Developer’s Guide

Borland®

Kylix™

Delphi™ for Linux®

Borland Software Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249

Printed in the U.S.A.

HDB7010WW21001   1E0R0201
0102030405-9 8 7 6 5 4 3 2 1
PDF

Contents

Tables

Figures

Chapter 1

Introduction

The *Developer's Guide* describes intermediate and advanced development topics, such as building database applications, writing custom components, and creating Internet Web server applications using Kylix, Delphi for the Linux operating system. The *Developer's Guide* assumes you are familiar with using Linux and understand fundamental programming techniques. For an introduction to Kylix programming and the integrated development environment (IDE), see the online Help and the *Quick Start* manual.

What's in this manual?

This manual contains the following parts:

- **Part I, "Programming with Kylix,"** describes how to build general-purpose Kylix applications. This part provides details on programming techniques you can use in any Kylix application. For example, it describes how to use common Borland Component Library for Cross Platform (CLX) objects that simplify user interface development such as handling strings and manipulating text. It also includes chapters on working with graphics, controls, error and exception handling, and writing international applications. The chapter on deployment describes the tasks involved in deploying your application to your application users.
- **Part II, "Developing database applications,"** describes how to build database applications using database tools and components. Kylix lets you access SQL server databases using DBExpress. Your version of Kylix comes with a set of DBExpress drivers for connecting to specific databases. Additional DBExpress drivers for connecting to other databases are available for purchase separately.
- **Part III, "Writing distributed applications,"** describes how to create applications that are distributed over a local area network. These include Web server applications, such as Apache and CGI applications. For lower-level support of distributed applications, this section also describes how to work with socket components, that handle the details of communication using TCP/IP and related

protocols. The components that support sockets and Web server applications are not available in the standard edition of Kylix.

- **Part IV, "Creating custom components,"** describes how to design and implement your own components, and how to make them available on the Component palette in Kylix's development environment. A component can represent almost any program element that you want to manipulate at design time. Implementing custom components involves deriving a new class from an existing class type in the CLX class library.

Manual conventions

This manual uses the typefaces and symbols described in Table 1.1 to indicate special text.

Table 1.1 Typefaces and symbols

Typeface or symbol	Meaning
`Monospace type`	Monospaced text represents text as it appears on screen or in Object Pascal code. It also represents anything you must type.
[ ]	Square brackets in text or syntax listings enclose optional items. Text of this sort should not be typed verbatim.
Boldface	Boldfaced words in text or code listings represent Object Pascal keywords or compiler options.
Italics	Italicized words in text represent Object Pascal identifiers, such as variable or type names. Italics are also used to emphasize certain words, such as new terms.
Keycaps	This typeface indicates a key on your keyboard. For example, "Press *Esc* to exit a menu."

Developer support services

Borland also offers a variety of support options to meet the needs of its diverse developer community. To find out about support offerings for Kylix, refer to http://www.borland.com/devsupport. Additional Kylix Technical Information documents and answers to Frequently Asked Questions (FAQs) are also available at this Web site.

From the Web site, you can access many newsgroups where Kylix developers exchange information, tips, and techniques.

Refer also to the Borland Community site at http://community.borland.com. It provides access to lots of information, articles, code examples, and upcoming news about Kylix.

Ordering printed documentation

For information about ordering additional documentation, refer to the Web site at shop.borland.com.

Part

I

Programming with Kylix

The chapters in "Programming with Kylix" introduce concepts and skills necessary for creating Kylix applications. They also introduce the concepts discussed in later sections of the *Developer's Guide*.

Chapter

2

Developing applications with Kylix

Borland Kylix is an object-oriented, visual programming environment for rapid development of 32-bit applications. Using Kylix, you can create highly efficient cross-platform applications with a minimum of manual coding.

Kylix provides a comprehensive class library called the Borland Component Library for Cross Platform (CLX) and a suite of Rapid Application Development (RAD) design tools, including application and form templates, and programming wizards. Kylix supports truly object-oriented programming: the class library includes many useful objects that you can use while developing applications.

This chapter briefly describes the Kylix development environment. The rest of this manual provides technical details on developing general-purpose, database, Internet and Intranet applications, and includes information on writing your own components.

Integrated development environment

When you start Kylix, you are immediately placed within the integrated development environment, also called the IDE. This environment provides all the tools you need to design, develop, test, debug, and deploy applications.

Kylix's development environment includes a visual form designer, Object Inspector, Component palette, Project Manager, source code editor, and debugger. You can move freely from the visual representation of an object (in the form designer), to the Object Inspector to edit the initial runtime state of the object, to the source code editor to edit the execution logic of the object. Changing code-related properties, such as the name of an event handler, in the Object Inspector automatically changes the corresponding source code. In addition, changes to the source code, such as renaming an event handler method in a form class declaration, is immediately reflected in the Object Inspector.

The IDE supports application development throughout the stages of the product life cycle—from design to deployment. Using the tools in the IDE allows for rapid prototyping and shortens development time.

A more complete overview of the development environment is presented in the *Quick Start* manual included with the product. In addition, the online Help system provides help on all menus, dialogs, and windows.

Designing applications

Kylix includes all the tools necessary for you to start designing applications:

- A blank window, known as a *form*, on which to design the UI for your application.
- An extensive class library (CLX) that contains many reusable objects.
- An Object Inspector for examining and changing object traits.
- A Code editor that provides direct access to the underlying program logic.
- A Project Manager for managing the files that make up one or more projects.
- Many other tools such as an integrated debugger to support application development in the IDE.
- Command-line tools including compilers, linkers, and other utilities.

You can use Kylix to design any kind of 32-bit application—from general-purpose utilities to sophisticated data access programs or distributed applications. Kylix's database tools and data-aware components let you quickly develop powerful desktop database and client/server applications. Using Kylix's data-aware controls, you can view live data while you design your application and immediately see the results of database queries and changes to the application interface.

Many of the objects provided in the class library are accessible in the IDE from the component palette. The component palette shows all of the controls, both visual and nonvisual, that you can place on a form. Each tab contains components grouped by functionality. By convention, the names of objects in the class library begin with a T, such as *TStatusBar*.

One of the revolutionary things about Kylix is that you can create your own components using Object Pascal. Most of the components provided are written in Object Pascal. You can add components that you write to the component palette and customize the palette for your use by including new tabs if needed.

Chapter 5, "Building applications and shared objects" introduces Kylix's support for different types of applications.

Developing applications

As you visually design the user interface for your application, Kylix generates the underlying Object Pascal code to support the application. As you select and modify the properties of components and forms, the results of those changes appear automatically in the source code, and vice versa. You can modify the source files directly with any text editor, including the built-in Code editor. The changes you make are immediately reflected in the visual environment as well. This feature is called “two-way tools.”

Creating projects

All of Kylix’s application development revolves around projects. When you create an application in Kylix you are creating a project. A project is a collection of files that make up an application. Some of these files are created at design time. Others are generated automatically when you compile the project source code.

You can view the contents of a project in a project management tool called the Project Manager. The Project Manager lists, in a hierarchical view, the unit names, the forms contained in the unit (if there is one), and shows the paths to the files in the project. Although you can edit many of these files directly, it is often easier and more reliable to use the visual tools in Kylix.

At the top of the project hierarchy, is a group file. You can combine multiple projects into a project group. This allows you to open more than one project at a time in the Project Manager. Project groups let you organize and work on related projects, such as applications that function together or parts of a multi-tiered application. If you are only working on one project, you do not need a project group file to create an application.

Project files, which describe individual projects, files, and associated options, have a .dpr extension. Project files contain directions for building an application or shared object. When you add and remove files using the Project Manager, the project file is updated. You specify project options using a Project Options dialog which has tabs for various aspects of your project such as forms, application, compiler. These project options are stored in the project file with the project.

Units and forms are the basic building blocks of a Kylix application. A project can share any existing form and unit file including those that reside outside the project directory tree. This includes custom procedures and functions that have been written as standalone routines.

If you add a shared file to a project, realize that the file is not copied into the current project directory; it remains in its current location. Adding the shared file to the current project registers the file name and path in the **uses** clause of the project file. Kylix automatically handles this as you add units to a project.

When you compile a project, it does not matter where the files that make up the project reside. The compiler treats shared files the same as those created by the project itself.

Editing code

The Kylix Code editor is a full-featured ASCII editor. If using the visual programming environment, a form is automatically displayed as part of a new project. You can start designing your application interface by placing objects on the form and modifying how they work in the Object Inspector. But other programming tasks, such as writing event handlers for objects, must be done by typing the code.

The contents of the form, all of its properties, its components, and their properties can be viewed and edited as text in the Code editor. You can adjust the generated code in the Code editor and add more components within the editor by typing code. As you type code into the editor, the compiler is constantly scanning for changed and updating the form with the new layout. You can then go back to the form, view and test the changes you made in the editor and continue adjusting the form from there.

The Kylix code generation and property streaming systems are completely open to inspection. The source code for everything that is included in your final executable file—all of the CLX objects, RTL sources, all of the Kylix project files can be viewed and edited in the Code editor.

Compiling applications

When you have finished designing your application interface on the form, writing additional code so it does what you want, you can compile the project from the IDE or from the command line.

All projects have as a target a single distributable executable file. You can view or test your application at various stages of development by compiling, building, or running it:

- When you compile, only units that have changed since the last compile are recompiled.
- When you build, all units in the project are compiled, regardless of whether or not they have changed since the last compile. This technique is useful when you are unsure of exactly which files have or have not been changed, or when you simply want to ensure that all files are current and synchronized. It's also important to use Build when you've changed global compiler directives, to ensure that all code compiles in the proper state. You can also test the validity of your source code without attempting to compile the project.
- When you run, you compile and then execute your application. If you modified the source code since the last compilation, the compiler recompiles those changed modules and relinks your application.

If you have grouped several projects together, you can compile or build all projects in a single project group at once. Choose Project | Compile All Projects or Project | Build All Projects with the project group selected in the Project Manager.

Debugging applications

Kylix provides an integrated debugger that helps you find and fix errors in your applications. The integrated debugger lets you control program execution, monitor variable values and items in data structures, and modify data values while debugging.

The integrated debugger can track down both runtime errors and logic errors. By running to specific program locations and viewing the values of variables, the functions on the call stack, and the program output, you can monitor how your program behaves and find the areas where it is not behaving as designed. The debugger is described in online Help.

You can also use exception handling to recognize, locate, and deal with errors. Exceptions in Kylix are classes, like other classes in Kylix. except, by convention, they begin with an E rather than the initial T for other classes. Refer to “Handling exceptions” on page 4-4 for details on exception handling.

Deploying applications

Kylix includes tools to help with application deployment. You can create an installation package for your application that includes all of the files needed for running a distributed application. Refer to Chapter 13, “Deploying applications” for specific information on deployment.

Note Not all versions of Kylix have deployment capabilities.

Chapter

3 Using CLX

This chapter presents an overview of the Borland Component Library for Cross Platform (CLX) and introduces some of the objects that you can use while developing applications.

Understanding CLX

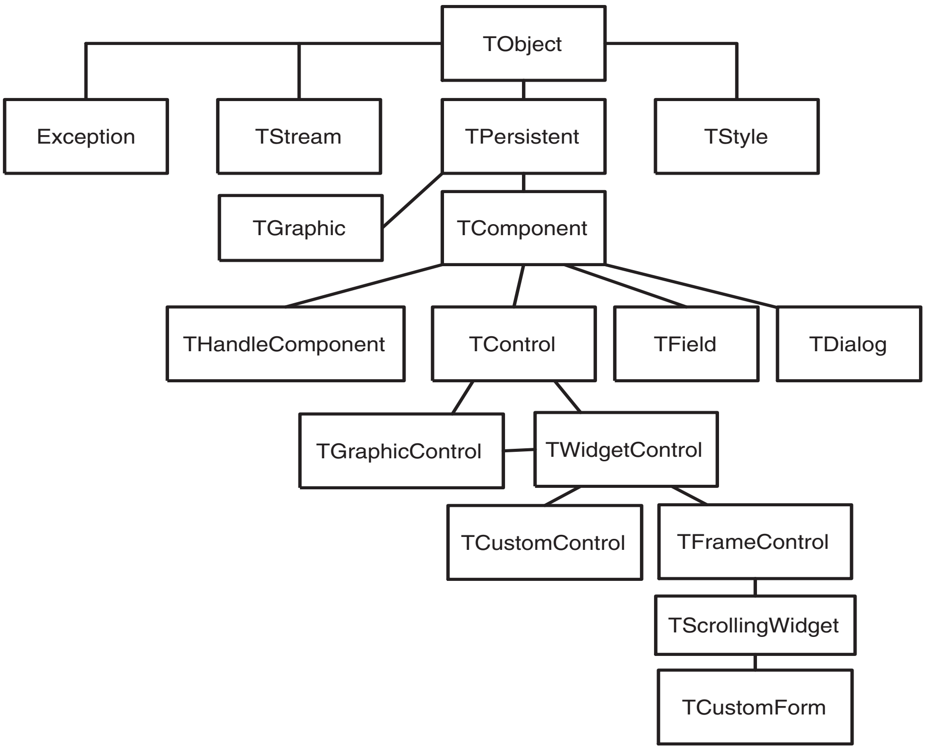
Kylix is a component-based development environment for two-way visual development of graphical user interface (GUI), Internet, database, and server applications. The components are implemented in Linux in a cross-platform version of the component library called the Borland Component Library for Cross Platform (CLX, pronounced “clicks”). CLX is designed to radically speed up native Linux application development time and simplify cross-platform development for the Linux and Windows operating systems.

Object-oriented programming is an extension of structured programming that emphasizes code reuse and encapsulation of data with functionality. Once an object (or, more formally, a class) is created, you and other programmers can use it in different applications, thus reducing development time and increasing productivity.

CLX is a class library made up of many objects, some of which are also components or controls, that you use when developing applications.

Figure 3.1 shows the organization of CLX including some of the important objects.

Figure 3.1 CLX organization



CLX objects are active entities that contain all necessary data and the “methods” (code) that modify the data. The data is stored in the fields and properties of the objects, and the code is made up of methods that act upon the field and property values. Each object is declared as a “class.” All CLX objects descend from the ancestor object *TObject* including objects that you develop yourself in Object Pascal.

A subset of objects are components. Components in CLX descend from the abstract class *TComponent*. Components are objects that you can place on a form or data module and manipulate at design time. Components appear on the component palette. You can specify their properties without writing code. All CLX components descend from the *TComponent* object.

