvo_*appname*_sync or nvo_*appname*_ulsync user object, and the user object's constructor event script sets the *appname\MobiLink* key in the Windows CE registry.

If you specified in the wizard that the status window should display, the global function opens the status window, whose ue_postopen event calls the uf_runsync_with_window function; otherwise, the global function calls the uf_runsync function. Both uf_runsync functions launch dbmlsync for remote SQL Anywhere databases as an external process using a special function in the PocketBuilder VM. For UltraLite databases, both functions call Synchronize on the connection object.

Supplying a MobiLink user name and password

The gf_*appname*_sync (SQL Anywhere) global function takes a MobiLink user name and password as arguments. The gf_*appname*_ulsync (UltraLite) global function also takes the name of the remote database connection object as an argument. The wizard does not set any default values for these arguments, so you generally need to provide them. If you pass valid arguments to the function, it sets the value of the *is_mluser* and *is_mlpassword* instance variables to the values supplied.

Example for a remote SQL Anywhere database You could code a menu item to open a response window with two single-line edit boxes, and pass the user-supplied values to the function in the script for an OK button:

```
if gf_myapp_sync (sle_usr.text, sle_pwd.text)<> 0 then
   MessageBox("Error", "MobiLink Error")
end if
```

If you pass null values or empty strings to the global function, the uf_runsync functions use MobiLink user name and password values stored in the registry to provide arguments for the dbmlsync utility. If each user of your application typically uses a given MobiLink user name, providing a mechanism that stores values in the registry lets users start a synchronization without reentering the information. The options window (described in "Using the synchronization options window" on page 266) provides such a mechanism.

If no user name is supplied
If there are no values in the registry and the publication has only one user associated with it, you can supply empty arguments to the global function, and dbmlsync will use the user name associated with the publication.

Example for a remote UltraLite database You could code a menu item to open a response window with two single-line edit boxes, and pass the user-supplied values to the function in the script for an OK button:

```
if gf_myapp_ulsync (sle_usr.text, sle_pwd.text, sqlca)&
   <> 0 then
   MessageBox("Error", "MobiLink Error")
end if
```

If you pass null values or empty strings to the global function, the uf_runsync functions use MobiLink user name and password values stored in the registry to provide arguments to the structure object passed in the Synchronize call.

Retrieving data after synchronization

After synchronizing, you would typically test for synchronization errors, then retrieve data from the newly synchronized database. For example, for synchronization involving a remote SQL Anywhere database, you could code:

```
if gf_myapp_sync("", "") <> 0 then
   MessageBox("Error", "MobiLink error")
else
   dw_1.Retrieve()
end if
```

Capturing dbmlsync messages

For synchronization with remote SQL Anywhere databases, the PocketBuilder VM traps messages from the dbmlsync process and triggers events in the nonvisual user object as the process runs. For remote UltraLite databases, synchronization messages are caught in a structure object, and the synchronization directly triggers events in the nonvisual user object generated by the UltraLite Synchronization wizard.

These events are triggered before synchronization begins as the upload stream is prepared:

ue_begin_logscan (long *rescan_log*)
ue_progress_info (long *progress_index*, long *progress_max*)
ue_end_logscan ( )

These events correspond to events on the synchronization server, as described in "Connection events" on page 270:

ue_begin_sync (string *user_name*, string *pub_names*)
ue_connect_MobiLink ( )
ue_begin_upload ( )
ue_end_upload ( )
ue_begin_download ( )
ue_end_download (long *upsert_rows*, long *delete_rows*)
ue_disconnect_MobiLink( )
ue_end_sync (long *status_code*)

These events are triggered after ue_end_upload and before ue_begin_download:

ue_wait_for_upload_ack ( )
ue_upload_ack (long *upload_status*)

These events are triggered when various messages are sent by the server:

ue_error_msg (string *error_msg*)
ue_warning_msg (string *warning_msg*)
ue_file_msg (string *file_msg*)
ue_display_msg (string *display_msg*)

The default event scripts created by a PocketBuilder synchronization wizard trigger corresponding events in the optional status window, if it exists. The window events write the status to the multiline edit control in the status window. Some window events also update a static text control that displays the phase of the synchronization operation that is currently running (log scan, upload, or download) and control a horizontal progress bar showing what percentage of the operation has completed.

You can also add code to the user object or window events that will execute at the point in the synchronization process when the corresponding MobiLink events are triggered. For synchronization with remote SQL Anywhere databases, the dbmlsync process sends the event messages to the controlling PowerBuilder application and waits until PowerBuilder event processing is completed before continuing. There is no external synchronization process called by an application for synchronizing a remote UltraLite database.

Cancelling synchronization

The Cancel button on the status window calls the uf_cancelsync user object function to cancel the synchronization process. If your application does not use the status window, you can call this function in an event script elsewhere in your application.

Using the synchronization options window

To use the options window (w_*appname*_sync_options for applications using remote SQL Anywhere databases or w_*appname*_sync_options for applications using remote UltraLite databases), code a menu item or button clicked event to call the gf_*appname*_configure_sync or the gf_*appname*_configure_ulsync function. This function creates an instance of the s_*appname*_sync_parms or s_*appname*_ulsync_parms structure and passes it to the options window.

The window's Open event creates an instance of the nvo_*appname*_sync or nvo_*appname*_ulsync user object, and its ue_postopen event retrieves values from the registry to populate the text boxes in the window—unless you have chosen to override registry settings. The user can verify or modify options in the window and click either OK or Cancel.

If the user clicks OK, the gf_*appname*_configure_sync or gf_*appname*_configure_ulsync function calls gf_*appname*_sync or gf_*appname*_ulsync to launch synchronization using the MobiLink user name and password returned from the window. The user's changes are also written to the registry.

The Close event of the window calls the wf_try_saving window function. If the user clicks OK, the wf_savesettings window function is launched. If the user clicks Cancel, no changes are made to the registry.

The options window has three pages: Subscriptions, MobiLink Server, and Settings. Although UltraLite databases do not have subscriptions, the information associating publications with a script version and a MobiLink user is the same type of information used to define subscriptions in SQL Anywhere databases.

Subscriptions page

When you used the MobiLink wizard, you selected one or more publications from the list of available publications. By default, the selected publications that display on the Subscriptions page cannot be edited at runtime for an application using a remote SQL Anywhere database. You can enable the Publications field for user entry at runtime by opening the options window in the Window painter, selecting the Publications text box, and clicking the Enabled check box in the Properties view.

Each remote user can supply a MobiLink synchronization user name on the Subscriptions page. For remote SQL Anywhere databases, the name must be associated in a subscription with the publications displayed on the page. If the application is always used by the same MobiLink user, this information never needs to be supplied again. The name is saved in the registry and used by default every time synchronization is launched from the application on this device.

If the user checks the Remember Password check box, the password is encrypted and saved in the registry. The uf_encrypt_pw and uf_decrypt_pw functions use a simple algorithm to ensure that the password does not display without encryption in the registry. You can replace this algorithm with a more sophisticated encryption technique.

MobiLink Server page

When you create a subscription, you specify a protocol, host, port, and other connection options. For ease of testing, the default protocol is TCP/IP and the default host is localhost. The default port is 2439 for TCP/IP, 80 for HTTP, and 443 for HTTPS.

You might need to change these defaults when you are testing, and your users might need to change them when your application is in use if the server is moved to another host or the port changes. For remote UltraLite database connections, the application user can enter additional parameters in the Additional text box in the format: `keyword1=value1[;keywordN=valueN]`, where keyword is the parameter name and value is the parameter value. For remote SQL Anywhere databases, additional and extended parameters for the MobiLink server can be set by the user on the Settings page.

If the user does not make any changes to the MobiLink Server page of the options window, the synchronization process uses the values you entered in the wizard, if any. For remote SQL Anywhere databases, if you did not enter values in the wizard, dbmlsync uses the values in the subscription.

For more information about subscriptions, see “Adding subscriptions for remote SQL Anywhere databases” on page 282.

Settings page

For remote SQL Anywhere databases The Settings page displays the file DSN, logging options, and any other dbmlsync options you specified in the wizard. It also shows the three display options available to the user. This page lets the user change any of these options.

Extended options
Extended options are added to the dbmlsync command line with the -e switch. You do not need to type the -e switch in the text box.

For remote UltraLite databases The Settings page lets the user choose to display the generated status window, to send column names (for automatic generation of synchronization scripts), and to add or change authentication parameters.

Modifying generated objects

If you want to give or restrict user access to synchronization options available in the options window, modify the window at design time or use it as a template for your own options window. At a minimum, you probably need to provide a way for each user to enter a MobiLink user name and password.

If you want the user to be able to save options without launching a synchronization, you could comment out the lines in gf_*appname*_configure_sync or gf_*appname*_configure_ulsync that call the global synchronization function (gf_*appname*_sync or gf_*appname*_ulsync), or add a third button called Save Only that contains the same code as the OK button, but returns a non-zero value.

Preparing to use the wizard for remote SQL Anywhere databases

The previous sections described how to try out the wizard in a test application and how to use the objects generated by the wizard. Before you use the MobiLink Synchronization for ASA wizard in a production application, you need to complete the following tasks:

- Set up a consolidated database and write synchronization scripts as described in “Preparing consolidated databases” on page 270
- Create a remote database on the desktop and set up one or more publications, users, and subscriptions as described in “Creating remote databases” on page 277
- Create a file DSN for the remote database, as described in “Defining the SQL Anywhere data source” on page 202
- Create a database connection profile for the remote database, as described in “Creating database profiles” on page 183

Preparing to use the wizard for remote UltraLite databases

Before you use the UltraLite Synchronization wizard in a production application, you need to complete the tasks listed in Table 17-1.

Table 17-1: Preparing databases for MobiLink synchronization

Preparation step	How to do this
Add tables and publications to the remote UltraLite database	For UltraLite 9, you can use the ulinit command line utiltiy to create a remote database using the schema from a consolidated SQL Anywhere database. Alternatively, you can start the UltraLite Schema Painter from the PocketBuilder Database painter, select File>New> UltraLite Schema, type a name with a USM extension for the schema, and click OK. Then you can select the Tables and Synchronization nodes under the new schema and click the items in the right pane of the painter to add tables, and optionally, publications. For UltraLite 10 you can create the database based on the schema of the consolidated database using the Create Synchronization Model wizard of the MobiLink plug-in to Sybase Central. You can also create an empty database and add tables to it using the UltraLite plug-in to Sybase Central..
Add MobiLink users, script versions, and synchronized tables in the consolidated database	Connect to the consolidated database using a MobiLink Synchronization connection in Sybase Central, add new users under the Users node, add script versions under the Versions node, add tables under the Synchronized Tables node, then add scripts for each synchronized table by selecting each table sequentially and clicking Add Table Script in the right pane. The table script associates a script version with a scripting event for each table. The above description is for the Admin mode of the MobiLink 10 plug-in to Sybase Central 5.0. This is the only mode available for MobiLink 9. For MobiLink 10, you can use the plug-in Mode menu to switch between Admin and Model modes. **Automatic addition of users and scripts** Adding users and script versions is optional for non-production databases, because you can add these automatically using special selections in the wizard or by modifying objects created by the wizard.
For UltraLite 9, generate the UltraLite schema as a remote UltraLite database	Start the Create UltraLite 9.x Database utility in PocketBuilder, browse to a USM file you created with the UltraLite 9 Schema Painter, enter a name for a new UltraLite database, and click the OK button.