Components are objects in the true object-oriented programming (OOP) sense because they

- Encapsulate some set of data and data-access functions
- Inherit data and behavior from the objects they are derived from
- Operate interchangeably with other objects derived from a common ancestor, through a concept called *polymorphism.*

Unlike most components, objects do not appear on the component palette. Instead, a default instance variable is declared in the unit of the object, or you have to declare one yourself.

Visual components—that is, components like *TForm* and *TSpeedButton,* which appear on the screen at runtime—are called *controls*, and they descend from *TControl*. Controls are a special kind of component that is visible to users at runtime. Controls are a subset of components. You can see controls in the user interface when your application is running. All controls have properties in common that specify their visual attributes, such as *Height* and *Width.* The properties, methods, and events that controls have in common are all inherited from *TControl*.

CLX also includes many nonvisual objects that you can add to your programs by dropping them onto forms. For example, if you were writing an application that connects to a database, you might place a *TDataSource* component on a form. Although *TDataSource* is nonvisual, it is represented on the form by an icon (that doesn't appear at runtime). You can manipulate the properties and events of *TDataSource* in the Object Inspector just as you would those of a visual control.

Detailed reference material on all of the objects in CLX is accessible using online Help while you are programming.

Properties, methods, and events

CLX is a hierarchy of objects, written in Object Pascal and tied to the Kylix IDE, where you can develop applications quickly. CLX is based on properties, methods, and events. It defines the data members (properties), the functions that operate on the data (methods), and a way to interact with users of the class (events).

Properties

Properties are characteristics of an object that influence either the visible behavior or the operations of the object. For example, the *Visible* property determines whether an object can be seen or not in an application interface. Well-designed properties make your components easier for others to use and easier for you to maintain.

Here are some of the useful features of properties:

- Unlike methods, which are only available at runtime, you can see and change properties at design time and get immediate feedback as the components change in the IDE.
- Properties can be accessed in the Object Inspector where you can modify the values of your object visually. Setting properties at design time is easier than writing code and makes your code easier to maintain.
- Because the data is encapsulated, it is protected and private to the actual object.
- The actual calls to get and set the values are methods, so special processing can be done that is invisible to the user of the object. For example, data could reside in a table, but could appear as a normal data member to the programmer.

- You can implement logic that triggers events or modifies other data during the access of the property. For example, changing the value of one property may require the modification of another. You can make the change in the methods created for the property.
- Properties can be virtual.
- A property is not restricted to a single object. Changing a one property on one object could effect several objects. For example, setting the *Checked* property on a radio button effects all of the radio buttons in the group.

Methods

A *method* is a procedure that is always associated with a class. Methods define the behavior of an object. Class methods can access all the *public*, *protected,* and *private* properties and data members of the class and are commonly referred to as member functions.

Events

An *event i*s an action or occurrence detected by a program. Most modern applications are said to be event-driven, because they are designed to respond to events. In a program, the programmer has no way of predicting the exact sequence of actions a user will perform next. They may choose a menu item, click a button, or mark some text. You can write code to handle the events you're interested in, rather than writing code that always executes in the same restricted order.

Regardless of how an event is called, Kylix looks to see if you have written any code to handle that event. If you have, that code is executed; otherwise, the default event handling behavior takes place.

Events can be widget events, such as highlighting a menu item, or system events, such as working with timers or key presses.

Widget events

Widget events are actions that are generated by user interaction with a widget. Widget events generate a signal that is passed onto the CLX component for processing. The CLX component has an associated event handler installed for the signal being passed to it. Examples of widget events are *OnChange* (the user changed text in an edit control), *OnHightlighted* (the user highlighted a menu item on a menu), and *OnReturnPressed* (the user pressed Enter in a memo control). These events are always tied to specific widgets and are defined within those widgets.

System events

System events are events that the operating system generates. For example, the *OnTimer* event (the *TTimer* component issues one of these events whenever a predefined interval has elapsed), the *OnCreate* event (the component is being created), the *OnPaint* event (a component or widget needs to be redrawn), *OnKeyPress* event (a key was pressed on the keyboard), and so on. These are events that the application programmer must respond to if they are not handled as you want them to be.

What is an object?

An object, or *class*, is a data type that encapsulates *data* and *operations on data* in a single unit. Before object-oriented programming, data and operations (functions) were treated as separate elements.

You can begin to understand objects if you understand Object Pascal *records*. Records (analogous to *structures* in C) are made of up fields that contain data, where each field has its own type. Records make it easy to refer to a collection of varied data elements.

Objects are also collections of data elements. But objects—unlike records—contain procedures and functions that operate on their data. These procedures and functions are called *methods*.

An object's data elements are accessed through *properties*. The properties of CLX objects have values that you can change at design time without writing code. If you want a property value to change at runtime, you need to write only a small amount of code.

The combination of data and functionality in a single unit is called *encapsulation*. In addition to encapsulation, object-oriented programming is characterized by *inheritance* and *polymorphism*. Inheritance means that objects derive functionality from other objects (called *ancestors*); objects can modify their inherited behavior. Polymorphism means that different objects derived from the same ancestor support the same method and property interfaces, which often can be called interchangeably.

Examining a Kylix object

When you create a new project, Kylix displays a new form for you to customize. In the Code editor, Kylix declares a new class type for the form and produces the code that creates the new form instance. The generated code looks like this:

```
unit Unit1;
interface

uses SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;

type
  TForm1 = class(TForm) { The type declaration of the form begins here }
  private
    { Private declarations }
  public
    { Public declarations }
  end; { The type declaration of the form ends here }

var
  Form1: TForm1;

implementation { Beginning of implementation part }
{$R *.DFM}
end.{ End of implementation part and unit}
```

The new class type is *TForm1*, and it is derived from type *TForm*, which is also a class.

A class is like a record in that they both contain data fields, but a class also contains methods—code that acts on the object's data. So far, *TForm1* appears to contain no fields or methods, because you haven't added to the form any components (the fields of the new object) and you haven't created any event handlers (the methods of the new object). *TForm1* does contain inherited fields and methods, even though you don't see them in the type declaration.

This variable declaration declares a variable named *Form1* of the new type *TForm1*.

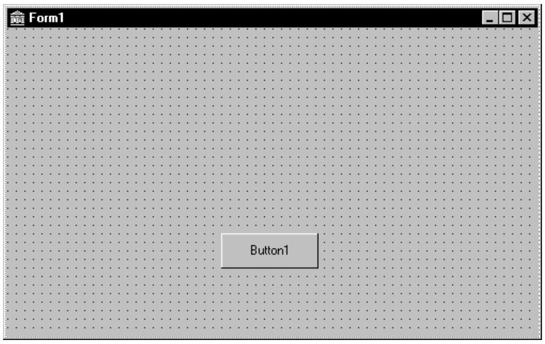
```
var
  Form1: TForm1;
```

Form1 represents an instance, or object, of the class type *TForm1*. You can declare more than one instance of a class type. Each instance maintains its own data, but all instances use the same code to execute methods.

Although you haven't added any components to the form or written any code, you already have a complete Kylix application that you can compile and run. All it does is display a blank form.

Suppose you add a button component to this form and write an *OnClick* event handler that changes the color of the form when the user clicks the button. The result might look like this:

Figure 3.2 A simple form



When the user clicks the button, the form's color changes to green. This is the event-handler code for the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Color := clGreen;
end;
```

Objects can contain other objects as data fields. Each time you place a component on a form, a new field appears in the form's type declaration. If you create the application described above and look at the code in the Code editor, this is what you see:

```
unit Unit1;

interface

uses SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;

type
  TForm1 = class(TForm)
```

```
      Button1: TButton;{ New data field }
      procedure Button1Click(Sender: TObject);{ New method declaration }
    private
      { Private declarations }
    public
      { Public declarations }
    end;

  var
    Form1: TForm1;

  implementation

  {$R *.xfm}

  procedure TForm1.Button1Click(Sender: TObject);{ The code of the new method }
  begin
    Form1.Color := clGreen;
  end;

  end.
```

TForm1 has a *Button1* field that corresponds to the button you added to the form. *TButton* is a class type, so *Button1* refers to an object.

All the event handlers you write in Kylix are methods of the form object. Each time you create an event handler, a method is declared in the form object type. The *TForm1* type now contains a new method, the *Button1Click* procedure, declared within the *TForm1* type declaration. The code that implements the *Button1Click* method appears in the **implementation** part of the unit.

Changing the name of a component

You should always use the Object Inspector to change the name of a component. For example, suppose you want to change a form's name from the default *Form1* to a more descriptive name, such as *ColorBox*. When you change the form's *Name* property in the Object Inspector, the new name is automatically reflected in the form's .xfm file (which you do not usually edit manually) and in the Object Pascal source code that Kylix generates:

```
unit Unit1;

interface

uses SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;

type
  TColorBox = class(TForm){ Changed Form1 to TColorBox }
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  ColorBox: TColorBox;{Changed Form1 to ColorBox }
```

```
implementation

{$R *.xfm}

procedure TColorBox.Button1Click(Sender: TObject);
begin
  Form1.Color := clGreen;{ The reference to Form1 didn't change! }
end;

end.
```

Note that the code in the *OnClick* event handler for the button hasn't changed. Because you wrote the code, you have to update it yourself and correct any references to the form:

```
procedure TColorBox.Button1Click(Sender: TObject);
begin
  ColorBox.Color := clGreen;
end;
```

Inheriting data and code from an object

The *TForm1* object described on page 3-5 seems simple. *TForm1* appears to contain one field (*Button1*), one method (*Button1Click*), and no properties. Yet you can show, hide, or resize of the form, add or delete standard border icons, and set up the form to become part of a Multiple Document Interface (MDI) application. You can do these things because the form has *inherited* all the properties and methods of CLX component *TForm*. When you add a new form to your project, you start with *TForm* and customize it by adding components, changing property values, and writing event handlers. To customize any object, you first derive a new object from the existing one; when you add a new form to your project, Kylix automatically derives a new form from the *TForm* type:

```
TForm1 = class(TForm)
```

A derived object inherits all the properties, events, and methods of the object it derives from. The derived object is called a *descendant* and the object it derives from is called an *ancestor*. If you look up *TForm* in the online Help, you'll see lists of its properties, events, and methods, including the ones that *TForm* inherits from *its* ancestors. An object can have only one immediate ancestor, but it can have many direct descendants.

Scope and qualifiers

Scope determines the accessibility of an object's fields, properties, and methods. All members declared within an object are available to that object and its descendants. Although a method's implementation code appears outside of the object declaration, the method is still within the scope of the object because it is declared within the object's declaration.

When you write code to implement a method that refers to properties, methods, or fields of the object where the method is declared, you don't need to preface those identifiers with the name of the object. For example, if you put a button on a new form, you could write this event handler for the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Color := clFuchsia;
  Button1.Color := clLime;
end;
```

The first statement is equivalent to

```
Form1.Color := clFuchsia
```

You don't need to qualify *Color* with *Form1* because the *Button1Click* method is part of *TForm1*; identifiers in the method body therefore fall within the scope of the *TForm1* instance where the method is called. The second statement, in contrast, refers to the color of the button object (not of the form where the event handler is declared), so it requires qualification.

Kylix creates a separate unit (source code) file for each form. If you want to access one form's components from another form's unit file, you need to qualify the component names, like this:

```
Form2.Edit1.Color := clLime;
```

In the same way, you can access a component's methods from another form. For example,

```
Form2.Edit1.Clear;
```

To access *Form2*'s components from *Form1*'s unit file, you must also add *Form2*'s unit to the **uses** clause of *Form1*'s unit.

The scope of an object extends to the object's descendants. You can, however, redeclare a field, property, or method within a descendant object. Such redeclarations either hide or override the inherited member.

For more information about scope, inheritance, and the **uses** clause, see the *Object Pascal Language Guide*.

Private, protected, public, and published declarations

When you declare a field, property, or method, the new member has a *visibility* indicated by one of the keywords **private**, **protected**, **public**, or **published**. The visibility of a member determines its accessibility to other objects and units.

- A private member is accessible only within the unit where it is declared. Private members are often used within a class to implement other (public or published) methods and properties.
- A protected member is accessible within the unit where its class is declared and within any descendant class, regardless of the descendant class's unit.
- A public member is accessible from wherever the object it belongs to is accessible—that is, from the unit where the class is declared and from any unit that uses that unit.

- A published member has the same visibility as a public member, but the compiler generates runtime type information for published members. Published properties appear in the Object Inspector at design time.

For more information about visibility, see the *Object Pascal Language Guide*.

Using object variables

You can assign one object variable to another object variable if the variables are of the same type or assignment compatible. In particular, you can assign an object variable to another object variable if the type of the variable you are assigning to is an ancestor of the type of the variable being assigned. For example, here is a *TSimpleForm* type declaration and a variable declaration section declaring two variables, *AForm* and *SimpleForm*:

```
type
  TSimpleForm = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  AForm: TForm;
  SimpleForm: TSimpleForm;
```

AForm is of type *TForm*, and *SimpleForm* is of type *TSimpleForm*. Because *TSimpleForm* is a descendant of *TForm*, this assignment statement is legal:

```
AForm := SimpleForm;
```

Suppose you write an event handler for the *OnClick* event of a button. When the button is clicked, the event handler for the *OnClick* event is called. Each event handler has a *Sender* parameter of type *TObject*:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
⋮
end;
```

Because *Sender* is of type *TObject*, any object can be assigned to *Sender*. The value of *Sender* is always the control or component that responds to the event. You can test *Sender* to find the type of component or control that called the event handler using the reserved word **is**. For example,

```
if Sender is TEdit then
  DoSomething
else
  DoSomethingElse;
```

Creating, instantiating, and destroying objects

Many of the objects you use in Kylix, such as buttons and edit boxes, are visible at both design time and runtime. Some, such as common dialog boxes, appear only at runtime. Still others, such as timers and datasource components, have no visual representation at runtime.

You may want to create your own objects. For example, you could create a *TEmployee* object that contains *Name*, *Title*, and *HourlyPayRate* properties. You could then add a *CalculatePay* method that uses the data in *HourlyPayRate* to compute a paycheck amount. The *TEmployee* type declaration might look like this:

```
type
  TEmployee = class(TObject)
  private
    FName: string;
    FTitle: string;
    FHourlyPayRate: Double;
  public
    property Name: string read FName write FName;
    property Title: string read FTitle write FTitle;
    property HourlyPayRate: Double read FHourlyPayRate write FHourlyPayRate;
    function CalculatePay: Double;
  end;
```

In addition to the fields, properties, and methods you've defined, *TEmployee* inherits all the methods of *TObject*. You can place a type declaration like this one in either the **interface** or **implementation** part of a unit, and then create instances of the new class by calling the *Create* method that *TEmployee* inherits from *TObject*:

```
var
  Employee: TEmployee;
begin
  Employee := TEmployee.Create;
end;
```

The *Create* method is called a *constructor*. It allocates memory for a new instance object and returns a reference to the object.

Components on a form are created and destroyed automatically by Kylix. But if you write your own code to instantiate objects, you are responsible for disposing of them as well. Every object inherits a *Destroy* method (called a *destructor*) from *TObject*. To destroy an object, however, you should call the *Free* method (also inherited from *TObject*), because *Free* checks for a **nil** reference before calling *Destroy*. For example,

```
Employee.Free
```

destroys the *Employee* object and deallocates its memory.

Components and ownership

Kylix has a built-in memory-management mechanism that allows one component to assume responsibility for freeing another. The former component is said to *own* the latter. The memory for an owned component is automatically freed when its owner's memory is freed. The owner of a component—the value of its *Owner* property—is determined by a parameter passed to the constructor when the component is created. By default, a form owns all components on it and is in turn owned by the application. Thus, when the application shuts down, the memory for all forms and the components on them is freed.

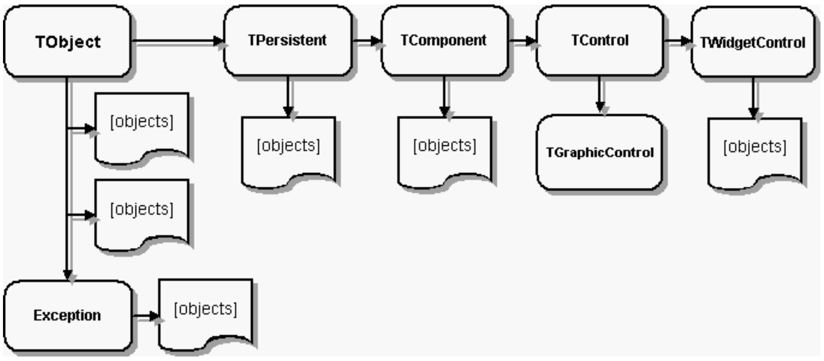
Ownership applies only to *TComponent* and its descendants. If you create, for example, a *TStringList* or *TCollection* object (even if it is associated with a form), you are responsible for freeing the object.

Note Don't confuse a component's *owner* with its *parent*.

Major branches of the CLX hierarchy

Figure 3.3 is a summary of the Borland Component Library for Cross Platform (CLX) that shows the major branches of the inheritance tree.

Figure 3.3 A simplified hierarchy diagram



Six important base classes are shown in the figure, and they are described in the following table:

Table 3.1 Important base classes

Class	Description
TObject	Signifies the base class and ultimate ancestor of everything in CLX. *TObject* encapsulates the fundamental behavior common to all objects in CLX by introducing methods that perform basic functions such as creating, maintaining, and destroying an instance of an object.
Exception	Specifies the base class of all classes that relate to exceptions. *Exception* provides a consistent interface for error conditions, and enables applications to handle error conditions gracefully.
TPersistent	Specifies the base class for all objects that implement properties. Classes under *TPersistent* deal with sending data to streams and allow for the assignment of classes.

Table 3.1 Important base classes (continued)

Class	Description
TComponent	Specifies the base class for all nonvisual components such as *TApplication*. *TComponent* is the common ancestor of all components. This class allows a component to be displayed on the component palette, lets the component own other components, and allows the component to be manipulated directly on a form.
TControl	Represents the base class for all controls that are visible at runtime. *TControl* is the common ancestor of all visual components and provides standard visual controls like position and cursor. This class also provides events that respond to mouse actions.
TWidgetControl	Specifies the base class of all user interface objects also called widgets. Controls under *TWidgetControl* are widget-based controls that can capture keyboard input.

The next few sections generally describe the main CLX branches. For a complete picture of the CLX object hierarchy, refer to the CLX Object Hierarchy wall chart that is included with this product.

TObject branch

The *TObject* branch includes all objects that descend from *TObject* but not from *TPersistent*. All CLX objects descend from *TObject*, an abstract class whose methods define fundamental behavior like construction, destruction, and event handling. Much of the powerful capability of CLX objects are established by the methods that *TObject* introduces. *TObject* encapsulates the fundamental behavior common to all objects in CLX by introducing methods that provide:

- The ability to respond when object instances are created or destroyed.
- Class type and instance information on an object, and runtime type information (RTTI) about its published properties.

TObject is the immediate ancestor of many simple classes that descend from *TObject* but not from one of the other base classes. Classes in this branch have one common, important characteristic, they are transitory. What this means is that these classes do not have a method to save the state that they are in prior to destruction; they are not persistent.

If you write classes of your own in Object Pascal, they should descend from *TObject*. By deriving new classes from CLX's base class (or one of its descendants), you provide your classes with essential functionality and ensure that they work with CLX.

One of the main groups of classes in this branch is the *Exception* class. This class provides a large set of built-in exception classes for automatically handling divide-by-zero errors, file I/O errors, invalid typecasts, and many other exception conditions.

Another type of class in the *TObject* branch are classes that encapsulate data structures, such as:

- *TBits,* a class that stores an "array" of Boolean values
- *TList,* a linked list class
- *TStack,* a class that maintains a last-in first-out array of pointers
- *TQueue,* a class that maintains a first-in first-out array of pointers

TStream is good example of another type of class in this branch. *TStream* is the base class type for stream objects that can read from or write to various kinds of storage media, such as disk files, dynamic memory, and so on. Refer to "Using streams" on page 3-38 to learn more about streams.

TPersistent branch

Objects in this branch of CLX descend from *TPersistent* but not from *TComponent*. *TPersistent* adds persistence to objects. Persistence determines what gets saved with a form file or data module and what gets loaded into the form or data module when it is retrieved from memory.

Objects in this branch implement properties for components. Properties are only loaded and saved with a form if they have an owner. The owner must be some component. This branch introduces the *GetOwner* function which lets you determine the owner of the property.

Objects in this branch are also the first to include a published section where properties can be automatically loaded and saved. A *DefineProperties* method also allows you to indicate how to load and save properties.

Other classes in this branch include:

- *TGraphicsObject,* an abstract base class for graphics objects, such as *TBrush*, *TFont*, and *TPen*.
- *TGraphic,* an abstract base class for objects that can store and display visual images like icons and bitmaps, such as *TBitmap* and *TPicture*.
- *TStrings,* a base class for objects that represent a list of strings.
- *TClipboard*, a class that contains text or graphics that have been cut or copied from an application.
- *TCollection*, *TOwnedCollection*, and *TCollectionItem,* classes that maintain indexed collections of specially defined items.

TComponent branch

TComponent branch contains objects that descend from *TComponent* but not *TControl*. Objects in this branch are components that you can manipulate on forms at design time. They are persistent objects that can do the following:

- Appear on the component palette and can be changed in the form designer.
- Own and manage other components.
- Load and save themselves.

Several methods in *TComponent* dictate how components act during design time and what information gets saved with the component. Streaming is introduced in this branch of CLX. Kylix handles most streaming chores automatically. Properties are persistent if they are published and published properties are automatically streamed.

The *TComponent* class also introduces the concept of ownership that is propagated throughout CLX. Two properties support ownership: *Owner* and *Components*. Every component has an *Owner* property that references another component as its owner. A component may own other components. In this case, all owned components are referenced in the component's *Array* property.

A component's constructor takes a single parameter that is used to specify the new component's owner. If the passed-in owner exists, the new component is added to the owner's Components list. Aside from using the Components list to reference owned components, this property also provides for the automatic destruction of owned components. As long as the component has an owner, it will be destroyed when the owner is destroyed. For example, since *TForm* is a descendant of *TComponent*, all components owned by the form are destroyed and their memory freed when the form is destroyed. This assumes that all of the components on the form clean themselves up properly when their destructors are called.

If a property type is a *TComponent* or a descendant, the streaming system creates an instance of that type when reading it in. If a property type is *TPersistent* but not *TComponent*, the streaming system uses the existing instance available through the property and read values for that instance's properties.

When creating a form file (a file used to store information about the components on the form), the form designer loops through its components array and saves all the components on the form. Each component "knows" how to write its changed properties out to a stream (in this case, a text file). Conversely, when loading the properties of components in the form file, the form designer loops through the components array and loads each component.

The types of classes you'll find in this branch include:

- *TMainMenu,* a class that provides a menu bar and its accompanying drop-down menus for a form.
- *TTimer,* a class that includes timer functions.
- *TOpenDialog*, *TFontDialog*, *TFindDialog*, *TColorDialog*, and so on, provide commonly used dialog boxes.
- *TActionList,* a class that maintains a list of actions used with components and controls, such as menu items and buttons.
- *TScreen,* a class that keeps track of what forms and data modules have been instantiated by the application, the active form, and the active control within that form, the size and resolution of the screen, and the cursors and fonts available for the application to use.

The *TComponent* branch also includes *THandleComponent*. This is the base class for nonvisual components that require a handle to an underlying Qt object such as dialogs and menus.

Components that do not need a visual interface can be derived directly from *TComponent*. To make a tool such as a *TTimer* device, you can derive from *TComponent*. This type of component resides on the component palette but performs internal functions that are accessed through code rather than appearing in the user interface at runtime.

TControl branch

The *TControl* branch consists of components that descend from *TControl* but not *TWidgetControl*. Objects in this branch are controls that are visual objects which the application user can see and manipulate at runtime. All controls have properties, methods, and events in common that relate to how the control looks, such as its position, the cursor associated with the control's widget, methods to paint or move the control, and events to respond to mouse actions. Controls can never receive keyboard input.

Whereas *TComponent* defines behavior for all components, *TControl* defines behavior for all visual controls. This includes drawing routines, standard events, and containership.

There are two basic types of control:

- Those that have a widget of their own
- Those that use the widget of their "parent"

Controls that have their own widget are called "widget-based" controls and descend from *TWidgetControl*. Buttons and check boxes fall into this class.

Controls that use a parent widget are called "graphic" controls and descend from *TGraphicControl*. Image controls fall into this class. The main difference between these types of components is that graphic controls do not have an associated widget, and thus cannot receive the input focus nor can they contain other controls. Because a graphic control does not need a handle, its demand on system resources is lessened, and painting a graphic control is quicker than painting a widget-based control.

TGraphicControl controls must draw themselves and include controls such as:

Table 3.2 Graphic controls

Control	Description
TImage	Displays graphical images.
TBevel	Represents a beveled outline.
TLabel	Displays text on a form.
TPaintBox	Provides a canvas that applications can use for drawing or rendering an image.

Notice that these include common paint routines (*Repaint*, *Invalidate*, and so on) that never need to receive focus.

To create a control that can receive input focus or contain other controls, but which needs a *Canvas* property and a *Paint* method, derive a class from *TCustomControl*.

TWidgetControl branch

The *TWidgetControl* branch includes all controls that descend from *TWidgetControl*. *TWidgetControl* is the base class for all widget-based controls or *widgets*. The term widget comes from combining “window” and “gadget.” A widget is almost anything you use in the user interface of an application. Examples of widgets are buttons, labels, and scroll bars.

The following are features of widgets:

- Widget-based controls have an associated widget.
- Widget-based controls can receive focus while an application is running.
- Other controls may display data, but the user can only use the keyboard to interact with widget-based controls.
- Widget-based controls can contain other controls.
- A control that contains other controls is called a parent. Only a widget-based control can be a parent of one or more child controls.

Descendants of *TWidgetControl* are controls that can receive focus, meaning they can receive keyboard input from the application user. This implies that many more standard events apply to them.

This branch includes both controls that are drawn automatically (including *TEdit*, *TListBox*, *TComboBox*, *TPageControl*, and so on) and custom controls that Kylix must draw (such as *TDBNavigato*r). Direct descendants of *TWidgetControl* typically implement standard controls, like an edit field, a combo box, list box, or page control, and, therefore, already know how to paint themselves. The *TCustomControl* class is provided for components that require a handle but do not encapsulate a standard control that includes the ability to repaint itself.

Refer to Chapter 7, “Working with controls” for details on using controls.

Using components

Many visual components are provided in the development environment itself on the component palette. All visual design work in Kylix takes place on forms. When you open Kylix or create a new project, a blank form is displayed on the screen. You select components from the component palette and drop them onto the form. You design the look and feel of the application’s user interface by arranging the visual components such as buttons and list boxes on the form. Once a visual component is on the form, you can adjust its position, size, and other design-time properties. Kylix takes care of the underlying programming details.

You can also place nonvisible components on forms to capture information from databases, perform calculations, and manage other interactions. Chapter 6, “Developing the application user interface” provides details on using forms such as creating modal forms dynamically, passing parameters to forms, and retrieving data from forms.

Kylix components are grouped functionally on different pages of the component palette. For example, commonly used components such as those to create menus, edit boxes, or buttons are located on the Standard page of the component palette.

At first glance, Kylix's components appear to be just like any other class. But there are differences between components in Kylix and the standard class hierarchies that most programmers work with. Some differences are described here:

- All Kylix components descend from *TComponent*.
- Components are most often used as is and are changed through their properties, rather than serving as "base classes" to be subclassed to add or change functionality. When a component is inherited, it is usually to add specific code to existing event handling member functions.
- Properties of components intrinsically contain runtime type information.
- Components can be added to the component palette in the Kylix user interface and manipulated on a form.

Most Linux system events are handled by Kylix components. When you want to respond to a message, you only need to provide an event handler.

Components on the component palette

The component palette contains a large selection of components that handle a wide variety of programming tasks. You can add, remove, and rearrange components on the palette. You can also create component templates and frames that are made up of several components and add them to the component palette.

The components on the palette are arranged in pages according to their purpose and functionality. Which pages appear in the default configuration depends on the version of Kylix you are running. Table 3.3 lists typical default pages and the types of components they contain.

Table 3.3 Component palette pages

Page name	Contents
Standard	Standard controls, menus
Additional	Specialized controls for use in applications
Common Controls	Common controls used for developing a graphical user interface
Dialogs	Dialog boxes
dbExpress	Database components that use dbExpress
Data Access	Dataset provider, client dataset, and data source components
Data Controls	Visual, data-aware controls
Internet	Components for Internet communication protocols and Web applications
Indy Clients	Components for writing Internet client applications
Indy Servers	Components for writing Internet server applications
Indy Misc	Additional internet components

The online Help provides information about many of the components on the default palette. You can press F1 on the component palette itself or on the component itself after it has been dropped onto a form to display Help.

Text controls

Many applications present text to the user or allow the user to enter text. The type of control used for this purpose depends on the size and format of the information.

Use this component:	When you want users to do this:
TEdit	Edit a single line of text.
TMemo	Edit multiple lines of text.
TMaskEdit	Adhere to a particular format, such as a postal code or phone number.

TEdit and *TMaskEdit* are simple text controls that include a single line text edit box in which you can type information. When the edit box has focus, a blinking insertion point appears.

You can include text in the edit box by assigning a string value to its *Text* property. You control the appearance of the text in the edit box by assigning values to its *Font* property. You can specify the typeface, size, color, and attributes of the font. The attributes affect all of the text in the edit box and cannot be applied to individual characters.

An edit box can be designed to change its size depending on the size of the font it contains. You do this by setting the *AutoSize* property to True. You can limit the number of characters an edit box can contain by assigning a value to the *MaxLength* property.

TMaskEdit is a special edit control that validates the text entered against a mask that encodes the valid forms the text can take. The mask can also format the text that is displayed to the user.

TMemo is for adding several lines of text.

Text control properties

Following are some of the important properties of text controls:

Table 3.4 Text control properties

Property	Description
Text	Determines the text that appears in the edit box or memo control.