Preparing consolidated databases

Whether you are designing a new database or preparing an existing one to be used as a MobiLink consolidated database, you must install the MobiLink system tables in that database. MobiLink setup scripts for SQL Anywhere 10, Adaptive Server Enterprise, Oracle 8 and 9, Microsoft SQL Server, and IBM DB2 databases are located in the *MobiLink\setup* directory of your SQL Anywhere installation. (Setup scripts are not required for Adaptive Server Anywhere 9, but are required for SQL Anywhere 10 consolidated databases.)

MobiLink system tables store information for MobiLink users, tables, scripts, and script versions in the consolidated database. You will probably not directly access these tables, but you alter them when you perform actions such as adding synchronization scripts.

ODBC connections and drivers

To carry out synchronization, the MobiLink synchronization server needs an ODBC connection to the consolidated database. You must have an ODBC driver for your server and you must create an ODBC data source for the database on the machine on which your MobiLink synchronization server is running. For a list of supported drivers, see Recommended ODBC Drivers for MobiLink at http://www.sybase.com/detail?id=1011880.

Writing synchronization scripts

There are two types of events that occur during synchronization and for which you need to write synchronization scripts:

- Connection events that perform global tasks required during every synchronization
- Table events that are associated with a specific table and perform tasks related to modifying data in that table

Connection events

At the connection level, the sequence of major events is as follows:

```
begin_connection
   begin_synchronization
      begin_upload
      end_upload
      prepare_for_download
      begin_download
      end_download
   end_synchronization
end_connection
```

When a synchronization request occurs, the begin_connection event is fired. When all synchronization requests for the current script version have been completed, the end_connection event is fired. Typically you place initialization and cleanup code in the scripts for these events, such as variable declaration and database cleanup.

Apart from begin_connection and end_connection, all of these events take the MobiLink user name stored in the ml_user table in the consolidated database as a parameter. You can use parameters in your scripts by placing question marks where the parameter value should be substituted.

To make scripts in SQL Anywhere databases easier to read, you might declare a variable in the begin_connection script, then set it to the value of ml_username in the begin_synchronization script.

For example, in begin_connection:

```
CREATE VARIABLE @sync_user VARCHAR(128);
```

In begin_synchronization:

```
SET @sync_user = ?
```

The begin_synchronization and end_synchronization events are fired before and after changes are applied to the remote and consolidated databases.

The begin_upload event marks the beginning of the upload transaction. Applicable inserts and updates to the consolidated database are performed for all remote tables, then rows are deleted as applicable for all remote tables. After end_upload, upload changes are committed.

If you do not want to delete rows from the consolidated database, do not write scripts for the upload_delete event, or use the STOP SYNCHRONIZATION DELETE statement in your PowerScript code. For more information, see “Deleting rows from the remote database only” on page 285.

The begin_download event marks the beginning of the download transaction. Applicable deletes are performed for all remote tables, and then rows are added as applicable for all remote tables in the download_cursor. After end_download, download changes are committed. These events have the date of the last download as a parameter.

Other connection-level events can also occur, such as handle_error, report_error, and synchronization_statistics. For a complete list of events and examples of their use, see the chapter on synchronization events in the *MobiLink Administration Guide*.

Table events

Many of the connection events that occur between the begin_synchronization and end_synchronization events, such as begin_download and end_upload, also have table equivalents. These and other overall table events might be used for tasks such as creating an intermediate table to hold changes or printing information to a log file.

You can also script table events that apply to each row in the table. For row-level events, the order of the columns in your scripts must match the order in which they appear in the CREATE TABLE statement in the remote database, and the column names in the scripts must refer to the column names in the consolidated database.

Generating default scripts

Although there are several row-level events, most tables need scripts for three upload events (for INSERT, UPDATE, and DELETE) and one download event. To speed up the task of creating these four scripts for every table, you can generate scripts for them automatically by running the "create a synchronization model" task from the MobiLink 10 plug-in in Sybase Central.

For information on the MobiLink plug-in, see the online *MobiLink Getting Started* book.

The MobiLink plug-in allows you to add more functionality to default scripts than default scripts generated in earlier versions of MobiLink. However, if you are using a remote Adaptive Server Anywhere 9 database (instead of a remote SQL Anywhere 10 database), you can still generate default synchronization scripts by starting the MobiLink synchronization server with the -za switch and setting the SendColumnNames extended option for dbmlsync. For applications using a remote UltraLite 9.x database, you (or an application user) can select a send Column Names check box.

Read-only remote databases
If the remote Adaptive Server Anywhere 9 database is read-only—that is, you never want to upload any data—you should not implement the upload scripts. You can use the -ze switch to generate sample scripts, and use the download samples as templates for your download scripts.

❖ **To generate synchronization scripts automatically in PocketBuilder:**

1. Select the Automatic Script Generation check box in the MobiLink Synchronize Server Options dialog box and click OK to start the server.

   You open this dialog box from the Utilities folder in the Database painter or the Database Profiles dialog box.

2 In an application using a remote Adaptive Server Anywhere database, enter `SendColumnNames=ON` in the Extended text box on the Settings page of the w_*appname*_sync_options window.
or
In an application using a remote UltraLite database, select the Send Column Names check box on the Settings page of the w_*appname*_ulsync_options window.

You must have at least one publication and user defined in the remote database. For a remote Adaptive Server Anywhere database, you must also have at least one subscription defined in the database. If you have more than one publication or user, you must use the -n and/or -u switches to specify which subscription you want to work with.

If there are existing scripts in the consolidated database, MobiLink does nothing. If there are no existing scripts, MobiLink generates them for all tables specified in the publication. The scripts control the upload and download of data to and from your client and consolidated databases.

If the column names on the remote and consolidated database differ, the generated scripts must be modified to match the names on the consolidated database.

You can also generate synchronization scripts from a command prompt. Start the server using the -za switch, then run dbmlsync and set the SendColumnNames extended option to on. For example:

```
dbmlsrv9 -c "dsn=masterdb" -za
dbmlsync -c "dsn=remotedb" -e SendColumnNames=ON
```

Generated scripts

Table 17-2 shows the scripts that are generated for a table named emp with the columns emp_id, emp_name, and dept_id. The primary key is emp_id.

Table 17-2: Sample default scripts generated by dbmlsrv9 -za

Script name	Script
upload_insert	`INSERT INTO emp (emp_id, emp_name, dept_id) VALUES (?,?,?)`
upload_update	`UPDATE emp SET emp_name = ?, dept_id = ? WHERE emp_id=?`
upload_delete	`DELETE FROM emp WHERE emp_id=?`
download_cursor	`SELECT emp_id, emp_name, dept_id FROM emp`

The scripts generated for downloading data perform "snapshot" synchronization. A complete image of the table is downloaded to the remote database. Typically you need to edit these scripts to limit the data transferred. For more information, see "Limiting data downloads" on page 283.

Before modifying any scripts, you should test the synchronization process to make sure that the generated scripts behave as expected. Performing a test after each modification will help you narrow down errors.

Working with scripts and users in Sybase Central

You can view and modify existing scripts and write new ones in the MobiLink Synchronization plug-in in Sybase Central. These procedures describe how to connect to the plug-in and write scripts, and how to add a user to the consolidated database.

❖ **To connect to a consolidated database in Sybase Central:**

1. Start Sybase Central, select Tools>Connect from the menu bar, select MobiLink Synchronization from the New Connection dialog box, and click OK.
2. On the Identification page in the Connect dialog box, select ODBC DataSource Name, browse to select the DSN of the consolidated database, and click OK.

When you expand the node for a consolidated database in the MobiLink Synchronization plug-in, you see five folders: Tables, Connection Scripts, Synchronized Tables, Users, and Versions. All the procedures in this section begin by opening one of these folders.

Script versions

Scripts are organized into groups called script versions. By specifying a particular version, MobiLink clients can select which set of synchronization scripts is used to process the upload stream and prepare the download stream. If you want to define different versions for scripts, you must add a script version to the consolidated database before you add scripts for it.

If you create two different versions, make sure that you have scripts for all required events in both versions.

Global script version
For MobiLink 10 and later, you can create a script version called ml_global that is used differently from other script versions. If you create a script version called ml_global, you define it once and then the connection scripts associated with it are used by default in all synchronizations. You never explicitly specify ml_global as a script version.

The ml_global script version can contain connection-level scripts only. For more information, see the *MobiLink Server Administration* book.

❖ **To add a script version:**

1 Select the Versions folder and double-click Add Version.

2 In the Add a New Script Version dialog box, provide a name for the version and optionally a description, and click Finish.

Sybase Central creates the new version and gives it a unique integer identifier.

Adding scripts

Scripts added for connection events are executed for every synchronization. Scripts added for table events are executed when a specific table has been modified. You must specify that a table is synchronized before you can add scripts for it.

❖ **To add a synchronized table to a consolidated database:**

1 Select the Tables folder and double-click DBA.

2 Right-click the table you want to add to the list of synchronized tables and select Add to Synchronized Tables from its pop-up menu.

❖ **To add a script to a synchronized table:**

1 Select the Synchronized Tables folder, select the table for which you want to add a script, and double-click Add Table Script.

2 From the first drop-down list, select the version for which you want to add a script.

3 From the second drop-down list, select the event for which you want to add a script.

Events that already have a script do not appear in the drop-down list.

4 From the third drop-down list, select the language in which you want to write a script.

5 Make sure the Edit the Script of the New Event Immediately check box is selected and click Finish.

6 Type your script in the editor that displays, then save and close the file.

For example, if you want to remove rows that have been shipped from the Order table in a remote database, you can place the following SELECT statement in the download_delete_cursor event, where order_id is the primary key column. The first parameter to this event is the last_download timestamp. It is used here to supply the value for a last_modified column:

```
SELECT order_id
  FROM Order
WHERE status = 'Shipped'
  AND last_modified >= ?
```

For more information about using the download_delete_cursor event, see the section on "Writing download_delete_cursor scripts" in the *MobiLink Server Administration* book.

❖ **To add a connection-level script:**

1 Select the Connection Scripts folder and double-click Add Connection Script.

2 Follow steps 2 to 6 in the previous procedure.

Modifying scripts

To modify an existing script, navigate to the script in Sybase Central as described in the preceding procedures, then double-click the Edit icon to the left of the version name.

Adding users

You can add users directly to the ml_user table in the consolidated database, then provide the user names and optional passwords to your users. To add a user, select the Users folder, double-click Add User, and complete the Add User wizard.

You also have to add at least one user name to each remote database, as described in "Creating MobiLink users" on page 280.

Creating remote databases

Creating a remote SQL Anywhere database

Any SQL Anywhere database can be converted for use as a remote database in a MobiLink installation. You can also create a new SQL Anywhere remote database that uses all or part of the schema of the consolidated SQL Anywhere database.

You create the database on your desktop using the Sybase Central SQL Anywhere plug-in, the Create ASA Database utility in the Database painter, or another tool. If your database uses an English character set, use the 1252 Latin1 collation sequence. If Sybase Central detects that Microsoft ActiveSync is installed on your computer, it enables a wizard page that lets you set up the database for use on Windows CE.

Creating a remote UltraLite database

You can create a remote UltraLite database by starting the UltraLite Schema Painter from the Utilities folder in the PocketBuilder Database Profile painter. After you create a USM database schema, you can start another utility in the Utilities folder that converts the schema into an UltraLite database. The Create UltraLite Database utility consists of a single dialog box that prompts you for the USM name and the name of the database you want to create.

For more information on using the UltraLite Schema Painter, see the *UltraLite Database User's Guide* or the online Help. For a description of options in the Create UltraLite Database utility, see the dialog box Help.

Preparing and deploying the remote database

To use a database as a remote database for MobiLink synchronization, you need to create at least one publication and MobiLink user. For a remote SQL Anywhere database, you must also add a subscription to the publication. See "Creating and modifying publications" on page 278, "Creating MobiLink users" on page 280, and "Adding subscriptions for remote SQL Anywhere databases" on page 282.

To copy the database to the Pocket PC or emulator, select the Explore button in Microsoft ActiveSync, or use the Windows CE Remote File Viewer (cefilevw) for older emulators. For more information about copying files to a Pocket PC device or emulator, see the *Installation Guide*.

Remote database schemas

Tables in a remote database need not be identical to those in the consolidated database, but you can often simplify your design by using a table structure in the remote database that is a subset of the one in the consolidated database. Using this method ensures that every table in the remote database exists in the consolidated database. Corresponding tables have the same structure and foreign key relationships as those in the consolidated database.

Tables in the consolidated database frequently contain extra columns that are not synchronized. Extra columns can even aid synchronization. For example, a timestamp column can identify new or updated rows in the consolidated database. In other cases, extra columns or tables in the consolidated database might hold information that is not required at remote sites.

Creating and modifying publications

You create publications using Sybase Central or the SQL CREATE PUBLICATION statement. In Sybase Central, all publications and articles appear in the Publications folder. This section describes how to create publications in Sybase Central. For information about creating and modifying publications using SQL, see the *MobiLink - Client Administration* book.

Connecting to Sybase Central

You use the SQL Anywhere plug-in in Sybase Central, not the MobiLink Synchronization plug-in, to work with MobiLink clients and remote databases. The SQL Anywhere plug-in has a MobiLink Synchronization Client folder where you perform all actions related to remote databases.

You must have DBA authority to create or modify publications, MobiLink users, and subscriptions.

❖ **To connect to Sybase Central to work with MobiLink Synchronization clients:**

1. Start Sybase Central, select Tools>Connect from the menu bar, select SQL Anywhere from the New Connection dialog box, and click OK.
2. On the Identification page in the Connect dialog box, enter DBA as the user name and SQL as the password, select the ODBC DataSource Name radio button, browse to select the DSN of the remote database, and click OK.
3. In the SQL Anywhere plug-in, expand the node for the remote database and open the MobiLink Synchronization Client folder.

Publishing all the rows and columns in a table

The simplest publication you can make is a single article that consists of all rows and columns of one or more tables. The tables must already exist.

❖ **To publish one or more entire tables in Sybase Central:**

1. Connect to Sybase Central as described in “Connecting to Sybase Central” on page 278.
2. Open the Publications folder and double-click Add Publication.
3. Type a name for the new publication and click Next.

4 On the Tables page, select a table from the list of Matching Tables and click Add.

The table appears in the list of Selected Tables on the right.

5 Optionally, add more tables. The order of the tables is not important.

6 Click Finish.

Publishing only some columns in a table

You can create a publication that contains all the rows but only some of the columns of a table.

❖ **To publish only some columns in a table in Sybase Central:**

1 Follow the first four steps of the procedure in "Publishing all the rows and columns in a table" on page 278.

2 On the Columns page, double-click the table's icon to expand the list of available columns, select each column you want to publish, and click Add.

The selected columns appear on the right.

3 Click Finish.

Publishing only some rows in a table

You can create a publication that contains some or all of the columns in a table, but only some of the rows. You do so by writing a search condition that matches only the rows you want to publish.

In MobiLink, you can use the WHERE clause to exclude the same set of rows from all subscriptions to a publication. All subscribers to the publication upload any changes to the rows that satisfy the search condition.

❖ **To create a publication using a WHERE clause in Sybase Central:**

1 Follow the first four steps of the procedure in "Publishing all the rows and columns in a table" on page 278.

2 On the Where page, select the table and type the search condition in the lower box.

Optionally, you can use the Insert dialog box to help you format the search condition.

3 Click Finish.

Adding articles

You can add articles to existing publications.

❖ **To add articles in Sybase Central:**

1 Connect to Sybase Central and open the MobiLink Synchronization Client folder as described in "Connecting to Sybase Central" on page 278.

2 Open the Publications folder and double-click the name of the publication to which you want to add an article.

3 Double-click Add Article.

4 In the Article Creation wizard, select a table and click Next.

5 If you want only some columns to be synchronized, select the Selected Columns radio button and select the columns.

6 If you want to add a WHERE clause, click Next and enter the clause.

7 Click Finish.

Modifying and removing publications and articles

You can modify or drop existing publications in Sybase Central by navigating to the location of the publication and selecting Properties or Delete from its pop-up menu. You can modify and remove articles in the same way.

Publications can be modified only by the DBA or the publication's owner. You must have DBA authority to drop a publication. If you drop a publication, all subscriptions to that publication are automatically deleted as well.

Avoid altering publications in a running MobiLink setup
Altering publications in a running MobiLink setup is likely to cause replication errors and can lead to loss of data unless carried out with care.

Creating MobiLink users

MobiLink users are not the same as database users. Each type of user resides in a different namespace. MobiLink user IDs can match the names of database users, but there is no requirement that they match.

❖ **To add a MobiLink user to a remote database in Sybase Central:**

1 Connect to Sybase Central and open the MobiLink Synchronization Client folder as described in “Connecting to Sybase Central” on page 278.

2 Open the MobiLink Users folder and double-click Add MobiLink User.

3 Enter a name for the MobiLink user.

The name is supplied to the MobiLink synchronization server during synchronization. In production databases, each user name is usually added to the consolidated database, then provided to the individual user.

4 Click Finish.

❖ **To configure MobiLink user properties in Sybase Central:**

1 Connect to Sybase Central and open the MobiLink Synchronization Client folder as described in "Connecting to Sybase Central" on page 278.

2 Open the MobiLink Users folder, right-click the MobiLink user, and select Properties from the pop-up menu.

3 Change the properties as needed.

❖ **To drop a MobiLink user in Sybase Central:**

1 Connect to Sybase Central and open the MobiLink Synchronization Client folder as described in "Connecting to Sybase Central" on page 278.

2 Open the MobiLink Users folder, right-click the MobiLink user, and select Delete from the pop-up menu.

Dropping MobiLink users
You must drop all subscriptions for a MobiLink user before you drop the user from a remote database.

Adding MobiLink users to the consolidated database

The consolidated database contains a table called ml_user that is used to authenticate the names of MobiLink users when a synchronization is requested. When you add a user to a remote database, you need to be sure that the user is also added to the ml_user table.

You can add users automatically by selecting the Automatic Addition of Users check box in the MobiLink Synchronization Server Options dialog box and then starting the server. You open this dialog box from the Utilities folder in the Database painter or Database Profiles dialog box. You can also start the server from a command prompt, passing it the -zu+ switch.

Any users defined in the remote database are added to the ml_user table in the consolidated database, as long as the script for the authenticate_user connection event is undefined. Usually the -zu+ switch should not be used in a production environment. Names are usually added to the ml_user table in the consolidated database, then added to each of the remote databases. Each user is given a unique name and optional password.

Adding subscriptions for remote SQL Anywhere databases

A synchronization subscription links a particular MobiLink user with a publication. It can also carry other information needed for synchronization. For example, you can specify the address of the MobiLink server and other connection options. Values for a specific subscription override those set for individual MobiLink users.

Overriding options in the wizard
You can override the MobiLink server name and port set for the subscription and user in the MobiLink Synchronization for ASA wizard.

Synchronization subscriptions are required in MobiLink SQL Anywhere remote databases. Server logic is implemented through synchronization scripts, stored in the MobiLink system tables in the consolidated database.

A single SQL Anywhere database can synchronize with more than one MobiLink synchronization server. To allow synchronization with multiple servers, create different subscriptions for each server.

❖ **To add a subscription for a MobiLink user in Sybase Central:**

1. Connect to Sybase Central and open the MobiLink Synchronization Client folder as described in “Connecting to Sybase Central” on page 278.
2. For Sybase Central 4.x, open the MobiLink Users folder, right-click the user for whom you want to add a subscription, and select Properties from the pop-up menu.

   For Sybase Central 5.x, open the Publications folder, select the publication for which you want to enter a subscription, select the Synchronization Subscriptions tab in the right pane of Sybase Central, then select File>New>Synchronization Subscription from the menu bar.
3. For Sybase Central 4.x, click the Subscribe button on the Subscriptions page, select the Publication for which you want to add a subscription, and click OK.

   For Sybase Central 5.x, in the Create Synchronization Subscription wizard, select the user for whom you want to enter a subscription and click Finish.

❖ **To modify a subscription in Sybase Central:**

1. Connect to Sybase Central and open the MobiLink Synchronization Client folder as described in “Connecting to Sybase Central” on page 278.

2 For Sybase Central 4.x, open the MobiLink Users folder, right-click the MobiLink user, select Properties from the pop-up menu, then on the Subscriptions page, select the subscription you want to change and click Advanced.

For Sybase Central 5.x, open the MobiLink Users folder, double-click the name of the MobiLink user who owns the subscription you want to modify, then on the Synchronization Subscriptions tab, right-click the subscription you want to modify and select Properties from the pop-up menu.

3 Change the properties as needed.

❖ **To delete a synchronization subscription in Sybase Central:**

1 Connect to Sybase Central and open the MobiLink Synchronization Client folder as described in "Connecting to Sybase Central" on page 278.

2 For Sybase Central 4.x, open the MobiLink Users folder, right-click the MobiLink user, select Properties from the pop-up menu, the on the Subscriptions page, select the subscription you want to delete and click Unsubscribe.

For Sybase Central 5.x, open the MobiLink Users folder, double-click the name of the MobiLink user who owns the subscription you want to delete, then on the Synchronization Subscriptions tab, right-click the subscription you want to delete and click Delete.

3 Click Yes in the Confirm Delete dialog box.

Synchronization techniques

This section highlights some issues that you need to consider when designing an application that uses MobiLink synchronization.

Limiting data downloads

One of the major goals of synchronization is to increase the speed and efficiency of data movement by restricting the amount of data moved. To limit the data transferred by the download_cursor script, you can partition data based on its timestamp, the MobiLink user name, or both.

Timestamp partitioning One way to limit downloads to data changed since the last download is to add a last_modified column to each table in the consolidated database (or, if the table itself cannot be changed, to a shadow table that holds the primary key and that is joined to the original table in the download_cursor script). The last_modified column need only be added to the consolidated database.