Font	Controls the attributes of text written in the edit box or memo control.
AutoSize	Enables the edit box to dynamically change its height depending on the currently selected font.
ReadOnly	Specifies whether the user is allowed to change the text.
MaxLength	Limits the number of characters in simple text controls.

Properties of memo controls

TMemo is another type of edit box, which handles multiple lines of text. The lines in a memo control can extend beyond the right boundary of the edit box, or they can wrap onto the next line. You control whether the lines wrap using the *WordWrap* property.

Memo controls include other properties such as the following:

- *Alignment* specifies how text is aligned (left, right, or center) in the component.
- The *Text* property contains the text in the control. Your application can tell if the text changes by checking the *Modified* property.
- *Lines* contains the text as a list of strings.
- *WordWrap* determines whether the text will wrap at the right margin.
- *WantReturns* determines whether the user can insert hard returns in the text.
- *WantTabs* determines whether the user can insert tabs in the text.
- *SelText* contains the currently selected (highlighted) part of the text.
- *SelStart* and *SelLength* indicate the position and length of the selected part of the text.

At runtime, you can select all the text in the memo with the *SelectAll* method.

Specialized input controls

The following components provide additional ways of capturing input.

Use this component:	When you want users to do this:
TScrollBar	Select values on a continuous range.
TTrackBar	Select values on a continuous range (more visually effective than a scroll bar).
TSpinEdit	Select a value from a spinner widget.

Scroll bars

The scroll bar component creates a scroll bar that you can use to scroll the contents of a window, form, or other control. In the *OnScroll* event handler, you write code that determines how the control behaves when the user moves the scroll bar.

The scroll bar component is not used very often, because many visual components include scroll bars of their own and thus don't require additional coding. For example, *TForm* has *VertScrollBar* and *HorzScrollBar* properties that automatically configure scroll bars on the form. To create a scrollable region within a form, use *TScrollBox*.

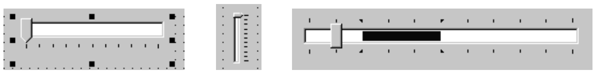
Track bars

A track bar can set integer values on a continuous range. It is useful for adjusting properties like color, volume and brightness. The user moves the slide indicator by dragging it to a particular location or clicking within the bar.

- Use the *Max* and *Min* properties to set the upper and lower range of the track bar.
- Use *SelEnd* and *SelStart* to highlight a selection range. See Figure 3.4.

- The *Orientation* property determines whether the track bar is vertical or horizontal.
- By default, a track bar has one row of ticks along the bottom. Use the *TickMarks* property to change their location. To control the intervals between ticks, use the *TickStyle* property and *SetTick* method.

Figure 3.4 Three views of the track bar component



- *Position* sets a default position for the track bar and tracks the position at runtime.
- By default, users can move one tick up or down by pressing the up and down arrow keys. Set *LineSize* to change that increment.
- Set *PageSize* to determine the number of ticks moved when the user presses the *Page Up* and *Page Down* keys.

Spin edit controls

A spin edit control is also called an up-down widget, little arrows widget, or spin button. This control lets the application user change an integer value in fixed increments, either by clicking the up or down arrow buttons to increase or decrease the value currently displayed, or by typing the value directly into the spin box.

The current value is given by the *Value* property; the increment, which defaults to 1, is specified by the *Increment* property. Use the *Associate* property to attach another component (such as an edit control) to the up-down control.

Buttons and similar controls

Aside from menus, buttons provide the most common way to initiate an action or command in an application. Kylix offers several button-like controls:

Use this component:	To do this:
TButton	Present command choices on buttons with text.
TSpeedButton	Create grouped toolbar buttons.
TCheckBox	Present on/off options.
TRadioButton	Present a set of mutually exclusive choices.
TToolBar	Arrange tool buttons and other controls in rows and automatically adjust their sizes and positions.

Action lists let you centralize responses to user commands (actions) for objects such as menus and buttons that respond to those commands. See “Using action lists” on page 6-13 for details on how to use action lists with buttons and toolbars.

CLX allows you to custom draw buttons individually or application wide. See Chapter 6, “Developing the application user interface.”

Button controls

Users click button controls with the mouse to initiate actions. Buttons are labeled with text that represent the action. The text is specified by assigning a string value to the *Caption* property. Most buttons can also be selected by pressing a key on the keyboard as a keyboard shortcut. The shortcut is shown as an underlined letter on the button.

You can assign an action to a *TButto*n component by creating an *OnClick* event handler for it. Double-clicking a button at design time takes you to the button's *OnClick* event handler in the Code editor.

- Set *Cancel* to *True* if you want the button to trigger its *OnClick* event when the user presses *Esc*.
- Set *Default* to *True* if you want the *Enter* key to trigger the button's *OnClick* event.

Bitmap buttons

A bitmap button (*TBitBtn*) is a button control that presents a bitmap image on its face.

- To choose a bitmap for your button, set the *Glyph* property.
- Use *Kind* to automatically configure a button with a glyph and default behavior.
- By default, the glyph appears to the left of any text. To move it, use the *Layout* property.
- The glyph and text are automatically centered on the button. To move their position, use the *Margin* property. *Margin* determines the number of pixels between the edge of the image and the edge of the button.
- By default, the image and the text are separated by 4 pixels. Use *Spacing* to increase or decrease the distance.
- Bitmap buttons can have 3 states: up, down, and held down. Set the *NumGlyphs* property to 3 to show a different bitmap for each state.

Speed buttons

Speed buttons (*TSpeedButton*), which usually have images on their faces, can function in groups. They are commonly used with panels to create toolbars.

- To make speed buttons act as a group, give the *GroupIndex* property of all the buttons the same nonzero value.
- By default, speed buttons appear in an up (unselected) state. To initially display a speed button as selected, set the *Down* property to *True*.
- If *AllowAllUp* is *True*, all of the speed buttons in a group can be unselected. Set *AllowAllUp* to *False* if you want a group of buttons to act like a radio group.

For more information on speed buttons, refer to subtopics in the section "Adding a toolbar using a panel component" on page 6-32.

Check boxes

A check box is a toggle that lets the user select an on or off state. When the choice is turned on, the check box is checked. Otherwise, the check box is blank. You create check boxes using *TCheckBox*.

- Set *Checked* to *True* to make the box appear checked by default.

- Set *AllowGrayed* to *True* to give the check box three possible states: checked, unchecked, and grayed.
- The *State* property indicates whether the check box is checked (*cbChecked*), unchecked (*cbUnchecked*), or grayed (*cbGrayed*).

Note Check box controls display one of two binary states. The indeterminate state is used when other settings make it impossible to determine the current value for the check box.

Radio buttons

Radio buttons, also called option buttons, present a set of mutually exclusive choices. You can create individual radio buttons using *TRadioButton* or use the *radio group* component (*TRadioGroup*) to arrange radio buttons into groups automatically. You can group radio buttons to let the user select one from a limited set of choices. See "Grouping components" on page 3-26 for more information.

A selected radio button is displayed as a circle filled in the middle. When not selected, the radio button shows an empty circle. Assign the value True or False to the Checked property to change the radio button's visual state.

Toolbars

Toolbars provide an easy way to arrange and manage visual controls. You can create a toolbar out of a panel component and speed buttons, or you can use the *TToolBar* component, then right-click and choose New Button to add tool buttons to the toolbar.

Using the *TToolBar* component has several advantages: buttons on a toolbar automatically maintain uniform dimensions and spacing; other controls maintain their relative position and height; controls can automatically wrap around to start a new row when they do not fit horizontally; and *TToolBar* offers display options like transparency, pop-up borders, and spaces and dividers to group controls.

You can use a centralized set of actions on toolbars and menus, by creating an action list. See "Using action lists" on page 6-13 for details on how to use action lists with buttons and toolbars.

Toolbars can also parent other controls such as edit boxes, combo boxes, and so on.

Splitter controls

A splitter (*TSplitter*) placed between aligned controls allows users to resize the controls. Used with components like panels and group boxes, splitters let you divide a form into several panes with multiple controls on each pane.

After placing a panel or other control on a form, add a splitter with the same alignment as the control. The last control should be client-aligned, so that it fills up the remaining space when the others are resized. For example, you can place a panel at the left edge of a form, set its *Alignment* to *alLeft*, then place a splitter (also aligned to *alLeft*) to the right of the panel, and finally place another panel (aligned to *alLeft* or *alClient*) to the right of the splitter.

Set *MinSize* to specify a minimum size the splitter must leave when resizing its neighboring control. Set *Beveled* to *True* to give the splitter's edge a 3D look.

Handling lists

Lists present the user with a collection of items to select from. Several components display lists:

Use this component:	To display:
TListBox	A list of text strings.
TComboBox	An edit box with a scrollable drop-down list.
TTreeView	A hierarchical list.
TListView	A list of (draggable) items with optional icons, columns, and headings.

Use the nonvisual *TStringList* and *TImageList* components to manage sets of strings and images. For more information about string lists, see "Working with string lists" on page 3-32.

List boxes and check-list boxes

List boxes (*TListBox*) and check-list boxes display lists from which users can select one or more choices from a list of possible options. The choices are represented using text, graphics, or both.

- *Items* uses a *TStrings* object to fill a control with values.
- *ItemIndex* indicates which item in the list is selected.
- *MultiSelect* specifies whether a user can select more than one item at a time.
- *Sorted* determines whether the list is arranged alphabetically.
- *Columns* specifies the number of columns in the list control.
- *ItemHeight* specifies the height of each item in pixels. The *Style* property can cause *ItemHeight* to be ignored.
- The *Style* property determines how a list control displays its items. By default, items are displayed as strings. By changing the value of *Style*, you can create *owner-draw* list boxes that display items graphically or in varying heights. For information on owner-draw controls, see "Adding graphics to controls" on page 7-7.

To create a simple list box,

1. Within your project, drop a list box component from the component palette onto a form.
2. Size the list box and set its alignment as needed.
3. Double-click the right side of the *Items* property or choose the ellipsis button to display the String List Editor.
4. Use the editor to enter free form text arranged in lines for the contents of the list box.
5. Then choose OK.

To let users select multiple items in the list box, you can use the *ExtendedSelect* and *MultiSelect* properties.

Combo boxes

A combo box (*TComboBox*) combines an edit box with a scrollable list. When users enter data into the control—by typing or selecting from the list—the value of the *Text* property changes. If *AutoComplete* is enabled, the application looks for and displays the closest match in the list as the user types the data.

Three types of combo boxes are: standard, drop-down (the default), and drop-down list.

- Use the *Style* property to select the type of combo box you need.
- Use *csDropDown* to create an edit box with a drop-down list. Use *csDropDownList* to make the edit box read-only (forcing users to choose from the list). Set the *DropDownCount* property to change the number of items displayed in the list.
- Use *csSimple* to create a standard combo box with a fixed list that does not close. Be sure to resize the combo box so that the list items are all displayed.
- Use *csOwnerDrawFixed* or *csOwnerDrawVariable* to create *owner-draw* combo boxes that display items graphically or in varying heights. For information on owner-draw controls, see "Adding graphics to controls" on page 7-7.

At runtime, combo boxes work differently on Kylix than they do in Delphi. On Kylix (but not on Delphi), you can add a item to a drop down by entering text and pressing Enter in the edit field of a combo box. You can turn this feature off by setting *InsertMode* to ciNone. It is also possible to add empty (no string) items to the list in the combo box. Also, if you keep pressing the down arrow key, it does not stop at the last item of the combo box list. It cycles around to the top again.

Tree views

A tree view *(TTreeView)* displays items in an indented outline. The control provides buttons that allow nodes to be expanded and collapsed. *Indent* sets the number of pixels horizontally separating items from their parents. You can include icons with items' text labels and display different icons to indicate whether a node is expanded or collapsed. You can also include graphics, such as check boxes, that reflect state information about the items.

To add items to a tree view control at design time, double-click on the control to display the TreeView Items editor. The items you add become the value of the *Items* property.

Tree views can display columns and subitems similar to list views in vsReport mode.

List views

List views, created using *TListView*, display lists in various formats. Use the *ViewStyle* property to choose the kind of list you want:

- *vsList* displays items as labeled icons that cannot be dragged.
- *vsReport* displays items on separate lines with information arranged in columns. The leftmost column contains a small icon and label, and subsequent columns

contain subitems specified by the application. Use the *ShowColumnHeaders* property to display headers for the columns.

Grouping components

A graphical interface is easier to use when related controls and information are presented in groups. Kylix provides several components for grouping components:

Use this component:	When you want this:
TGroupBox	A standard group box with a title.
TRadioGroup	A simple group of radio buttons.
TPanel	A more visually flexible group of controls.
TScrollBox	A scrollable region containing controls.
TTabControl	A set of mutually exclusive notebook-style tabs.
TPageControl	A set of mutually exclusive notebook-style tabs with corresponding pages, each of which may contain other controls.
THeaderControl	Resizable column headers.

Group boxes and radio groups

A group box (*TGroupBox*) arranges related controls on a form. The most commonly grouped controls are radio buttons. After placing a group box on a form, select components from the component palette and place them in the group box. The *Caption* property contains text that labels the group box at runtime.

The radio group component (*TRadioGroup*) simplifies the task of assembling radio buttons and making them work together. To add radio buttons to a radio group, edit the *Items* property in the Object Inspector; each string in *Items* makes a radio button appear in the group box with the string as its caption. The value of the *ItemIndex* property determines which radio button is currently selected. Display the radio buttons in a single column or in multiple columns by setting the value of the *Columns* property. To respace the buttons, resize the radio group component.

Panels

The *TPanel* component provides a generic container for other controls. Panels are typically used to visually group components together on a form. Panels can be aligned with the form to maintain the same relative position when the form is resized. The *BorderWidth* property determines the width, in pixels, of the border around a panel.

You can also place other controls onto a panel and use the *Align* property to ensure proper positioning of all the controls in the group on the form. You can make a panel alTop aligned so that its position will remain in place even if the form is resized.

The look of the panel can be changed to a raised or lowered look by using the *BevelOuter* and *BevelInner* properties. You can vary the values of these properties to create different visual 3-D effects. Note that if you merely want a raised or lowered bevel, you can use the less resource intensive *TBevel* control instead.

You can also use one or more panels to build various status bars or information display areas.

Scroll boxes

Scroll boxes (*TScrollBox*) create scrolling areas within a form. Applications often need to display more information than will fit in a particular area. Some controls—such as list boxes, memos, and forms themselves—can automatically scroll their contents.

Another use of scroll boxes is to create multiple scrolling areas (views) in a window. Views are common in commercial word-processor, spreadsheet, and project management applications. Scroll boxes give you the additional flexibility to define arbitrary scrolling subregions of a form.

Like panels and group boxes, scroll boxes contain other controls, such as *TButton* and *TCheckBox* objects. But a scroll box is normally invisible. If the controls in the scroll box cannot fit in its visible area, the scroll box automatically displays scroll bars.

Another use of a scroll box is to restrict scrolling in areas of a window, such as a toolbar or status bar (*TPanel* components). To prevent a toolbar and status bar from scrolling, hide the scroll bars, and then position a scroll box in the client area of the window between the toolbar and status bar. The scroll bars associated with the scroll box will appear to belong to the window, but will scroll only the area inside the scroll box.

Tab controls

The tab control component (*TTabControl*) creates a set of tabs that look like notebook dividers. You can create tabs by editing the *Tabs* property in the Object Inspector; each string in *Tabs* represents a tab. The tab control is a single panel with one set of components on it. To create a multipage dialog box, use a page control instead.

Page controls

The page control component (*TPageControl*) is a page set suitable for multipage dialog boxes. A page control displays multiple overlapping pages that are *TTabSheet* objects. A page is selected in the user interface by clicking a tab on top of the control.

To create a new page in a page control at design time, right-click the control and choose New Page. At runtime, you add new pages by creating the object for the page and setting its *PageControl* property:

```
NewTabSheet = TTabSheet.Create(PageControl1);
NewTabSheet.PageControl := PageControl1;
```

To access the active page in code, use the *ActivePage* property. To change the active page programmatically, you can set either the *ActivePage* or the *ActivePageIndex* property.

Header controls

A header control (*THeaderControl*) is a is a set of column headers that the user can select or resize at runtime. Edit the control's *Sections* property to add or modify headers. You can place the header sections above columns or fields. For example, header sections might be placed over a list box (*TListBox*).

Providing visual feedback

There are many ways to provide users with information about the state of an application. For example, some components—including *TForm*—have a *Caption* property that can be set at runtime to identify the object. You can also create dialog boxes to display messages. In addition, the following components are especially useful for providing visual feedback at runtime.

Use this component or property:	**To do this:**
TLabel	Display non-editable text.
TStatusBar	Display a status region (usually at the bottom of a window).
TProgressBar	Show the amount of work completed for a particular task.
Hint and *ShowHint*	Activate fly-by or "tooltip" Help.
HelpContext and *HelpFile*	Link context-sensitive online Help.

Labels

Labels (*TLabel*) display text or pixmap fields and are usually placed next to other controls. You place a label on a form when you need to identify or annotate another component such as an edit box or when you want to include text on a form. The standard label component, *TLabel*, is not a widget-based control, so it cannot receive focus.

Label properties include the following:

- *Caption* contains the text string for the label.
- *Font*, *Color*, and other properties determine the appearance of the label. Each label can use only one typeface, size, and color.
- *FocusControl* links the label to another control on the form. If *Caption* includes an accelerator key, the control specified by *FocusControl* receives focus when the user presses the accelerator key.
- *Transparent* determines whether items under the label (such as graphics) are visible.

Labels usually display read-only static text that cannot be changed by the application user. The application can change the text while it is executing by assigning a new value to the *Caption* property. To add a text object to a form that a user can scroll or edit, use *TEdit*.

Status bars

Although you can use a panel to make a status bar, it is simpler to use the status bar component (*TStatusBar*). By default, the status bar's *Align* property is set to *alBottom*, which takes care of both position and size.

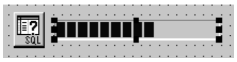
If you only want to display one text string at a time in the status bar, set its *SimplePanel* property to *True* and use the *SimpleText* property to control the text displayed in the status bar.

You can also divide a status bar into several text areas, called panels. To create panels, edit the *Panels* property in the Object Inspector, setting each panel's *Width*, *Alignment*, and *Text* properties from the Panels editor. Each panel's *Text* property contains the text displayed in the panel.

Progress bars

When your application performs a time-consuming operation, you can use a progress bar to show how much of the task is completed. A progress bar (*TProgressBar*) displays a dotted line that grows from left to right.

Figure 3.5 A progress bar



The *Position* property tracks the length of the dotted line. *Max* and *Min* determine the range of *Position*. To make the line grow, increment *Position* by calling the *StepBy* or *StepIt* method. The *Step* property determines the increment used by *StepIt*.

Help and hint properties

Most visual controls can display context-sensitive Help as well as fly-by hints at runtime. The *HelpContext* and *HelpFile* properties establish a Help context number and Help file for the control.

The *Hint* property contains the text string that appears when the user moves the mouse pointer over a control or menu item. To enable hints, set *ShowHint* to *True*; setting *ParentShowHint* to *True* causes the control's *ShowHint* property to have the same value as its parent's.

Grids

Grids display information in rows and columns. If you're writing a database application, use the *TDBGrid* component described in Chapter 15, "Using data controls". Otherwise, use a standard draw grid or string grid.

Draw grids

A draw grid (*TDrawGrid*) displays arbitrary data in tabular format. Write an *OnDrawCell* event handler to fill in the cells of the grid.

- The *CellRect* method returns the screen coordinates of a specified cell, while the *MouseToCell* method returns the column and row of the cell at specified screen coordinates. The *Selection* property indicates the boundaries of the currently selected cells.

- The *TopRow* property determines which row is currently at the top of the grid. The *LeftCol* property determines the first visible column on the left. *VisibleColCount* and *VisibleRowCount* are the number of columns and rows visible in the grid.
- You can change the width or height of a column or row with the *ColWidths* and *RowHeights* properties. Set the width of the grid lines with the *GridLineWidth* property. Add scroll bars to the grid with the *ScrollBars* property.
- You can choose to have fixed or non-scrolling columns and rows with the *FixedCols* and *FixedRows* properties. Assign a color to the fixed columns and rows with the *FixedColor* property.
- The *Options*, *DefaultColWidth*, and *DefaultRowHeight* properties also affect the appearance and behavior of the grid.

String grids

The string grid component is a descendant of *TDrawGrid* that adds specialized functionality to simplify the display of strings. The *Cells* property lists the strings for each cell in the grid; the *Objects* property lists objects associated with each string. All the strings and associated objects for a particular column or row can be accessed through the *Cols* or *Rows* property.

Graphics display

The following components make it easy to incorporate graphics into an application.

Use this component:	To display:
TImage	Graphics files
TShape	Geometric shapes
TBevel	3D lines and frames
TPaintBox	Graphics drawn by your program at runtime

Images

The image component (*TImage*) displays a graphical image, like a bitmap, icon, or drawing. The *Picture* property determines the graphic to be displayed. Use *Center*, *AutoSize*, *Stretch*, and *Transparent* to set display options. For more information, see “Overview of graphics programming” on page 8-1.

Shapes

The shape component (*TShape*) displays a geometric shape. It is not a widget-based control and therefore, it cannot receive user input. The *Shape* property determines which shape the control assumes. To change the shape’s color or add a pattern, use the *Brush* property, which holds a *TBrush* object. How the shape is painted depends on the *Color* and *Style* properties of *TBrush*.

Bevels

The bevel component (*TBevel*) is a line that can appear raised or lowered. Some components, such as *TPanel*, have built-in properties to create beveled borders. When such properties are unavailable, use *TBevel* to create beveled outlines, boxes, or frames.

Paint boxes

The paint box component (*TPaintBox*) allows your application to draw on a form. Write an *OnPaint* event handler to render an image directly on the paint box's *Canvas*. Drawing outside the boundaries of the paint box is prevented. For more information, see “Overview of graphics programming” on page 8-1.

Dialog boxes

The dialog box components on the Dialogs page of the component palette make various dialog boxes available to your applications. These dialog boxes provide applications with a familiar, consistent interface that enables the user to perform common file operations such as opening and saving files. Dialog boxes display and/ or obtain data.

Each dialog box opens when its *Execute* method is called. *Execute* returns a Boolean value: if the user chooses OK to accept any changes made in the dialog box, *Execute* returns *True*; if the user chooses Cancel to escape from the dialog box without making or saving changes, *Execute* returns *False*.

Using open dialog boxes

One of the commonly used dialog box components is *TOpenDialog*. This component is usually invoked by a New or Open menu item under the File option on the main menu bar of a form. The dialog box contains controls that let you select groups of files using a wildcard character and navigate through directories.

The *TOpenDialog* component makes an Open dialog box available to your application. The purpose of this dialog box is to let a user specify a file to open. You use the *Execute* method to display the dialog box.

When the user chooses OK in the dialog box, the user’s file is stored in *TOpenDialog’s FileName* property, which you can then process as you want.

The following code snippet can be placed in an *Action* and linked to the *Action* property of a *TMainMenu* subitem or be placed in the subitem’s *OnClick* event:

```
if OpenDialog1.Execute then
  filename := OpenDialog1.FileName;
```

This code will show the dialog box and if the user presses the OK button, it will copy the name of the file into a previously declared *AnsiString* variable named `filename`.

Using helper objects

CLX includes a variety of nonvisual objects that simplify common programming tasks. This section describes a few helper objects that make it easier to perform the following tasks:

- Working with lists
- Working with string lists
- Creating drawing spaces
- Printing
- Using streams

Working with lists

Several CLX objects provide functionality for creating and managing lists:

Table 3.5 Components for creating and managing lists

Object	Maintains
TList	A list of pointers
TObjectList	A memory-managed list of instance objects
TComponentList	A memory-managed list of components (that is, instances of classes descended from *TComponent*)
TQueue	A first-in first-out list of pointers
TStack	A last-in first-out list of pointers
TObjectQueue	A first-in first-out list of objects
TObjectStack	A last-in first-out list of objects
TClassList	A list of class types
TCollection, *TOwnedCollection*, and *TCollectionItem*	Indexed collections of specially defined items
TStringList	A list of strings

For more information about these objects, see the online reference.

Working with string lists

Applications often need to manage lists of character strings. Examples include items in a combo box, lines in a memo, names of fonts, and names of rows and columns in a string grid. The CLX provides a common interface to any list of strings through an object called *TStrings* and its descendant *TStringList*. *TStringList* implements the abstract properties and methods introduced by *TStrings*, and introduces properties, events, and methods to

- Sort the strings in the list.
- Prohibit duplicate strings in sorted lists.
- Respond to changes in the contents of the list.

In addition to providing functionality for maintaining string lists, these objects allow easy interoperability; for example, you can edit the lines of a memo (which are an instance of *TStrings*) and then use these lines as items in a combo box (also an instance of *TStrings*).

You can also work with string-list objects at runtime to perform such tasks as

- Loading and saving string lists
- Creating a new string list
- Manipulating strings in a list
- Associating objects with a string list

Loading and saving string lists

String-list objects provide *SaveToFile* and *LoadFromFile* methods that let you store a string list in a text file and load a text file into a string list. Each line in the text file corresponds to a string in the list. Using these methods, you could, for example, create a simple text editor by loading a file into a memo component, or save lists of items for combo boxes.

The following example loads a copy of the passwd file into a memo field and makes a backup copy called passwd.bak.

```
procedure EditFile;
var
  FileName: string;{ storage for file name }
begin
  FileName := '/etc/passwd';{ set the file name }
  with Form1.Memo1.Lines do
  begin
    LoadFromFile(FileName);{ load from file }
    SaveToFile(ChangeFileExt(FileName, '.bak'));{ save into backup file }
  end;
end;
```

Creating a new string list

A string list is typically part of a component. There are times, however, when it is convenient to create independent string lists, for example to store strings for a lookup table. The way you create and manage a string list depends on whether the list is short-term (constructed, used, and destroyed in a single routine) or long-term (available until the application shuts down). Whichever type of string list you create, remember that you are responsible for freeing the list when you finish with it.

Short-term string lists

If you use a string list only for the duration of a single routine, you can create it, use it, and destroy it all in one place. This is the safest way to work with string lists. Because the string-list object allocates memory for itself and its strings, you should use a **try...finally** block to ensure that the memory is freed even if an exception occurs.

1 Construct the string-list object.
2 In the **try** part of a **try...finally** block, use the string list.

3 In the **finally** part, free the string-list object.

The following event handler responds to a button click by constructing a string list, using it, and then destroying it.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TempList: TStrings;{ declare the list }
begin
  TempList := TStringList.Create;{ construct the list object }
  try
    { use the string list }
  finally
    TempList.Free;{ destroy the list object }
  end;
end;
```

Long-term string lists

If a string list must be available at any time while your application runs, construct the list at start-up and destroy it before the application terminates.

1. In the unit file for your application's main form, add a field of type *TStrings* to the form's declaration.
2. Write an event handler for the main form's *constructor*, which executes before the form appears. It should create a string list and assign it to the field you declared in the first step.
3. Write an event handler that frees the string list for the form's *OnClose* event.

This example uses a long-term string list to record the user's mouse clicks on the main form, then saves the list to a file before the application terminates.

```
unit Unit1;
interface
uses SysUtils, Classes, QGraphics, QControls, QForms, Qialogs;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  private
    { Private declarations }
  public
    { Public declarations }
    ClickList: TStrings;{ declare the field }
  end;

var
  Form1: TForm1;

implementation
```

```
{$R *.xfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  ClickList := TStringList.Create;{ construct the list }
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  ClickList.SaveToFile(ExtractFilePath(Application.ExeName) + '.LOG')); { save the list }
  ClickList.Free;{ destroy the list object }
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  ClickList.Add(Format('Click at (%d, %d)', [X, Y]));{ add a string to the list }
end;

end.
```

Manipulating strings in a list

Operations commonly performed on string lists include:

- Counting the strings in a list
- Accessing a particular string
- Locating items in a string list
- Iterating through strings in a list
- Adding a string to a list
- Moving a string within a list
- Deleting a string from a list
- Copying a complete string list

Counting the strings in a list

The read-only *Count* property returns the number of strings in the list. Since string lists use zero-based indexes, *Count* is one more than the index of the last string.