In SQL Anywhere, you can use built-in DEFAULT TIMESTAMP datatypes for this column. In other DBMSs, you need to provide an update trigger to set the timestamp of the last_modified column.

The timestamp is generated on the consolidated database and downloaded unmodified to the remote database during synchronization; the time zone of the remote database does not affect it.

User-based partitioning The download_cursor script has two parameters: last_download, of datatype datetime, and ml_username, of type varchar(128). You can use these parameters to restrict the download not only to rows that have changed since the last synchronization, but also to rows that belong to the current user.

In this sample download_cursor script, only those rows are downloaded that have been modified since the last synchronization, and that apply to the sales representative whose ID matches the MobiLink user ID:

```
SELECT order_id, cust_id, order_date
   FROM Sales_Order
WHERE last_modified >= ?
   AND sales_rep = ?
```

For this to work correctly, the MobiLink user ID must match the sales_rep ID. If this is not the case, you might need to join a table that associates these two IDs.

Primary key uniqueness

In a conventional client/server environment where clients are always connected, referential integrity is directly imposed. In a mobile environment, you must ensure that primary keys are unique and that they are never updated. There are several techniques for achieving this, such as using primary key pools.

Handling conflicts

You need to handle conflicts that arise when, for example, two remote users update the same rows but synchronize at different intervals, so that the latest synchronization might not be the latest update. MobiLink provides mechanisms to detect and resolve conflicts.

Deleting rows from the remote database only

By default, when a user starts a synchronization, the net result of all the changes made to the database since the last synchronization is uploaded to the consolidated database. However, sometimes a remote user deletes certain rows from the remote database to recapture space, perhaps because the data is old or a customer has transferred to another sales agent. Usually, those deleted rows should not be deleted from the consolidated database.

One way to handle this is to use the command `STOP SYNCHRONIZATION DELETE` in a script in your PocketBuilder application to hide the SQL DELETE statements that follow it from the transaction log. None of the subsequent DELETE operations on the connection will be synchronized until the `START SYNCHRONIZATION DELETE` statement is executed.

For example, you might provide a menu item called Delete Local where the code that handles the delete is wrapped as in this example:

```
STOP SYNCHRONIZATION DELETE;
// call code to perform delete operation
START SYNCHRONIZATION DELETE;
COMMIT;
```

There are other approaches to handling deletes. For more information, see the chapter on synchronization techniques in the *MobiLink - Server Administration* book.

CHAPTER 18

Setting Additional Connection Parameters

About this chapter

This chapter describes how to set database parameters and database preferences in PocketBuilder to fine-tune your database connection and take advantage of DBMS-specific features.

Contents

Setting database parameters

In PocketBuilder, you can set database parameters by doing either of the following:

- Editing the Database Profile Setup dialog box for your connection in the development environment
- Specifying connection parameters in an application script

For more information about the Database Profile Setup dialog box, see "About database profiles" on page 182. For descriptions of database parameters, see the PocketBuilder *Connection Reference*.

Setting database parameters in the development environment

Editing database profiles

To set database parameters for a database connection in the PocketBuilder development environment, you must edit the database profile for that connection.

Character limit for DBParm strings

Strings containing database parameters that you specify in the Database Profile Setup dialog box for your connection can be up to 999 characters in length. No limit applies to DBParm strings that you specify in PocketBuilder scripts, as properties of the Transaction object are *not* limited to a specified length.

Setting database parameters in a PocketBuilder application script

If you are developing a PocketBuilder application that connects to a database, you must specify the required connection parameters in the appropriate script as properties of the default Transaction object (SQLCA) or a Transaction object that you create. For example, you might specify connection parameters in the script that opens the application.

One of the connection parameters you might want to specify in a script is DBParm. You can do this by:

- *(Recommended)* Copying PowerScript DBParm syntax from the Preview tab in the Database Profile Setup dialog box into your script
- Coding PowerScript to set values for the DBParm property of the Transaction object
- Reading DBParm values from an external text file

Copying DBParm syntax from the Preview tab

The easiest way to specify database parameters in a PocketBuilder application script is to copy the PowerScript DBParm syntax from the Preview tab in the Database Profile Setup dialog box into your script, modifying the default Transaction object name (SQLCA) if necessary.

As you set database parameters in the Database Profile Setup dialog box in the development environment, PocketBuilder generates the correct connection syntax on the Preview tab. Therefore, copying the syntax directly from the Preview tab ensures that you use the correct PowerScript DBParm syntax in your script.

❖ **To copy DBParm syntax from the Preview tab into your script:**

1. On one or more tab pages in the Database Profile Setup dialog box for your connection, supply values for any database parameters you want to set.
2. Click Apply to save your changes to the current tab without closing the Database Profile Setup dialog box.

3 Click the Preview tab.

The correct PowerScript DBParm syntax for each selected option displays in the Database Connection Syntax box.

4 Select one or more lines of text in the Database Connection Syntax box and click Copy.

PocketBuilder copies the selected text to the clipboard.

5 Click OK to close the Database Profile Setup dialog box.

6 Paste the selected text from the Preview tab into your script, modifying the default Transaction object name (SQLCA) if necessary.

Coding PowerScript to set values for the DBParm property

Another way to specify connection parameters in a script is by coding PowerScript to assign values to properties of the Transaction object. As explained in Chapter 16, "Using Transaction Objects," PocketBuilder uses a special nonvisual object called a Transaction object to communicate with the database. The default Transaction object is named SQLCA, which stands for SQL Communications Area.

SQLCA has 15 properties, 10 of which are used to connect to your database. One of the 10 connection properties is DBParm. DBParm contains DBMS-specific parameters that let your application take advantage of various features supported by the database interface.

❖ **To set values for the DBParm property in a PocketBuilder script:**

1 Open the application script in which you want to specify connection parameters.

2 Use the following PowerScript syntax to specify DBParm parameters. Make sure you separate the DBParm parameters with commas, and enclose the entire DBParm string in double quotes.

SQLCA.dbParm = "*parameter_1*, *parameter_2*, *parameter_n*"

For example, the following statement in a script sets the DBParm property for an ODBC data source named Sales. In this example, the DBParm property consists of two parameters: ConnectString and Async.

```
SQLCA.dbParm="ConnectString='DSN=Sales;UID=PB;
   PWD=xyz',Async=1"
```

3 Compile the script to save your changes.

Reading DBParm values from an external text file

As an alternative to setting the DBParm property in a PocketBuilder application script, you can use the PowerScript ProfileString function to read DBParm values from a specified section of an external text file, such as an application-specific initialization file.

❖ **To read DBParm values from an external text file:**

1 Open the application script in which you want to specify connection parameters.

2 Use the following PowerScript syntax to specify the ProfileString function with the SQLCA.DBParm property:

```
SQLCA.dbParm = ProfileString ( file, section, key,
    default )
```

For example, the following statement in a PocketBuilder script reads the DBParm values from the [Database] section of the *APP.INI* file:

```
SQLCA.dbParm=ProfileString("APP.INI","Database",
    "dbParm","")
```

3 Compile the script to save your changes.

Setting database preferences

The way you set connection-related database preferences in PocketBuilder varies. AutoCommit and Lock are the only database preferences that you can set in a PocketBuilder application script, and the only database preferences that you set in the Database Profile Setup dialog box for your connection. All other database preferences can be set only in the Database Preferences dialog box.

The following sections give the steps for setting database preferences in the development environment and (for AutoCommit and Lock) in a PocketBuilder application script.

For information about using a specific database preference, see the chapter on database preferences in the PocketBuilder *Connection Reference*.

Setting database preferences in the development environment

There are two ways to set database preferences in the PocketBuilder development environment on *all* supported development platforms, depending on the preference you want to set:

- Set AutoCommit and Lock (Isolation Level) in the Database Profile Setup dialog box for your connection
- Set all other database preferences in the Database Preferences dialog box in the Database painter

Setting AutoCommit and Lock in the database profile

The AutoCommit and Lock (Isolation Level) preferences are properties of the default Transaction object, SQLCA. For AutoCommit and Lock to take effect in the PocketBuilder development environment, you must specify them *before* you connect to a database. Changes made to these preferences after the connection occurs have no effect on the current connection.

To set AutoCommit and Lock before PocketBuilder connects to your database, you specify their values in the Database Profile Setup dialog box for your connection.

❖ **To set AutoCommit and Lock (Isolation Level) in a database profile:**

1 Display the Database Profiles dialog box.

2 Click the plus sign (+) to the left of the interface you are using, or double-click the interface name.

The list expands to display the database profiles defined for your interface.

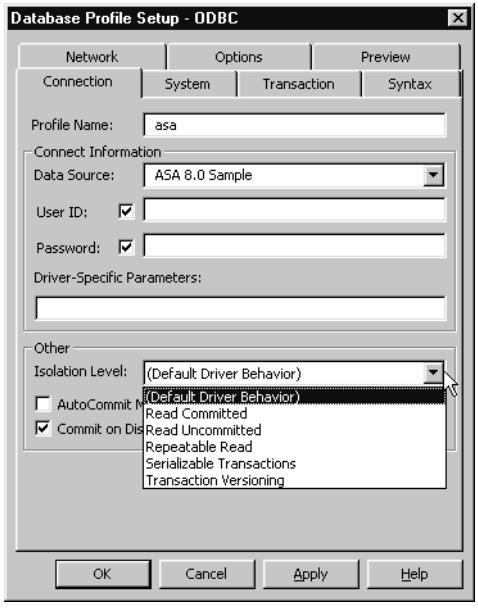
3 Select the name of the profile you want and click Edit.

The Database Profile Setup dialog box for the selected profile displays.

4 On the Connection tab page, supply values for one or both of the following:

- **Isolation Level** Select the isolation level you want to use for this connection from the Isolation Level drop-down list. (The Isolation Level drop-down list contains valid lock values for your interface.)
- **AutoCommit Mode** The setting of AutoCommit controls whether PocketBuilder issues SQL statements outside (True) or inside (False) the scope of a transaction. Select the AutoCommit Mode check box to set AutoCommit to True or clear the AutoCommit Mode check box (the default) to set AutoCommit to False.

Figure 18-1: Connection page showing Isolation Level settings



5 (Optional) Click the Preview tab if you want to see the PowerScript connection syntax generated for Lock and AutoCommit.

PocketBuilder generates correct PowerScript connection syntax for each option you set in the Database Profile Setup dialog box. You can copy this syntax directly into a PocketBuilder application script.

For instructions, see “Copying DBParm syntax from the Preview tab” on page 288.

6 Click OK to close the Database Profile Setup dialog box.

PocketBuilder saves your settings in the database profile entry in the registry.

Setting preferences in the Database Preferences dialog box

To set the following connection-related database preferences, complete the Database Preferences dialog box in the PocketBuilder Database painter:

- Shared Database Profiles
- Connect to Default Profile
- Read Only

- Keep Connection Open
- Use Extended Attributes
- SQL Terminator Character

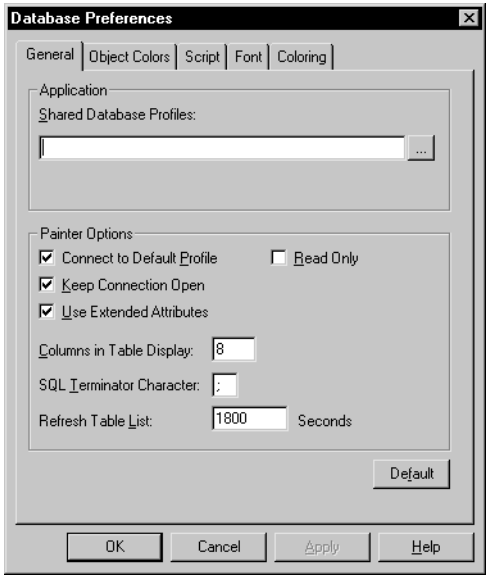
Other database preferences
The Database Preferences dialog box also lets you set other database preferences that affect the behavior of the Database painter itself. For information about the other preferences you can set in the Database Preferences dialog box, see the *User's Guide*.

❖ **To set connection-related preferences in the Database Preferences dialog box:**

1 Open the Database painter.

2 Select Design>Options from the menu bar.

The Database Preferences dialog box displays. If necessary, click the General tab to display the General property page.



3 Specify values for one or more of the connection-related database preferences in the following table.

Table 18-1: Connection-related database preferences

Preference	Description	For details, see
Shared Database Profiles	Specifies the path name of the file containing the database profiles you want to share. You can type the path name or click Browse to display it.	"Sharing database profiles" on page 191 and the Shared Database Profiles database preference in the PocketBuilder *Connection Reference*
Connect to Default Profile	Controls whether the Database painter establishes a connection to a database using a default profile when the painter is invoked. If not selected, the Database painter opens without establishing a connection to a database.	The Connect to Default Profile database preference in the PocketBuilder *Connection Reference*
Read Only	Specifies whether PocketBuilder should update the extended attribute system tables and any other tables in your database. Select or clear the Read Only check box as follows: • **Select the check box** Does not update the extended attribute system tables or any other tables in your database. You *cannot* modify (update) information in the extended attribute system tables or any other database tables from the DataWindow painter when the Read Only check box is selected. • **Clear the check box** (Default) Updates the extended attribute system tables and any other tables in your database.	The Read Only database preference in the PocketBuilder *Connection Reference*

Preference	Description	For details, see
Keep Connection Open	When you connect to a database in PocketBuilder without using a database profile, specifies when PocketBuilder closes the connection. Select or clear the Keep Connection Open check box as follows: • **Select the check box** (Default) Stays connected to the database throughout your session and closes the connection when you exit. • **Clear the check box** Opens the connection only when a painter requests it and closes the connection when you close a painter or finish compiling a script. **Not used with profile** This preference has no effect when you connect using a database profile.	The Keep Connection Open database preference in the PocketBuilder *Connection Reference*
Use Extended Attributes	Specifies whether PocketBuilder should create and use the extended attribute system tables. Select or clear the Use Extended Attributes check box as follows: • **Select the check box** (Default) Creates and uses the extended attribute system tables. • **Clear the check box** Does *not* create the extended attribute system tables.	The Use Extended Attributes database preference in the PocketBuilder *Connection Reference*
SQL Terminator Character	Specifies the SQL statement terminator character used in the ISQL view in the Database painter in PocketBuilder. The default terminator character is a semicolon (;). If you are creating stored procedures and triggers in the ISQL view of the database painter, change the terminator character to one that you do not expect to use in the stored procedure or trigger syntax for your DBMS. A good choice is the backquote (`) character.	The SQL Terminator Character database preference in the PocketBuilder *Connection Reference*

4 Do one of the following:

- Click Apply to apply the preference settings to the current connection without closing the Database Preferences dialog box
- Click OK to apply the preference settings to the current connection and close the Database Preferences dialog box

PocketBuilder saves your preference settings in the database section of *PK.INI*.

Setting AutoCommit and Lock in a PocketBuilder application script

If you are developing a PocketBuilder application that connects to a database, you must specify the required connection parameters in the appropriate script as properties of the default Transaction object (SQLCA) or a Transaction object that you create. For example, you might specify connection parameters in the script that opens the application.

AutoCommit and Lock are properties of SQLCA. As such, they are the *only* database preferences that you can set in a PocketBuilder script. You can do this by:

- *(Recommended)* Copying PowerScript syntax for AutoCommit and Lock from the Preview tab in the Database Profile Setup dialog box into your script
- Coding PowerScript to set values for the AutoCommit and Lock properties of the Transaction object
- Reading AutoCommit and Lock values from an external text file

For more about using Transaction objects to communicate with a database in a PocketBuilder application, see Chapter 16, “Using Transaction Objects.”

Copying AutoCommit and Lock syntax from the Preview tab

The easiest way to specify AutoCommit and Lock in a PocketBuilder application script is to copy the PowerScript syntax from the Preview tab in the Database Profile Setup dialog box into your script, modifying the default Transaction object name (SQLCA) if necessary.

As you complete the Database Profile Setup dialog box in the development environment, PocketBuilder generates the correct connection syntax on the Preview tab for each selected option. Therefore, copying the syntax directly from the Preview tab ensures that you use the correct PowerScript syntax in your script.

❖ **To copy AutoCommit and Lock syntax from the Preview tab into your script:**

1 On the Connection tab in the Database Profile Setup dialog box for your connection, supply values for AutoCommit and Lock (Isolation Level) as required.

For instructions, see "Setting AutoCommit and Lock in the database profile" on page 291.

2 Click Apply to save your changes to the current tab without closing the Database Profile Setup dialog box.

3 Click the Preview tab.

The correct PowerScript syntax for each selected option displays in the Database Connection Syntax box.

4 Select one or more lines of text in the Database Connection Syntax box and click Copy.

PocketBuilder copies the selected text to the clipboard.

5 Click OK to close the Database Profile Setup dialog box.

6 Paste the selected text from the Preview tab into your script, modifying the default Transaction object name (SQLCA) if necessary.

Coding PowerScript to set values for AutoCommit and Lock

Another way to specify the AutoCommit and Lock properties in a script is by coding PowerScript to assign values to the AutoCommit and Lock properties of the Transaction object.

❖ **To set the AutoCommit and Lock properties in a PocketBuilder script:**

1 Open the application script in which you want to set connection properties.

For instructions, see the *User's Guide*.

2 Use the following PowerScript syntax to set the AutoCommit and Lock properties. (This syntax assumes you are using the default Transaction object SQLCA, but you can also define your own Transaction object.)

```
SQLCA.AutoCommit = value
SQLCA.Lock = "value"
```

For more information, see AutoCommit or Lock in the online Help.

3 Compile the script to save your changes.

For instructions, see the *User's Guide*.

Reading AutoCommit and Lock values from an external text file

As an alternative to setting the AutoCommit and Lock properties in a PocketBuilder application script, you can use the PowerScript ProfileString function to read the AutoCommit and Lock values from a specified section of an external text file, such as an application-specific initialization file.

❖ **To read AutoCommit and Lock values from an external text file:**

1 Open the application script in which you want to set connection properties.

For instructions, see the *User's Guide*.

2 Use the following PowerScript syntax to specify the ProfileString function with the SQLCA.Lock property:

```
SQLCA.Lock = ProfileString ( file, section, key, default )
```

The AutoCommit property is a boolean, so you need to convert the string returned by ProfileString to a boolean. For example, the following statements in a PocketBuilder script read the AutoCommit and Lock values from the [Database] section of the *APP.INI* file:

```
string ls_string
ls_string = Upper(ProfileString("APP.INI", &
   "Database", "Autocommit",""))
if ls_string = "TRUE" then
   SQLCA.Autocommit = TRUE
else
   SQLCA.Autocommit = FALSE
end if
SQLCA.Lock=ProfileString("APP.INI","Database",
   "Lock","")
```

3 Compile the script to save your changes.

Getting values from the registry

If the AutoCommit and Lock values are stored in an application settings key in the registry, use the RegistryGet function to obtain them. For example:

```
string ls_string
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp", &
   "Autocommit", RegString!, ls_string)
if Upper(ls_string) = "TRUE" then
   SQLCA.Autocommit = TRUE
else
   SQLCA.Autocommit = FALSE
end if
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp", &
   "Lock", RegString!, ls_string)
```

PART 5

Miscellaneous Techniques

This part describes how to handle external functions, work with the Message object, and interact with the Windows registry. It also describes command-line arguments you can use with the PocketBuilder executable file.

CHAPTER 19

Working with Unicode

About this chapter

PocketBuilder uses the Unicode character set. This chapter describes how PocketBuilder handles Unicode and ANSI file formats.

Contents

Working with Unicode in PocketBuilder

PocketBuilder supports the Unicode standard, a universal character set that encodes the characters of over 650 of the world's languages. Using a single character set to encode data in multiple languages allows you to create a single multilingual application that can process data in different languages rather than creating multiple monolingual applications.

Every application development tool encodes text in a particular character set. PocketBuilder uses Unicode UTF-16, a two-byte encoding format. Therefore, any text you enter while developing your application is in Unicode.

Working with ANSI and Unicode files outside PocketBuilder

In WordPad, you can open and save files as ANSI or Unicode. In TextPad, you can open and save files as ANSI or Unicode, and you can determine whether an open file is ANSI or Unicode by viewing its properties.

This section describes how PocketBuilder handles Unicode in several areas of the product. The next two sections, “Importing and exporting DataWindow data” on page 305 and “Reading and writing text or binary files” on page 306, describe how you use these techniques with particular reference to Unicode.

Fonts	Some fonts do not display Unicode characters correctly or do not work appropriately on all platforms, and the number of fonts available on Pocket PC devices is limited. To ensure consistent display on all platforms, PocketBuilder uses Tahoma as its default font.
External function calls	If you call external functions in your application, the functions must be defined and compiled with Unicode support. All strings must be passed as Unicode strings. You can call Windows CE API functions. For more information, see "Using external functions" on page 309.
Converting applications in PocketBuilder	You can convert a PocketBuilder application to a PowerBuilder application and vice versa. You must use PocketBuilder to perform both conversions. If you convert a PocketBuilder application to PowerBuilder 10 or higher, you must select the Create Unicode Libraries check box in the Export Pocket to Desktop conversion tool. If you convert the PocketBuilder application to PowerBuilder 9, you must clear the Create Unicode Libraries check box. Unicode characters that are not supported in ANSI environments, however, cannot be converted correctly for PowerBuilder 9 applications. For more information about the conversion tools, see the *User's Guide*.
Importing and exporting files	PocketBuilder allows you to import both ANSI and Unicode files in the System Tree and Library painter, although you cannot import ANSI files at runtime on a handheld device. Exported source (*.sr**) files are always in Unicode format. To convert from ANSI to Unicode and from Unicode to ANSI in PocketBuilder, use the FromANSI and ToANSI functions.
PocketBuilder resource files	PocketBuilder resource (*.pkr*) files can be in either Unicode or ANSI file format.
Target and Workspace files	Target (*.pkt*) and workspace (*.pkw*) files are saved in ANSI format.
Script view and file editor	The Script view and the file editor accept both Unicode and ANSI file formats. New text files are saved in the Unicode file format.
Writing to initialization files with SetProfileString	On the desktop, the SetProfileString function writes to the text file in the format, either ANSI or Unicode, in which it was opened. On a handheld device, the SetProfileString function writes to the text file in Unicode only. To write Unicode characters to an initialization file, open and save the file as Unicode before calling SetProfileString. The ProfileInt and ProfileString functions also require references to valid Unicode files.