Accessing a particular string

The *Strings* array property contains the strings in the list, referenced by a zero-based index. Because *Strings* is the default property for string lists, you can omit the *Strings* identifier when accessing the list; thus

```
StringList1.Strings[0] := 'This is the first string.';
```

is equivalent to

```
StringList1[0] := 'This is the first string.';
```

Locating items in a string list

To locate a string in a string list, use the *IndexOf* method. *IndexOf* returns the index of the first string in the list that matches the parameter passed to it, and returns –1 if the parameter string is not found. *IndexOf* finds exact matches only; if you want to match partial strings, you must iterate through the string list yourself.

For example, you could use *IndexOf* to determine whether a given file name is found among the *Items* of a list box:

```
if FileListBox1.Items.IndexOf('solitaire.so') > -1 ...
```

Iterating through strings in a list

To iterate through the strings in a list, use a **for** loop that runs from zero to *Count* – 1.

This example converts each string in a list box to uppercase characters.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Index: Integer;
begin
  for Index := 0 to ListBox1.Items.Count - 1 do
    ListBox1.Items[Index] := UpperCase(ListBox1.Items[Index]);
end;
```

Adding a string to a list

To add a string to the end of a string list, call the *Add* method, passing the new string as the parameter. To insert a string into the list, call the *Insert* method, passing two parameters: the string and the index of the position where you want it placed. For example, to make the string “Three” the third string in a list, you would use:

```
Insert(2, 'Three');
```

To append the strings from one list onto another, call *AddStrings*:

```
StringList1.AddStrings(StringList2);  { append the strings from StringList2 to StringList1 }
```

Moving a string within a list

To move a string in a string list, call the *Move* method, passing two parameters: the current index of the string and the index you want assigned to it. For example, to move the third string in a list to the fifth position, you would use:

```
Move(2, 4)
```

Deleting a string from a list

To delete a string from a string list, call the list’s *Delete* method, passing the index of the string you want to delete. If you don’t know the index of the string you want to delete, use the *IndexOf* method to locate it. To delete all the strings in a string list, use the *Clear* method.

This example uses *IndexOf* and *Delete* to find and delete a string:

```
with ListBox1.Items do
begin
  BIndex := IndexOf('bureaucracy');
  if BIndex > -1 then
    Delete(BIndex);
end;
```

Copying a complete string list

You can use the *Assign* method to copy strings from a source list to a destination list, overwriting the contents of the destination list. To append strings without overwriting the destination list, use *AddStrings*. For example,

```
Memo1.Lines.Assign(ComboBox1.Items);    { overwrites original strings }
```

copies the lines from a combo box into a memo (overwriting the memo), while

```
Memo1.Lines.AddStrings(ComboBox1.Items);   { appends strings to end }
```

appends the lines from the combo box to the memo.

When making local copies of a string list, use the *Assign* method. If you assign one string-list variable to another—

```
StringList1 := StringList2;
```

—the original string-list object will be lost, often with unpredictable results.

Associating objects with a string list

In addition to the strings stored in its *Strings* property, a string list can maintain references to objects, which it stores in its *Objects* property. Like *Strings*, *Objects* is an array with a zero-based index. The most common use for *Objects* is to associate bitmaps with strings for owner-draw controls.

Use the *AddObject* or *InsertObject* method to add a string and an associated object to the list in a single step. *IndexOfObject* returns the index of the first string in the list associated with a specified object. Methods like *Delete*, *Clear,* and *Move* operate on both strings and objects; for example, deleting a string removes the corresponding object (if there is one).

To associate an object with an existing string, assign the object to the *Objects* property at the same index. You cannot add an object without adding a corresponding string.

Creating drawing spaces

For objects that must render their own images, you can create abstract drawing spaces using *TCanvas*. The *TCanvas* object encapsulates a paint device (Qt painter), which handles all drawing for both forms, visual containers (such as panels) and the printer object (covered in "Printing" on page 3-38). Using the canvas object, you no longer have to worry about allocating pens, brushes, palettes, and so on—all the allocation and deallocation are handled for you.

TCanvas includes a large number of primitive graphics routines to draw lines, shapes, polygons, and fonts onto any control that contains a canvas. For example, the following code shows a button event handler that draws a line from the upper left corner to the middle of the form and outputs text onto the form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Canvas.Pen.Color := clBlue;
  Canvas.MoveTo( 10, 10 );
  Canvas.LineTo( 100, 100 );
```

```
    Canvas.Brush.Color := clred;
    Canvas.Font.Name := 'Arial';
    Canvas.TextOut( Canvas.PenPos.x, Canvas.PenPos.y,'This is the end of the line');
  end;
```

TCanvas is used in Kylix where drawing is required or possible, and it makes drawing graphics both fail-safe and easy.

See the online help under *TCanvas* for a complete listing of properties and methods.

Printing

The *TPrinter* object is a paint device that paints on a printer. It generates postscript and sends that to lpr, lp, or another print command. The printer object uses *TCanvas* (which is identical to the form's *TCanvas*) so that anything that can be drawn on a form can be printed as well. To print an image, call the *BeginDoc* method followed by whatever canvas graphics you want to print (including text through the *TextOut* method) and send the job to the printer by calling the *EndDoc* method.

This example uses a button and a memo on a form. When the user clicks the button, the content of the memo is printed with a 200-pixel border around the page.

To run this example successfully, add Printers to your **uses** clause.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r: TRect;
  i: Integer;
begin
  with Printer do
    begin
      r := Rect(200,200,(Pagewidth - 200),(PageHeight - 200));
      BeginDoc;
      for i := 0 to Memo1.Lines.Count do
       Canvas.TextOut(200,200 + (i *
Canvas.TextHeight(Memo1.Lines.Strings[i])),
                                  Memo1.Lines.Strings[i]);
      Canvas.Brush.Color := clBlack;
      Canvas.FrameRect(r);
      EndDoc;
    end;
end;
```

For more information on the use of the *TPrinter* object, look in the online help under *TPrinter*.

Using streams

Streams are just ways of reading and writing data. Steams provide a common interface for reading and writing to different media such as memory, strings, sockets, and blob streams.

In the following streaming example, one file is copied to another one using streams. The application includes two edit controls (From and To) and a Copy File button.

```
procedure TForm1.CopyFileClick(Sender: TObject);
var
  stream1, stream2:TStream;
begin
  stream1:=TFileStream.Create(From.Text,fmOpenRead or fmShareDenyWrite);
  try
    stream2 := TFileStream.Create(To.Text fmOpenCreate or fmShareDenyRead);
    try
      stream2.CopyFrom(Stream1,Stream1.Size);
    finally
      stream2.Free;
  finally
    stream1.Free
end;
```

Use specialized stream objects to read or write to storage media. Each descendant of *TStream* implements methods for accessing a particular medium, such as disk files, dynamic memory, and so on. *TStream* descendants include *TFileStream*, *TStringStream*, and *TMemoryStream*. In addition to methods for reading and writing, these objects permit applications to seek to an arbitrary position in the stream. Properties of *TStream* provide information about the stream, such as size and current position.

Chapter 4

Common programming tasks

This chapter discusses how to perform some of the common programming tasks in Kylix:

- Understanding classes
- Defining classes
- Handling exceptions
- Using interfaces
- Working with strings
- Working with files

Understanding classes

A *class* is an abstract definition of properties, methods, events, and class members (such as variables local to the class). When you create an instance of a class, this instance is called an object. The term object is often used more loosely in the Kylix documentation and where the distinction between a class and an instance of the class is not important, the term "object" may also refer to a class.

Although Kylix includes many classes in CLX, you are likely to need to create additional classes if you are writing object-oriented programs. The classes you write must descend from *TObject* or one of its descendants. A class type declaration contains three possible sections that control the accessibility of its fields and methods:

```
Type
  TClassName = Class(TObject)
    public
      {public fields}
      {public methods}
    protected
      {protected fields}
      {protected methods}
    private
      {private fields}
      {private methods}
end;
```

- The public section declares fields and methods with no access restrictions; class instances and descendant classes can access these fields and methods.
- The protected section includes fields and methods with some access restrictions; descendant classes can access these fields and methods.
- The private section declares fields and methods that have rigorous access restrictions; they cannot be accessed by class instances or descendant classes.

The advantage of using classes comes from being able to create new classes as descendants of existing ones. Each descendant class inherits the fields and methods of its parent and ancestor classes. You can also declare methods in the new class that override inherited ones, introducing new, more specialized behavior.

The general syntax of a descendant class is as follows:

```
Type
  TClassName = Class (TParentClass)
    public
      {public fields}
      {public methods}
    protected
      {protected fields}
      {protected methods}
    private
      {private fields}
      {private methods}
end;
```

If no parent class name is specified, the class inherits directly from *TObject*. *TObject* defines only a handful of methods, including a basic constructor and destructor.

For more information about the syntax, language definitions, and rules for classes, see the *Object Pascal Language Guide* online Help on Class types.

Defining classes

Kylix allows you to declare classes that implement the programming features you need to use in your application. Some versions of Kylix include a feature called class completion that simplifies the work of defining and implementing new classes by generating skeleton code for the class members you declare.

To define a class,

1. In the IDE, start with a project open and choose File | New Unit to create a new unit where you can define the new class.
2. Add the **uses** clause and **type** section to the **interface** section.
3. In the **type** section, write the class declaration. You need to declare all the member variables, properties, methods, and events.

```
TMyClass = class; {This implicitly descends from TObject}
public
  .
  .
  .
  .
  .
  .
private
  .
  .
  .
published {If descended from TPersistent or below}
  .
  .
  .
```

Note You should only include the published section for objects descended from *TPersistent* or below.

If you want the class to descend from a specific class, you need to indicate that class in the definition:

```
TMyClass = class(TParentClass); {This descends from TParentClass}
```

For example:

```
type TMyButton = class(TButton)
  property Size: Integer;
  procedure DoSomething;
end;
```

If your version of Kylix includes class completion: place the cursor within a method definition in the **interface** section and press *Ctrl+Shift+C* (or right-click and select Complete Class at Cursor). Kylix completes any unfinished property declarations and creates the empty methods you need in the **implementation** section. (If you do not have class completion, you'll need to write the code yourself, completing property declarations and writing the methods.)

Given the example above, if you have class completion, Kylix adds **read** and **write** specifiers to your interface declaration, including any supporting fields or methods:

```
type TMyButton = class(TButton)
  property Size: Integer read FSize write SetSize;
  procedure DoSomething;
private
  FSize: Integer;
  procedure SetSize(const Value: Integer);
```

It also adds the following code to the **implementation** section of the unit.

```
{ TMyButton }
procedure TMyButton.DoSomething;
begin

end;
procedure TMyButton.SetSize(const Value: Integer);
begin
FSize := Value;
end;
```

4 Fill in the methods. For example, to make it so the button beeps when you call the DoSomething method, add the Beep between **begin** and **end**.

```
{ TMyButton }
procedure TMyButton.DoSomething;
begin
  Beep;
end;

procedure TMyButton.SetSize(const Value: Integer);
begin
  FSize := Value;
  DoSomething;
end;
```

Note that the button also beeps when you call SetSize to change the size of the button.

For more information about the syntax, language definitions, and rules for classes and methods, see the *Object Pascal Language Guide* online Help on Class types and methods.

Handling exceptions

Kylix provides a mechanism to handle errors in a consistent manner. Exception handling allows the application to recover from errors if possible and to shut down if need be, without losing data or resources. Error conditions in Kylix are indicated by exceptions. This section describes the following tasks for using exceptions to create safe applications:

- Protecting blocks of code
- Protecting resource allocations
- Handling RTL exceptions
- Handling component exceptions
- Exception handling with external sources
- Silent exceptions
- Defining your own exceptions

Protecting blocks of code

To make your applications robust, your code needs to recognize exceptions when they occur and respond to them. If you don't specify a response, the application will present a message box describing the error. Your job, then, is to recognize places where errors might happen, and define responses, particularly in areas where errors could cause the loss of data or system resources.

When you create a response to an exception, you do so on blocks of code. When you have a series of statements that all require the same kind of response to errors, you can group them into a block and define error responses that apply to the whole block.

Blocks with specific responses to exceptions are called protected blocks because they can guard against errors that might otherwise either terminate the application or damage data.

To protect blocks of code you need to understand

- Responding to exceptions
- Exceptions and the flow of control
- Nesting exception responses

Responding to exceptions

When an error condition occurs, the application raises an exception, meaning it creates an exception object. Once an exception is raised, your application can execute cleanup code, handle the exception, or both.

Executing cleanup code

The simplest way to respond to an exception is to guarantee that some cleanup code is executed. This kind of response doesn't correct the condition that caused the error but lets you ensure that your application doesn't leave its environment in an unstable state. You typically use this kind of response to ensure that the application frees allocated resources, regardless of whether errors occur.

Handling an exception

This is a specific response to a specific kind of exception. Handling an exception clears the error condition and destroys the exception object, which allows the application to continue execution. You typically define exception handlers to allow your applications to recover from errors and continue running. Types of exceptions you might handle include attempts to open files that don't exist, writing to full disks, or calculations that exceed legal bounds. Some of these, such as "File not found," are easy to correct and retry, while others, such as running out of memory, might be more difficult for the application or the user to correct.

To handle exceptions effectively, you need to understand the following:

- Creating an exception handler
- Exception handling statements
- Using the exception instance
- Scope of exception handlers
- Providing default exception handlers
- Handling classes of exceptions
- Reraising the exception

Exceptions and the flow of control

Object Pascal makes it easy to incorporate error handling into your applications because exceptions don't get in the way of the normal flow of your code. In fact, by moving error checking and error handling out of the main flow of your algorithms, exceptions can simplify the code you write.

When you declare a protected block, you define specific responses to exceptions that might occur within that block. When an exception occurs in that block, execution immediately jumps to the response you defined, then leaves the block.

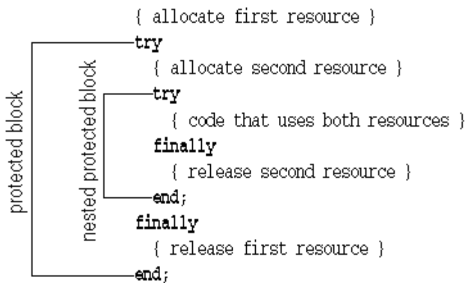
Example The following code that includes a protected block. If any exception occurs in the protected block, execution jumps to the exception-handling part, which beeps. Execution resumes outside the block.