Unicode two-byte flag

Unicode files often have two extra bytes at the start of the file to indicate that they use Unicode byte ordering. On the desktop, PocketBuilder does not require that these two bytes be present in Unicode files. It determines whether the file uses Unicode byte ordering using other methods. However, PocketBuilder applications deployed to a handheld device do require the Unicode byte order mark (BOM) at the beginning of the file.

PocketBuilder also does not always add the two-byte flag to the beginning of files saved with Unicode encoding. A Unicode file that you create in a PocketBuilder application with the FileOpen command does not contain the Unicode byte order mark (BOM) at the beginning of the file when you set the *filemode* argument to StreamMode!. To include the BOM in a file that you open in stream mode, you can do one of the following:

- Create the file by calling FileOpen in line mode, add an innocuous character, such as a single space, then close the file and reopen it in stream mode using the Append! value for the *writemode* argument.
- Call FileWrite or FileWriteEx and pass the 2 byte binary blob "Char(65279)" before the rest of the string that you want to write to the file in stream mode.

SQL Anywhere and Unicode

The SQL Anywhere ODBC driver supports either ASCII (8-bit) strings or Unicode code (wide character) strings. The UNICODE macro controls whether ODBC functions expect ASCII or Unicode strings. If your application must be built with the UNICODE macro defined, but you want to use the ASCII ODBC functions, then the SQL_NOUNICODEMAP macro must also be defined. For more information, see the SQL Anywhere documentation.

Importing and exporting DataWindow data

PowerBuilder 9 formats not supported
In PowerBuilder 9, ImportFile was enhanced to support CSV and XML, and SaveAs was enhanced to support PDF, XML, and XSL-FO. Except for CSV, these formats are not supported in PocketBuilder.

You can use the ImportFile function to import data into a DataWindow control from a tab-delimited text (*.txt*) file or a comma-separated values (*.csv*) file.

Table 19-1: Formats for ImportFile

File type	File format to be imported
.txt	On the desktop, both ANSI and Unicode files can be imported in PocketBuilder, but only Unicode files can be imported on a deployment device. Unicode files cannot be imported into PowerBuilder 9, but can be imported into PowerBuilder 10 and higher.
.csv	Files created using either PowerBuilder or PocketBuilder can be imported into both versions.

You use the SaveAs function to save the contents of a DataWindow control to one of several file formats. The following table summarizes whether the file formats differ—specifically how character strings are handled and whether they are saved in ANSI or Unicode format.

Table 19-2: Formats for SaveAs

File type	Format of saved file
CSV!	Comma-separated values saved as Unicode strings
DIF!, Excel!, Excel5!, WKS!	Character strings saved as ANSI strings
HTMLTable!	HTML syntax saved as Unicode strings
SQLInsert!	SQL syntax and data values saved as Unicode strings
Text!	Text saved as Unicode strings

Reading and writing text or binary files

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

The FileOpen function can open Unicode and ANSI files. If the file does not exist, FileOpen creates a Unicode file. The FileClose function saves the file in the format in which it was opened.

- In *line mode*, the FileRead, FileReadEx, FileWrite, and FileWriteEx functions can read and write to Unicode files.

  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*, the FileRead, FileReadEx, FileWrite, and FileWriteEx functions can read and write to Unicode and ANSI files. FileOpen and the file read and write functions assume that any file is a binary file. FileOpen opens the file as a binary file; FileWrite and FileWriteEx writes to it as a binary file.

  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).

  The format in which FileWrite and FileWriteEx append data to a file depends on the format of the data, not the format of the file. If you append a string entered in PocketBuilder to an ANSI file, it is written as a Unicode character string. If you are reading from or writing to ANSI files in stream mode, use the FromANSI and ToANSI functions to convert ANSI blobs to Unicode character strings, or Unicode characters to ANSI blobs.

  This code opens an ANSI file in stream mode, reads the data into a blob, then converts the blob into a Unicode string:

```
long ll_fnum
integer li_bytes
string ls_unicode
blob  lb_ansi

ll_fnum = FileOpen("employee.dat",StreamMode!,
   Read!, LockWrite!, Replace!)

li_bytes = FileRead(ll_fnum, lb_ansi)
ls_unicode = FromANSI(lb_ansi)
FileClose(ll_fnum)
```

  This code converts a Unicode character string into an ANSI blob, opens an ANSI file in stream mode, and writes the blob to the file:

```
lb_ansi = ToANSI(ls_unicode)
ll_fnum = FileOpen("employee.dat",StreamMode!,
   Write!, LockWrite!, Replace!)

li_bytes = FileWrite(ll_fnum, lb_ansi)
FileClose(ll_fnum)
```

  If the file is in Unicode format and has the two-byte flag, use the Mid function to skip the leading bytes when the file is opened in stream mode:

```
bytes = fileread(ll_fnum, ls_unicode)
ls_string = Mid(ls_unicode, 2)
```

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 PocketBuilder 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. PocketBuilder advances the pointer automatically after each read or write.

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

File functions

Table 19-3 lists the built-in PowerScript functions that manipulate files.

Table 19-3: PowerScript functions that manipulate files

Function	Datatype 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 and FileReadEx	Integer	Read from the specified file
FileSeek	Long	Seeks to a position in the specified file
FileWrite and FileWriteEx	Integer	Write to the specified file

CHAPTER 20

Using External Functions and Other Processing Extensions

About this chapter

This chapter describes how to use external functions and other processing extensions in PocketBuilder.

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 written in any language that supports the standard calling sequence for Windows CE platforms.

If you are calling functions in libraries that you have written yourself, remember that you need to export the functions. Depending on your compiler, you can do this in the function prototype or in a linker definition (DEF) file.

Declaring external functions

Before you can use an external function in a script, you must declare it.

Two types

You can declare two types of external functions:

- **Global external functions**, which are available anywhere in the application
- **Local external functions**, which are defined for a particular type of window, menu, 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.

Datatypes for external function arguments

When you declare an external function, the datatypes of the arguments must correspond with the datatypes as declared in the function's source definition.

For a comparison of datatypes in external functions and in PocketBuilder, see the section on declaring and calling external functions in the *PowerScript Reference*.

❖ **To declare an external function:**

1. If you are declaring a local external function, open the object for which you want to declare it.
2. In the Script view, select Declare in the first drop-down list and either Global External Functions or Local External Functions from the second list.
3. Enter the function declaration in the Script view.

   For the syntax to use, see the *PowerScript Reference* or the examples below.
4. Save the object.

   PocketBuilder compiles the declaration. If there are syntax errors, an error window opens, and you must correct the errors before PocketBuilder can save the declaration.

Sample declarations

Suppose you have created a C dynamic library, *SIMPLE.DLL*, that contains a function called SimpleFunc that accepts two parameters: a character string and a structure. The following statement declares the function in PocketBuilder, passing the arguments by reference:

```
FUNCTION int SimpleFunc(REF string lastname, &
      REF my_str pbstr) LIBRARY "simple.dll"
```

Declaring Windows CE functions

The Windows CE API includes a subset of the functions in the Windows API. The Windows CE API libraries have different names. The following examples show sample declarations for functions in the Windows CE library *coredll.dll*.

On the desktop, these functions are in the Windows library *user32.dll*. To test the function call on the desktop, substitute the desktop equivalent, shown in comment lines after each call, for the Windows CE version.

```
FUNCTION ulong CreateWindowEx( ulong dwExStyle, &
   readonly string ClassName, &
   readonly string WindowName, &
   long dwStyle, &
   long xPos, long yPos, long wwidth, long wheight, &
   ulong hwndParent, ulong hMenu, ulong hInstance, &
   ulong lParams ) &
   library "Coredll.dll" alias for "CreateWindowExW"
// library "user32.dll" alias for "CreateWindowExW"

FUNCTION ulong BringWindowToTop( ulong hwnd ) &
   library "Coredll.dll"
//FUNCTION ulong BringWindowToTop( ulong hwnd ) &
//   library "user32.dll"

FUNCTION ulong SendMessageStr( ulong hwnd, &
   ulong wMsg, ulong wParam, &
   readonly string lParam ) &
   library "Coredll.dll" alias for "SendMessageW"
//  library "user32.dll" alias for "SendMessageW"

FUNCTION ulong SendMessageLong( ulong hwnd, &
   ulong wMsg, ulong wParam, &
   ulong lParam ) &
   library "Coredll.dll" alias for "SendMessageW"
//  library "user32.dll" alias for "SendMessageW"

FUNCTION ulong SendMessagePtr( ulong hwnd, &
   ulong wMsg, ulong wParam, &
   REF ulong lParam[] ) &
   library "Coredll.dll" alias for "SendMessageW"
//   library "user32.dll" alias for "SendMessageW"

FUNCTION ulong SetWindowPos( ulong hwnd, ulong
hwndAfter, &
   ulong xPos, ulong yPos, ulong cX, ulong cY, &
   ulong wFlage ) &
   library "Coredll.dll"
//   library "user32.dll"
```

The following statement declares the function that registers common controls classes from the common control dynamic-link library *commctrl.dll* on Windows CE:

```
FUNCTION ulong InitCommonControlsEx ( ulong &
   pInitCtrls[] )library "commctrl.dll" // WinCE
```

This is the equivalent statement on the desktop:

```
FUNCTION ulong InitCommonControlsEx( REF ulong &
   pInitCtrls[2] )  library "comctl32.dll" // Win32
```

For a partial example that uses some of these declarations, see "Using external functions in a script" on page 316.

You can find a sample application that uses Windows CE API functions in the PocketBuilder project on the Sybase CodeXchange Web site at http://pocketbuilder.codexchange.sybase.com.

For more information about *coredll.dll*, *commctrl.dll*, and other modules, see the section on Windows CE modules in the Microsoft Windows CE 3.0 API documentation at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcesdkr/html/wce30oriWindowsCE30APIReference.asp.

The following statement declares the function that loads and initializes the Microsoft RichInk edit control:

```
FUNCTION ulong InitInkX_CE()  &
   library "inkx.dll"  alias for "InitInkX"
```

For more information about using the RichInk edit control, see "Using Rich Ink Technology in Microsoft Windows CE 3.0" in the MSDN library at http://msdn2.microsoft.com/en-us/library/ms834457.aspx.

RAPI

Remote API (RAPI) is a set of application programming interfaces that let applications running on the desktop invoke functions directly on the Windows CE based device. The set of functions is similar to the Windows CE API, with functions for managing the registry, file system, and databases, and for querying the system configuration. There are additional functions for initializing the RAPI subsystem and enhancing performance.