```
try
  AssignFile(F, FileName);
  Reset(F);
  ⋮
except
  on Exception do Beep;
end;
  ⋮ { execution resumes here, outside the protected block }
```

Nesting exception responses

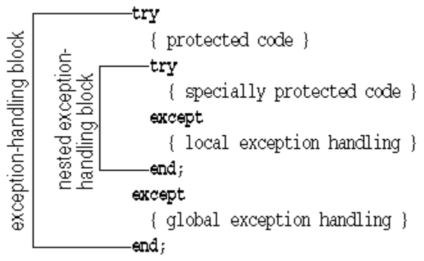
Your code defines responses to exceptions that occur within blocks. Because Pascal allows you to nest blocks of code, you can customize responses even within blocks that already contain customized responses.

In the simplest case, for example, you can protect a resource allocation, and within that protected block, define blocks that allocate and protect other resources. Conceptually, that might look something like this:



```
{ allocate first resource }
try
  { allocate second resource }
  try
    { code that uses both resources }
  finally
    { release second resource }
  end;
finally
  { release first resource }
end;
```

You can also use nested blocks to define local handling for specific exceptions that overrides the handling in the surrounding block. Conceptually, that looks something like this:



```
try
  { protected code }
  try
    { specially protected code }
  except
    { local exception handling }
  end;
except
  { global exception handling }
end;
```

You can also mix different kinds of exception-response blocks, nesting resource protections within exception handling blocks and vice versa.

Protecting resource allocations

One key to having a robust application is ensuring that if it allocates resources, it also releases them, even if an exception occurs. For example, if your application allocates memory, you need to make sure it eventually releases the memory, too. If it opens a file, you need to make sure it closes the file later.

Keep in mind that exceptions don't come just from your code. A call to an RTL routine, for example, or another component in your application might raise an exception. Your code needs to ensure that if these conditions occur, you release allocated resources.

To protect resources effectively, you need to understand the following:

- What kind of resources need protection?
- Creating a resource protection block

What kind of resources need protection?

Under normal circumstances, you can ensure that an application frees allocated resources by including code for both allocating and freeing. When exceptions occur, however, you need to ensure that the application still executes the resource-freeing code.

Some common resources that you should always be sure to release are:

- Files
- Memory
- Objects

Example The following event handler allocates memory, then generates an error, so it never executes the code to free the memory:

```
procedure TForm1.Button1Click(Sender: TComponent);
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  AnInteger := 10 div ADividend;{ this generates an error }
  FreeMem(APointer, 1024);{ it never gets here }
end;
```

Although most errors are not that obvious, the example illustrates an important point: When the division-by-zero error occurs, execution jumps out of the block, so the *FreeMem* statement never gets to free the memory.

To guarantee that the *FreeMem* gets to free the memory allocated by *GetMem*, you need to put the code in a resource-protection block.

Creating a resource protection block

To ensure that you free allocated resources, even in case of an exception, you embed the resource-using code in a protected block, with the resource-freeing code in a special part of the block. Here's an outline of a typical protected resource allocation:

```
{ allocate the resource }
try
  { statements that use the resource }
finally
  { free the resource }
end;
```

The key to the **try..finally** block is that the application always executes any statements in the **finally** part of the block, even if an exception occurs in the protected block. When any code in the **try** part of the block (or any routine called by code in the **try** part) raises an exception, execution halts at that point. Once an exception handler is found, execution jumps to the **finally** part, which is called the cleanup code. After the **finally** part is executed, the exception handler is called. If no exception occurs, the cleanup code is executed in the normal order, after all the statements in the **try** part.

Example The following code illustrates an event handler that allocates memory and generates an error, but still frees the allocated memory:

```
procedure TForm1.Button1Click(Sender: TComponent);
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;

begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  try
    AnInteger := 10 div ADividend;{ this generates an error }
  finally
    FreeMem(APointer, 1024);{ execution resumes here, despite the error }
  end;
end;
```

The statements in the **finally** block do not depend on an exception occurring. If no statement in the **try** part raises an exception, execution continues through the **finally** block.

Handling RTL exceptions

When you write code that calls routines in the runtime library (RTL), such as mathematical functions or file-handling procedures, the RTL reports errors back to your application in the form of exceptions. By default, RTL exceptions generate a message that the application displays to the user. You can define your own exception handlers to handle RTL exceptions in other ways.

There are also silent exceptions that do not, by default, display a message.

RTL exceptions are handled like any other exceptions. To handle RTL exceptions effectively, you need to understand the following:

- What are RTL exceptions?
- Creating an exception handler
- Exception handling statements
- Using the exception instance
- Scope of exception handlers
- Providing default exception handlers
- Handling classes of exceptions
- Reraising the exception

What are RTL exceptions?

The runtime library's exceptions are defined in the SysUtils unit, and they all descend from a generic exception-object type called *Exception*. By convention, all exception classes should descend from *Exception*. It provides the string for the message that RTL exceptions display by default.

Several kinds of exceptions are raised by the RTL, as described in the following table.

Table 4.1 RTL exceptions

Error type	Cause	Meaning
Input/output	Error accessing a file or I/O device	Most I/O exceptions are related to error codes returned when accessing a file.
Heap	Error using dynamic memory	Heap errors can occur when there is insufficient memory available, or when an application disposes of a pointer that points outside the heap.
Integer math	Illegal operation on integer-type expressions	Errors include division by zero, numbers or expressions out of range, and overflows.
Floating-point math	Illegal operation on real-type expressions	Floating-point errors can come from either a hardware coprocessor or the software emulator. Errors include invalid instructions, division by zero, and overflow or underflow.
Typecast	Invalid typecasting with the **as** operator	Objects can only be typecast to compatible types.
Conversion	Invalid type conversion	Type-conversion functions such as IntToStr, StrToInt, and StrToFloat raise conversion exceptions when the parameter cannot be converted to the desired type.
Hardware	System condition	Hardware exceptions indicate that either the processor or the user generated some kind of error condition or interruption, such as an access violation, stack overflow, or keyboard interrupt.
Variant	Illegal type coercion	Errors can occur when referring to variants in expressions where the variant cannot be coerced into a compatible type.

For a list of all of the RTL exception types, see the SysUtils unit.

Creating an exception handler

An exception handler is code that handles a specific exception or exceptions that occur within a protected block of code. In general Kylix programming, it is very rare that you will need to write an exception handler. Most exceptions can be handled using **try..finally** blocks as described in “Protecting blocks of code” on page 4-4 and “Protecting resource allocations” on page 4-7.

To define an exception handler, embed the code you want to protect in an exception-handling block and specify the exception handling statements in the **except** part of the block. Here is an outline of a typical exception-handling block:

```
try
  { statements you want to protect }
except
  { exception-handling statements }
end;
```

The application executes the statements in the **except** part only if an exception occurs during execution of the statements in the **try** part. Execution of the **try** part statements includes routines called by code in the **try** part. That is, if code in the **try** part calls a routine that doesn't define its own exception handler, execution returns to the exception-handling block, which handles the exception.

When a statement in the **try** part raises an exception, execution immediately jumps to the **except** part, where it steps through the specified exception-handling statements, or exception handlers, until it finds a handler that applies to the current exception.

Once the application locates an exception handler that handles the exception, it executes the statement, then automatically destroys the exception object. Execution continues at the end of the current block.

Exception handling statements

Each **on** statement in the **except** part of a **try..except** block defines code for handling a particular kind of exception. The form of these exception-handling statements is as follows:

```
on <type of exception> do <statement>;
```

Example You can define an exception handler for division by zero to provide a default result:

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
  try
    Result := Sum div NumberOfItems;{ handle the normal case }
  except
    on EDivByZero do Result := 0;{ handle the exception only if needed }
  end;
end;
```

Note that this is clearer than having to test for zero every time you call the function. Here's an equivalent function that doesn't take advantage of exceptions:

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
```

```
    if NumberOfItems <> 0 then{ always test }
      Result := Sum div NumberOfItems{ use normal calculation }
    else Result := 0;{ handle exceptional case }
  end;
```

The difference between these two functions really defines the difference between programming with exceptions and programming without them. This example is quite simple, but you can imagine a more complex calculation involving hundreds of steps, any one of which could fail if one of dozens of inputs were invalid.

By using exceptions, you can spell out the “normal” expression of your algorithm, then provide for those exceptional cases when it doesn't apply. Without exceptions, you have to test every single time to make sure you're allowed to proceed with each step in the calculation.

Using the exception instance

Most of the time, an exception handler doesn't need any information about an exception other than its type, so the statements following **on..do** are specific only to the type of exception. In some cases, however, you might need some of the information contained in the exception instance.

To read specific information about an exception instance in an exception handler, you use a special variation of on..do that gives you access to the exception instance. The special form requires that you provide a temporary variable to hold the instance.

Example If you create a new project that contains a single form, you can add a scroll bar and a command button to the form. Double-click the button and add the following line to its click-event handler:

```
ScrollBar1.Max := ScrollBar1.Min - 1;
```

That line raises an exception because the maximum value of a scroll bar must always exceed the minimum value. The default exception handler for the application opens a dialog box containing the message in the exception object. You can override the exception handling in this handler and create your own message box containing the exception's message string:

```
try
  ScrollBar1.Max := ScrollBar1.Min - 1;
except
  on E: EInvalidOperation do
    MessageDlg('Ignoring exception: ' + E.Message, mtInformation, [mbOK], 0);
end;
```

The temporary variable (E in this example) is of the type specified after the colon (*EInvalidOperation* in this example). You can use the as operator to typecast the exception into a more specific type if needed.

Note Never destroy the temporary exception object. Handling an exception automatically destroys the exception object. If you destroy the object yourself, the application attempts to destroy the object again, generating an access violation.

Scope of exception handlers

You do not need to provide handlers for every kind of exception in every block. In fact, you only need handlers for exceptions that you want to handle specially within a particular block.

If a block does not handle a particular exception, execution leaves that block and returns to the block that contains the code that called the block, with the exception still raised. This process repeats with increasingly broad scope until either execution reaches the outermost scope of the application or a block at some level handles the exception.

Providing default exception handlers

You can provide a single default exception handler to handle any exceptions you haven't provided specific handlers for. To do that, you add an else part to the **except** part of the exception-handling block:

```
try
  { statements }
except
  on ESomething do
    { specific exception-handling code };
  else
    { default exception-handling code };
end;
```

Adding default exception handling to a block guarantees that the block handles every exception in some way, thereby overriding all handling from the containing block.

Caution It is not advisable to use this all-encompassing default exception handler. The **else** clause handles all exceptions, including those you know nothing about. In general, your code should handle only exceptions you actually know how to handle. If you want to handle cleanup and leave the exception handling to code that has more information about the exception and how to handle it, then you can do so use an enclosing **try..finally** block:

```
try
  try
  { statements }
  except
    on ESomething do { specific exception-handling code };
  end;
finally
  {cleanup code };
end;
```

For another approach to augmenting exception handling, see Reraising the exception.

Handling classes of exceptions

Because exception objects are part of a hierarchy, you can specify handlers for entire parts of the hierarchy by providing a handler for the exception class from which that part of the hierarchy descends.

Example The following block outlines an example that handles all integer math exceptions specially:

```
try
  { statements that perform integer math operations }
except
  on EIntError do { special handling for integer math errors };
end;
```

You can still specify specific handlers for more specific exceptions, but you need to place those handlers above the generic handler, because the application searches the handlers in the order they appear in, and executes the first applicable handler it finds. For example, this block provides special handling for range errors, and other handling for all other integer math errors:

```
try
  { statements performing integer math }
except
  on ERangeError do { out-of-range handling };
  on EIntError do { handling for other integer math errors };
end;
```

Note that if the handler for *EIntError* came before the handler for *ERangeError*, execution would never reach the specific handler for *ERangeError* because *ERangeError* descends from *EIntError*.

Reraising the exception

Sometimes when you handle an exception locally, you actually want to augment the handling in the enclosing block, rather than replacing it. Of course, when your local handler finishes its handling, it destroys the exception instance, so the enclosing block's handler never gets to act. You can, however, prevent the handler from destroying the exception, giving the enclosing handler a chance to respond.

Example When an exception occurs, you might want to record the error in a log file, then proceed with the standard handling. To do that, you declare a local exception handler that displays the message then calls the reserved word **raise**. This is called reraising the exception, as shown in this example:

```
try
  { statements }
  try
    { special statements }
  except
    on ESomething do
    begin
      { handling for only the special statements }
      raise;{ reraise the exception }
    end;
  end;
```

```
except
  on ESomething do ...;{ handling you want in all cases }
end;
```

If code in the { statements } part raises an *ESomething* exception, only the handler in the outer **except** part executes. However, if code in the { special statements } part raises *ESomething*, the handling in the inner **except** part is executed, followed by the more general handling in the outer **except** part.

By reraising exceptions, you can easily provide special handling for exceptions in special cases without losing (or duplicating) the existing handlers.

Handling component exceptions

Kylix's components raise exceptions to indicate error conditions. Most component exceptions indicate programming errors that would otherwise generate a runtime error. The mechanics of handling component exceptions are no different than handling RTL exceptions.

Example A common source of errors in components is range errors in indexed properties. For example, if a list box has three items in its list (0..2) and your application attempts to access item number 3, the list box raises a “List index out of bounds” exception.

The following event handler contains an exception handler to notify the user of invalid index access in a list box:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add('a string');{ add a string to list box }
  ListBox1.Items.Add('another string');{ add another string... }
  ListBox1.Items.Add('still another string');{ ...and a third string }
  try
    Caption := ListBox1.Items[3];{ set form caption to fourth string in list box }
  except
    on EStringListError do
      MessageDlg('List box contains fewer than four strings', mtWarning, [mbOK], 0);
  end;
end;
```

If you click the button once, the list box has only three strings, so accessing the fourth string (Items[3]) raises an exception. Clicking a second time adds more strings to the list, so it no longer causes the exception.

Exception handling with external sources

HandleException provides default handling of exceptions for the application. Normally, you do not need to call *TApplication.HandleException*. However, you may need it when writing shared object files or callback functions. You can use *TApplication.HandleException* to block an exception from escaping from your code particularly when the code is being called from an external source that does not support exceptions.