The following example shows declarations of the CeCloseHandle and CeCreateFile RAPI functions:

```
Function Boolean CeCloseHandle  ( Long hObject) Library
"rapi.dll"
```

```
Function Long CeCreateFile  ( &
    String lpFileName, &
    uLong dwDesiredAccess, &
    uLong dwShareMode, &
    REF SECURITY_ATTRIBUTES lpSecurityAttributes, &
    Long dwCreationDistribution, &
    Long dwFlagsAndAttributes, &
    Long hTemplateFile)    Library "rapi.dll"
```

For more information about RAPI, see the MSDN library at http://msdn2.microsoft.com/en-us/library/aa182819.aspx.

For a sample using RAPI, see the Sybase CodeXchange Web site at http://pockettbuilder.codexchange.sybase.com.

Passing arguments

In PocketBuilder, 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 PocketBuilder. When you pass the argument by value, the external function receives a copy of the argument and can change the contents of the copy of the argument. The changes affect only the local copy; the contents of the original argument are unchanged.

The syntax for an argument that is passed by reference is:

REF *datatype arg*

The syntax for an argument that is passed by value is:

datatype arg

Passing numeric datatypes

The following statement declares the external function TEMP in PocketBuilder. This function returns an integer and expects an integer argument to be passed by reference:

```
FUNCTION int TEMP(ref int degree)    LIBRARY
"LibName.DLL"
```

The same statement in C would be:

```
int _stdcall TEMP(int * degree)
```

Since the argument is passed by reference, the function can change the contents of the argument, and changes made to the argument within the function will directly affect the value of the original variable in PocketBuilder. For example, the C statement `*degree = 75` would change the argument named degree to 75 and return 75 to PocketBuilder.

The following statement declares the external function TEMP2 in PocketBuilder. This function returns an integer and expects an integer argument to be passed by value:

```
FUNCTION int TEMP2(int degree) LIBRARY "LibName.DLL"
```

The same statement in C would be:

```
int _stdcall TEMP2(int degree)
```

Since the argument is passed by value, the function can change the contents of the argument. All changes are made to the local copy of the argument; the variable in PocketBuilder is not affected.

Passing strings

Passing by value The following statement declares the external C function NAME in PocketBuilder. 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 * _stdcall NAME(char  * CODE)
```

Since the string is passed by value, the C function can change the contents of its local copy of *CODE*, but the original variable in PocketBuilder is not affected.

Passing by reference PocketBuilder has access only to its own memory. Therefore, an external function cannot return a pointer to a string. (It cannot return a memory address.)

When you pass a string to an external function, either by value or by reference, PocketBuilder passes a pointer to the string. If you pass by value, any changes the function makes to the string are not accessible to PocketBuilder. If you pass by reference, they are.

The following statement declares the external C function NAME2 in PocketBuilder. This function returns a string and expects a string argument to be passed by reference:

```
FUNCTION string NAME2(ref string CODE) &
   LIBRARY "LibName.DLL"
```

In C, the statement would be the same as when the argument is passed by value, shown above:

```
char * _stdcall NAME2(char * CODE)
```

The string argument is passed by reference, and the C function can change the contents of the argument and the original variable in PocketBuilder. For example, `Strcpy(CODE,STUMP)` would change the contents of *CODE* to *STUMP* and change the variable in the calling PocketBuilder script to the contents of variable *STUMP*.

If the function NAME2 in the preceding example takes a user ID and replaces it with the user's name, the PowerScript string variable *CODE* must be long enough to hold the returned value. To ensure that this is true, declare the String and then use the Space function to fill the String with blanks equal to the maximum number of characters you expect the function to return.

If the maximum number of characters allowed for a user's name is 40 and the ID is always five characters, you would fill the string *CODE* with 35 blanks before calling the external function:

```
String CODE
CODE = ID + Space(35)
. . .
NAME2(CODE)
```

For information about the Space function, see the *PowerScript Reference*.

Passing chars to C functions Char variables passed to external C functions are converted to the C char type before passing. Arrays of char variables are converted to the equivalent C array of char variables.

An array of string variables embedded in a structure produces an embedded array in the C structure. This is different from an embedded string, which results in an embedded pointer to a string in the C structure.

Recommendation
Whenever possible, pass string variables back to PocketBuilder as a return value from the function.

Using external functions in a script

This section shows an example that uses some of the external function declarations for Windows CE functions in "Sample declarations" on page 310. The example displays a Date Time Picker control from the Microsoft common controls library *commctrl.dll*. For more information about the constant values and other parameters to the external functions, see the Microsoft documentation.

```
// First set instance variables
// ulong g_hwndCal
// CONSTANT ulong ICC_DATE_CLASSES = 256 // 0x100
// CONSTANT ulong WS_BORDER = 8388608    // 0x0080 0000
// CONSTANT ulong WS_CHILD = 1073741824  // 0x4000 0000
// CONSTANT ulong WS_VISIBLE = 268435456 // 0x1000 0000
// CONSTANT ulong SWP_NOZORDER = 4
// CONSTANT long HWND_TOP      = 0
// CONSTANT ulong MCM_GETMINREQRECT = 4105
// CONSTANT ulong MCM_SETCOLOR  = 4106  // 0x100A
// CONSTANT ulong MCSC_MONTHBK = 4      // background
                                        // of a month
// CONSTANT ulong COLOR_MONTH = 12648447  // 0xC0ffff
                                      // (muted yellow)


// Specify the class of the date picker control
string ClassName = "SysDateTimePick32"

long lret
ulong sizeRect[]
ulong xPBUnits, yPBUnits

ulong HeightTitleBar = 0      //  For PDA, set to 0

// Initialize the Common Controls DLL
ulong aInitCtrls[2]
aInitCtrls[1] = 8      // structure size
aInitCtrls[2] = ICC_DATE_CLASSES

lret = InitCommonControlsEx( aInitCtrls ) // external
if lret = 0 then
   return 0
end if
```

```
// make the calendar control
g_hwndCal = CreateWindowEx( 0, ClassName, "", &
      WS_BORDER + WS_CHILD + WS_VISIBLE, &
      0,0,0,0, &
      Handle(this), 0, 0, 0 )

// If really created, initialize the control
if g_hwndCal <> 0 then
   // Set the calendar color
   SendMessageLong( g_hwndCal, MCM_SETCOLOR, &
      MCSC_MONTHBK, COLOR_MONTH )

   // Set the size to what the control requests
   sizeRect[1] = 0
   sizeRect[2] = 0
   sizeRect[3] = 0
   sizeRect[4] = 0
   // Get the minimum size required to display a
   // full month in the control
   SendMessagePtr( g_hwndCal, MCM_GETMINREQRECT, 0, &
      sizeRect )

   // set the calendar control size
   SetWindowPos( g_hwndCal, HWND_TOP, 0, 0, &
      sizeRect[3], sizeRect[4], SWP_NOZORDER )

   // Set the PARENT window (this) to that size
   this.width  = PixelsToUnits( sizeRect[3],
      XPixelsToUnits! )
   this.height = PixelsToUnits
      ( sizeRect[4] + HeightTitleBar, YPixelsToUnits! )
end if
```

Sending Windows messages

To send Windows messages to a window that you created in PocketBuilder or to an external window (such as a window you created using an external function), use the Post or Send function. To trigger a PocketBuilder event, use the EVENT syntax or the TriggerEvent or PostEvent function.

Using Post and Send

You usually use the Post and Send functions to trigger Windows events that are not PocketBuilder-defined events. You can include these functions in a script for the window in which the event will be triggered or in any script in the application.

Post is asynchronous: the message is posted to the message queue for the window or control. Send is synchronous: the window or control receives the message immediately.

All events posted by PocketBuilder are processed by a queue separate from the Windows system queue. PocketBuilder posted messages are processed before Windows posted messages.

Obtaining the window's handle
To obtain the handle of the window, use the Handle function. To combine two integers to form the long value of the message, use the Long function. Handle and Long are utility functions, described in "Using utility functions to manage information" on page 319.

Triggering PocketBuilder events

To trigger a PocketBuilder event, you can use the techniques that are listed in Table 20-1.

Table 20-1: Triggering PocketBuilder events

Technique	Description
TriggerEvent function	A synchronous function that triggers the event immediately in the window or control
PostEvent function	An asynchronous function: the event is posted to the event queue for the window or control
Event call syntax	A method of calling events directly for a control using dot notation

All three methods bypass the messaging queue and are easier to code than the Send and Post functions.

Example All three statements shown below click the CommandButton cb_OK and are in scripts for the window that contains cb_OK.

The Send function uses the Handle utility function to obtain the handle of the window that contains cb_OK, then uses the Long function to combine the handle of cb_OK with 0 (BN_CLICK) to form a long that identifies the object and the event:

```
Send(Handle(Parent),273,0,Long(Handle(cb_OK),0))
cb_OK.TriggerEvent(Clicked!)
cb_OK.EVENT Clicked()
```

The TriggerEvent function identifies the object in which the event will be triggered and then uses the enumerated datatype Clicked! to specify the clicked event.

The dot notation uses the EVENT keyword to trigger the Clicked event. TRIGGER is the default when you call an event. If you were posting the clicked event, you would use the POST keyword:

```
Cb_OK.EVENT POST Clicked()
```

Using utility functions to manage information

The utility functions provide a way to obtain and pass Windows information to external functions and can be used as arguments in the PowerScript Send function. There are four utility functions.

Utility functions

Table 20-2: Utility functions

Function	Return value	Purpose
Handle	UnsignedInt	Returns the handle to a specified object.
IntHigh	UnsignedInt	Returns the high word of the specified long value. IntHigh is used to decode Windows values returned by external functions or the LongParm attribute of the Message object.
IntLow	UnsignedInt	Returns the low word of the specified long value. IntLow is used to decode Windows values returned by external functions or the LongParm attribute of the Message object.
Long	Long	Combines the low word and high word into a long. The Long function is used to pass values to external functions.

Example

This script uses the external function CreateWindowEx to create a child window that displays a calendar. The Handle function is used to pass the handle to the parent window of the window being created:

```
// Instance variable: ulong g_hwndCal
g_hwndCal = CreateWindowEx( 0, ClassName, "", &
      WS_BORDER + WS_CHILD + WS_VISIBLE, &
      0,0,0,0, &
      Handle(this), 0, 0, 0 )
```

The Message object

The Message object is a predefined PocketBuilder 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 PocketBuilder-defined events.

When a Microsoft Windows event occurs that is not a PocketBuilder-defined event, PocketBuilder populates the Message object with information about the event.

Other uses of the Message object

The Message object is also used:

- To communicate parameters between windows when you open and close them

  For more information, see the descriptions of OpenWithParm, OpenSheetWithParm, and CloseWithReturn in the online Help.

- To pass information to an event if optional parameters were used in TriggerEvent or PostEvent

  For more information, see the online Help.

Customizing the Message object

You can customize the global Message object used in your application by defining a standard class user object inherited from the built-in Message object. In the user object, you can add additional properties (instance variables) and functions. You then populate the user-defined properties and call the functions as needed in your application.

For more information about defining standard class user objects, see the *User's Guide*.

Message object properties

Table 20-3 lists the properties of the Message object and the datatype and uses of each. The first four properties of the Message object correspond to the first four properties of the Microsoft Windows message structure.

Table 20-3: Message object properties

Property	Datatype	Use
Handle	Integer	The handle of the window or control.
Number	Integer	The number that identifies the event (this number comes from Windows).

Property	Datatype	Use
WordParm	UnsignedInt	The word parameter for the event (this parameter comes from Windows). The parameter's value and meaning are determined by the event.
LongParm	Long	The long parameter for the event (this number comes from Windows). The parameter's value and meaning are determined by the event.
DoubleParm	Double	A numeric or numeric variable.
StringParm	String	A string or string variable.
PowerObjectParm	PowerObject	Any PocketBuilder object type, including structures.
Processed	Boolean	A boolean value set in the script for the user-defined event: • true – the script processed the event. Do not call the default window Proc (DefWindowProc) after the event has been processed. • false (default) – call DefWindowProc after the event has been processed.
ReturnValue	Long	The value you want returned to Windows when Message.Processed is true. When Message.Processed is false, this attribute is ignored.

Use the values in the Message object in the event script that caused the Message object to be populated. For example, suppose the FileExists event contains the following script. OpenWithParm displays a response window that asks the user if it is OK to overwrite the file. The return value from FileExists determines whether the file is saved:

```
OpenWithParm( w_question, &
   "The specified file already exists. " + &
   "Do you want to overwrite it?" )
IF Message.StringParm = "Yes" THEN
   RETURN 0  // File is saved
ELSE
   RETURN -1 // Saving is canceled
END IF
```

When processing messages, Windows CE supports both system-defined messages and application-defined messages. Windows CE does not support hooking messages. For information on Microsoft message types, see the MSDN library at http://msdn2.microsoft.com/en-us/library/aa452701.aspx.

CHAPTER 21

Managing Initialization Files and the Windows CE Registry

About this chapter

This chapter describes how to manage preferences and default settings for PocketBuilder applications.

Contents

About preferences and default settings

Many applications store user preferences and default settings across sessions. For example, applications can keep track of settings that control the appearance and behavior of the application, or store default parameters for connecting to the database. PocketBuilder applications can manage this kind of information in initialization files or in the Windows CE registry.

Database connection parameters

You might need to set the values of the Transaction object from an external file. For example, you might want to retrieve values from your PocketBuilder initialization file when you are developing the application, or from an application-specific initialization file when you distribute the application.

For information about database connection parameters in an initialization file, see "Reading values from an external file" on page 238.

For an example of how to save and restore database connection parameters in the Windows CE registry, see "Managing information in the Windows CE registry" on page 325.

Custom Today item parameters

When you deploy a PocketBuilder application, you can add a custom Today item to the Pocket PC Today screen. Information about the custom Today item is entered in the Windows CE registry. You can also remove Windows CE registry information about custom Today items from the PocketBuilder IDE.

For information about custom Today items, see the chapter on "Working with PowerScript Targets" in the *User's Guide*.

Other settings you might want to save

In addition to the database connection custom Today item parameters, you might want to store a variety of other application-specific settings, such as user preferences for colors, fonts, and other display settings.

Managing information in initialization files

Functions for accessing initialization files

PocketBuilder provides several functions you can use to manage application settings in initialization files.

Table 21-1: PocketBuilder initialization file functions

Function	Description
ProfileInt	Obtains the integer value of a setting in a profile file
ProfileString	Obtains the string value of a setting in a profile file
SetProfileString	Writes a value in a profile file

For complete information about these functions, see the online Help.

The format of APP.INI

The examples below manage application information in a profile file called *APP.INI*. This file keeps track of user preferences that control the appearance of the application. It has a Preferences section that stores four color settings:

```
[Preferences]
WindowColor=Silver
BorderColor=Red
BackColor=Black
TextColor=White
```

Reading values

The following script retrieves color settings from the *APP.INI* file. *Wincolor*, *brdcolor*, *bckcolor*, and *txtcolor* are string variables:

```
wincolor = ProfileString("app.ini", "Preferences", "WindowColor", "")
brdcolor = ProfileString("app.ini", "Preferences", "BorderColor", "")
bckcolor = ProfileString("app.ini", "Preferences", "BackColor", "")
txtcolor = ProfileString("app.ini", "Preferences", "TextColor", "")
```

Setting values

The following script stores color settings in the *APP.INI* file:

```
SetProfileString("app.ini", "Preferences", "WindowColor", wincolor)
SetProfileString("app.ini", "Preferences", "BorderColor", brdcolor)
SetProfileString("app.ini", "Preferences", "BackColor", bckcolor)
SetProfileString("app.ini", "Preferences", "TextColor", txtcolor)
```

Managing information in the Windows CE registry

Functions for accessing the Registry

PocketBuilder provides several functions you can use to manage application settings in the Windows CE registry.

Table 21-2: PocketBuilder registry setting functions

Function	Description
RegistryDelete	Deletes a key or a value in a key in the Windows registry.
RegistryGet	Gets a value from the Windows registry.
RegistryKeys	Obtains a list of the keys that are child items (subkeys) one level below a key in the Windows registry.
RegistrySet	Sets the value for a key and/or a value name in the Windows registry. If the key or value name does not exist, RegistrySet creates a new key or value name.
RegistryValues	Obtains a list of named values associated with a key.

For complete information about these functions, see the online Help.

To explore and edit the Windows CE registry on a Pocket PC, you can download the PHM Registry Editor at http://www.phm.lu/Products/PocketPC/.

Reading values from the registry

The examples that follow use the registry to keep track of database connection parameters. The connection parameters are maintained in the registry in the *MyCo\MyApp\database* branch under *HKEY_CURRENT_USER\Software*.

The following script retrieves values for the default Transaction object from the registry.

```
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
   "dbms", sqlca.DBMS)
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "database", sqlca.database)
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "userid", sqlca.userid)
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "dbpass", sqlca.dbpass)
```

```
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "logid", sqlca.logid)
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "logpass", sqlca.logpass)
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "servername", sqlca.servername)
RegistryGet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "dbparm", sqlca.dbparm)
```

Setting values in the registry

The following script stores the values for the Transaction object (set elsewhere in the application) in the registry:

```
RegistrySet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "dbms", sqlca.DBMS)
RegistrySet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "database", sqlca.database)
RegistrySet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "userid", sqlca.userid)
RegistrySet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "dbpass", sqlca.dbpass)
RegistrySet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "logid", sqlca.logid)
RegistrySet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "logpass", sqlca.logpass)
RegistrySet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "servername", sqlca.servername)
RegistrySet("HKEY_CURRENT_USER\Software\MyCo\MyApp\database", &
    "dbparm", sqlca.dbparm)
```

CHAPTER 22

Using the Command Line

About this chapter

This chapter describes how to use command-line arguments to build applications and start PocketBuilder.

Contents

Starting PocketBuilder from a command line

You can start the PocketBuilder executable from a command line to build a workspace without opening the development environment, or to open the development environment with a specific painter or object open. You can also use the Windows Run dialog box to accomplish these tasks.

The general syntax is:

directory\pk20.exe {/workspace *workspacepath*} {*deployoptions* | *runoptions*} {/output *outputpath*}

where *directory* is the fully qualified name of the directory containing PocketBuilder.

Table 22-1: General command-line options for PocketBuilder

Option	Description
/W *workspacepath*	Opens the workspace *workspacepath*. This option is required for deployment. To open PocketBuilder, the default is the most recently used workspace if you have selected the Reopen Workspace on Startup check box in the System Options dialog box. If you have not selected this check box, you must specify the /W option before specifying any other options.
deployoptions	Options for deploying the workspace. See Table 22-2.
runoptions	Options for opening the development environment. See Table 22-3.
/OU *outputpath*	Logs the contents of the Output window to *outputpath*.

Short option names
The syntax statements in this chapter show the long form of option names. You need only use the initial letter or letters of the option name as long as the option is uniquely identified, as shown in Table 22-1.

Building workspaces from a command line

The deploy options are deploy, fullbuild, and incrementalbuild. These options must be used with the workspace option.

Table 22-2: Options for deploying a workspace

Option	Description
/deploy	Deploys the workspace and exits
/fullbuild	Fully builds the workspace and exits
/incrementalbuild	Incrementally builds the workspace and exits

You need to create projects and specify build and deploy options for the workspace in PocketBuilder *before* you start a build from the command line. Deploy builds the projects in the target in the order listed on the Deploy page of the target properties dialog box.

When you deploy or build a workspace from a command line, PocketBuilder starts, completes the build, and exits as soon as the operation is completed. To retain a log file for the session, you can send the contents of the Output window to a file.

This example assumes that the location of the PocketBuilder executable file is in your system path or in the directory from which you enter the command. It opens the workspace called CDShop, builds and deploys the targets in the workspace according to your specifications in the workspace and target properties, records the content of the Output window in the file *D:\tmp\cdshop.out*, and exits PocketBuilder:

```
pk20 /w D:\CDShop\CDShop.pkw /d /out D:\tmp\cdshop.out
```

Starting PocketBuilder with command-line arguments

When you start PocketBuilder from a command line, you can optionally open a workspace, target, and/or painter. These are the painters and tools you can open:

Application painter
Database painter
DataWindow painter
Debugger
File Editor
Function painter
Library painter

Menu painter
Query painter
Structure painter
User Object painter
Window painter

You can also add options to the command line after /painter *paintername* to open a specific object or create a new one:

{/target targetpath} {/painter paintername} {/library libraryname} {/object objectname} {/inherit objectname} {/new} {/run} {/runonly} {/argument arguments}

Table 22-3: Options for opening the development environment

Option	Description
/T *targetpath*	Opens the target in *targetpath*.
/P *paintername*	Opens the painter *paintername*. The default is the window that displays when you begin a new PocketBuilder session. The painter name must uniquely identify the painter. You do not have to enter the entire name. For example, you can enter `q` to open the Query painter and `datab` to open the Database painter. If you enter the full name, omit any spaces in the name (enter `UserObject`, for example). The painter name is not case sensitive. To open the file editor, you could set *paintername* to `FI` or `fileeditor`. Except for the /W, /T, and /L switches, switches must follow /P *paintername* on the command line, as shown in the examples after the table.
/L *libraryname*	The library that contains the object you want to open.
/O *objectname* (or /OBJ *objectname*)	The object, such as a DataWindow object or window, you want to open.
/I *objectname*	The object you want to inherit from.
/N	Creates a new DataWindow object.
/R	Runs the DataWindow object specified with /O and allows designing.
/RO	Runs the DataWindow object specified with /O but does not allow designing.
/A *arguments*	Arguments for the specified DataWindow object.

Examples

The following examples assume that the location of the PocketBuilder executable file is in your system path.

This example starts a PocketBuilder session by opening a new window (a window object is not specified) in the Window painter for the Client PKL in the Math workspace. The output of the session is sent to a file called *math.log*. The workspace file, the PKL, and the log file are all in the current directory:

```
pk20 /w Math.pkw /l Client.pkl /p window /out math.log
```

Enter this command to start PocketBuilder and open the DataWindow object called d_emp_report in the workspace *Emp.pkw*:

```
pk20 /w D:\pkws\Emp.pkw /P dataw /O d_emp_report
```

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W