For example, if an exception passes through all the **try** blocks in the application code, the application automatically calls the *HandleException* method, which displays a dialog box indicating that an error has occurred. You can use *HandleException* in this fashion:

```
try
  { statements }
except
  Application.HandleException(Self);
end;
```

For all exceptions but *EAbort*, *HandleException* calls the *OnException* event handler, if one exists. Therefore, if you want to both handle the exception, and provide this default behavior as CLX does, you can add a call to *HandleException* to your code:

```
try
  { special statements }
except
  on ESomething do
  begin
    { handling for only the special statements }
    Application.HandleException(Self);{ call HandleException }
  end;
end;
```

Note Do not call HandleException from within a thread's exception handling code.

For more information, search for exception handling routines in the Help index.

Silent exceptions

Kylix applications handle most exceptions that your code doesn't specifically handle by displaying a message box that shows the message string from the exception object. You can also define "silent" exceptions that do not, by default, cause the application to show the error message.

Silent exceptions are useful when you don't intend to report an exception to the user, but you want to abort an operation. Aborting an operation is similar to using the *Break* or *Exit* procedures to break out of a block, but can break out of several nested levels of blocks.

Silent exceptions all descend from the standard exception type *EAbort*. The default exception handler for Kylix CLX applications displays the error-message dialog box for all exceptions that reach it except those descended from *EAbort*.

Note For console applications, an error-message dialog is displayed on any unhandled *EAbort* exceptions.

There is a shortcut for raising silent exceptions. Instead of manually constructing the object, you can call the *Abort* procedure. *Abort* automatically raises an *EAbort* exception, which will break out of the current operation without displaying an error message.

Example The following code shows a simple example of aborting an operation. On a form containing an empty list box and a button, attach the following code to the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 10 do{ loop ten times }
  begin
    ListBox1.Items.Add(IntToStr(I));{ add a numeral to the list }
    if I = 7 then Abort;{ abort after the seventh one }
  end;
end;
```

Defining your own exceptions

In addition to protecting your code from exceptions generated by the runtime library and various components, you can use the same mechanism to manage exception conditions in your own code.

To use exceptions in your code, you need to complete these steps:

- Declaring an exception object type
- Raising an exception

Declaring an exception object type

As a convention, all exception classes should be derived from *Exception* or one of the other standard exceptions. That way, if you raise your new exception in a block of code that isn't protected by a specific exception handler for that exception, one of the standard handlers will handle it instead.

Example For example, consider the following declaration:

```
type
  EMyException = class(Exception);
```

If you raise *EMyException* but don't provide a specific handler for it, a handler for *Exception* (or a default exception handler) will still handle it. Because the standard handling for *Exception* displays the name of the exception raised, you can see that it is your new exception that is raised.

Raising an exception

To indicate a disruptive error condition in an application, you can raise an exception that involves constructing an instance of that type and calling the reserved word **raise**.

To raise an exception, call the reserved word **raise**, followed by an instance of an exception object. This allows you to establish an exception as coming from a particular address. When an exception handler actually handles the exception, it finishes by destroying the exception instance, so you never need to do that yourself.

Raising an exception sets the *ErrorAddr* variable in the System unit to the address where the application raised the exception. You can refer to *ErrorAddr* in your exception handlers, for example, to notify the user where the error occurred. You can also specify a value in the **raise** clause which will appear in *ErrorAddr* when an exception occurs.

Warning Do not assign a value to *ErrorAddr* yourself. It is intended as read only.

To specify an error address for an exception, add the reserved word **at** after the exception instance, followed by an address expression such as an identifier.

For example, given the following declaration,

```
type
  EPasswordInvalid = class(Exception);
```

you can raise a “password invalid” exception at any time by calling **raise** with an instance of *EPasswordInvalid*, like this:

```
if Password <> CorrectPassword then
  raise EPasswordInvalid.Create('Incorrect password entered') at SomeProcedure;
```

Using interfaces

Kylix’s **interface** keyword allows you to create and use interfaces in your application. Interfaces are a way extending the single-inheritance model of CLX by allowing a single class to implement more than one interface, and by allowing several classes descended from different bases to share the same interface. Interfaces are useful when the same sets of operations, such as streaming, are used across a broad range of objects.

Interfaces as a language feature

An interface is like a class that contains only abstract methods and a clear definition of their functionality. Strictly speaking, interface method definitions include the number and types of their parameters, their return type, and their expected behavior. Interface methods are usually named to indicate the purpose of the interface. It is the convention to name interfaces according to their behavior and to preface them with a capital *I*. For example, an *IMalloc* interface would allocate, free, and manage memory. Similarly, an *IPersist* interface could be used as a general base interface for descendants, each of which defines specific method prototypes for loading and saving the state of an object to a storage, stream, or file.

An interface has the following syntax:

```
IMyObject = interface
  Procedure MyProcedure;
end;
```

A simple example of declaring an interface is:

```
type
IEdit = interface
```

```
    procedure Copy; stdcall;
    procedure Cut; stdcall;
    procedure Paste; stdcall;
    function Undo: Boolean; stdcall;
  end;
```

Like abstract classes, interfaces themselves can never be instantiated. To use an interface, you need to obtain it from an implementing class.

To implement an interface, you must define a class that declares the interface in its ancestor list, indicating that it will implement all of the methods of that interface:

```
TEditor = class(TInterfacedObject, IEdit)
  procedure Copy; stdcall;
  procedure Cut; stdcall;
  procedure Paste; stdcall;
  function Undo: Boolean; stdcall;
end;
```

While interfaces define the behavior and signature of their methods, they do not define the implementations. As long as the class's implementation conforms to the interface definition, the interface is fully polymorphic, meaning that accessing and using the interface is the same for any implementation of it.

Implementing interfaces across the hierarchy

Using interfaces offers a design approach to separating the way a class is used from the way it is implemented. Two classes can implement the same interface without requiring that they descend from the same base class. This polymorphic invocation of the same methods on unrelated objects is possible as long as the objects implement the same interface. For example, consider the interface,

```
IPaint = interface
  procedure Paint;
end;
```

and the two classes,

```
TSquare = class(TPolygonObject, IPaint)
  procedure Paint;
end;

TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Whether or not the two classes share a common ancestor, they are still assignment compatible with a variable of *IPaint* as in

```
var
  Painter: IPaint;
begin
  Painter := TSquare.Create;
  Painter.Paint;
  Painter := TCircle.Create;
  Painter.Paint;
end;
```

This could have been accomplished by having *TCircle* and *TSquare* descend from say, *TFigure* which implemented a virtual method *Paint*. Both *TCircle* and *TSquare* would then have overridden the *Paint* method. The above *IPaint* would be replaced by *TFigure*. However, consider the following interface:

```
IRotate = interface
  procedure Rotate(Degrees: Integer);
end;
```

which makes sense for the rectangle to support but not the circle. The classes would look like

```
TSquare = class(TRectangularObject, IPaint, IRotate)
  procedure Paint;
  procedure Rotate(Degrees: Integer);
end;

TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Later, you could create a class *TFilledCircle* that implements the *IRotate* interface to allow rotation of a pattern used to fill the circle without having to add rotation to the simple circle.

Note For these examples, the immediate base class or an ancestor class is assumed to have implemented the methods of *IInterface* that manage reference counting. For more information, see “Implementing IInterface” on page 4-20 and “Memory management of interface objects” on page 4-23.

Using interfaces with procedures

Interfaces also allow you to write generic procedures that can handle objects without requiring the objects to descend from a particular base class. Using the above *IPaint* and *IRotate* interfaces you can write the following procedures,

```
procedure PaintObjects(Painters: array of IPaint);
var
  I: Integer;
begin
  for I := Low(Painters) to High(Painters) do
    Painters[I].Paint;
end;

procedure RotateObjects(Degrees: Integer; Rotaters: array of IRotate);
var
  I: Integer;
begin
  for I := Low(Rotaters) to High(Rotaters) do
    Rotaters[I].Rotate(Degrees);
end;
```

RotateObjects does not require that the objects know how to paint themselves and *PaintObjects* does not require the objects know how to rotate.   This allows the above generic procedures to be used more often than if they were written to only work against a *TFigure* class.

For details about the syntax, language definitions and rules for interfaces, see the *Object Pascal Language Guide* online Help section on Object interfaces.

Implementing IInterface

All interfaces derive either directly or indirectly from the *IInterface* interface. This interface provides the essential functionality of an interface, that is, dynamic querying and lifetime management. This functionality is established in the three *IInterface* methods:

- *QueryInterface* provides a method for dynamically querying a given object and obtaining interface references for the interfaces the object supports.
- *_AddRef* is a reference counting method that increments the count each time the call to *QueryInterface* succeeds. While the reference count is nonzero the object must remain in memory.
- *_Release* is used with *_AddRef* to enable an object to track its own lifetime and to determine when it is safe to delete itself. Once the reference count reaches zero, the object is freed from memory.

Every class that implements interfaces must implement the three *IInterface* methods, as well as all of the methods declared by any other ancestor interfaces, and all of the methods declared by the interface itself. You can, however, inherit the implementations of methods of interfaces declared in your class.

By implementing these methods yourself, you can provide an alternative means of life-time management, disabling reference-counting. This is a powerful technique that lets you decouple interfaces from reference-counting.

TInterfacedObject

CLX defines a simple class, *TInterfacedObject*, that serves as a convenient base because it implements the methods of *IInterface*. *TInterfacedObject* class is declared in the *System* unit as follows:

```
type
  TInterfacedObject = class(TObject, IInterface)
  protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
  end;
```

Deriving directly from *TInterfacedObject* is straightforward. In the following example declaration, *TDerived* is a direct descendant of *TInterfacedObject* and implements a hypothetical *IPaint* interface.

```
type
  TDerived = class(TInterfacedObject, IPaint)
    ...
  end;
```

Because it implements the methods of *IInterface*, *TInterfacedObject* automatically handles reference counting and memory management of interfaced objects. For more information, see "Memory management of interface objects" on page 4-23, which also discusses writing your own classes that implement interfaces but that do not follow the reference-counting mechanism inherent in *TInterfacedObject*.

Using the as operator

Classes that implement interfaces can use the **as** operator for dynamic binding on the interface. In the following example:

```
procedure PaintObjects(P: TInterfacedObject)
var
  X: IPaint;

begin
  X := P as IPaint;
{ statements }
end;
```

the variable *P* of type *TInterfacedObject*, can be assigned to the variable *X*, which is an *IPaint* interface reference. Dynamic binding makes this assignment possible. For this assignment, the compiler generates code to call the *QueryInterface* method of *P*'s *IInterface* interface. This is because the compiler cannot tell from *P*'s declared type whether *P*'s instance actually supports *IPaint*. At runtime, *P* either resolves to an *IPaint* reference or an exception is raised. In either case, assigning *P* to *X* will not generate a compile-time error as it would if *P* was a class type that did not implement *IInterface*.

When you use the **as** operator for dynamic binding on an interface, you should be aware of the following requirements:

- Explicitly declaring *IInterface*: Although all interfaces derive from *IInterface*, it is not sufficient, if you want to use the **as** operator, for a class to simply implement the methods of *IInterface*. This is true even if it also implements the interfaces it explicitly declares. The class must explicitly declare *IInterface* in its interface list.
- Using an IID: Interfaces can use an identifier that is based on a GUID (globally unique identifier). GUIDs that are used to identify interfaces are referred to as interface identifiers (IIDs). If you are using the **as** operator with an interface, it must have an associated IID. To create a new GUID in your source code you can use the *Ctrl+Shift+G* editor shortcut key.

Reusing code and delegation

One approach to reusing code with interfaces is to have an object contain, or be contained by another. CLX uses properties that are object types as an approach to containment and code reuse. To support this design for interfaces, Kylix has a keyword—**implements**—that makes if easy to write code to delegate all or part of the implementation of an interface to a subobject.

Using implements for delegation

Many classes in CLX have properties that are subobjects. You can also use interfaces as property types. When a property is of an interface type (or a class type that implements the methods of an interface), you can use the keyword **implements** to specify that the methods of that interface are delegated to the object or interface reference which is the property instance. The delegate only needs to provide implementation for the methods. It does not have to declare the interface support. The class containing the property must include the interface in its ancestor list.

By default using the keyword **implements** delegates all interface methods. However, you can use methods resolution clauses or declare methods in your class that implement some of the interface methods to override this default behavior.

The following example uses the implements keyword in the design of a color adapter object that converts an 8-bit RGB color value to a *Color* reference:

```
unit cadapt;

type
IRGB8bit = interface
    ['{1d76360a-f4f5-11d1-87d4-00c04fb17199}']
    function Red: Byte;
    function Green: Byte;
    function Blue: Byte;
  end;

IColorRef = interface
    ['{1d76360b-f4f5-11d1-87d4-00c04fb17199}']
    function Color: Integer;
  end;

{ TRGB8ColorRefAdapter   map an IRGB8bit to an IColorRef }
  TRGB8ColorRefAdapter = class(TInterfacedObject, IRGB8bit, IColorRef)
  private
    FRGB8bit: IRGB8bit;
    FPalRelative: Boolean;
  public
    constructor Create(rgb: IRGB8bit);
    property RGB8Intf: IRGB8bit read FRGB8bit implements IRGB8bit;
    property PalRelative: Boolean read FPalRelative write FPalRelative;
    function Color: Integer;
  end;

implementation

constructor TRGB8ColorRefAdapter.Create(rgb: IRGB8bit);
begin
```

```
  FRGB8bit := rgb;
end;

function TRGB8ColorRefAdapter.Color: Integer;
begin
  if FPalRelative then
    Result := PaletteRGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue)
  else
    Result := RGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue);
end;
end.
```

For more information about the syntax, implementation details, and language rules of the **implements** keyword, see the *Object Pascal Language Guide* online Help section on object interfaces.

Memory management of interface objects

One of the concepts behind the design of interfaces is ensuring the lifetime management of the objects that implement them. The *_AddRef* and *_Release* methods of *IInterface* provide a way of implementing this functionality. Their defined behavior states that they will track the lifetime of an object by incrementing the reference count on the object when an interface reference is passed to a client, and will destroy the object when that reference count is zero.

Using reference counting

Kylix provides much of the *IInterface* memory management through its implementation of interface querying and reference counting. Therefore, if you have an object that lives and dies by its interfaces, you can easily use reference counting by deriving from these classes. *TInterfacedObject* is the class that provides this behavior. If you decide to use reference counting, you must be careful to only hold the object as an interface reference and to be consistent in your reference counting. For example:

```
procedure beep(x: ITest);

function test_func()
var
  y: ITest;
begin
  y := TTest.Create; // because y is of type ITest, the reference count is one
  beep(y); // the act of calling the beep function increments the reference count
  // and then decrements it when it returns
  y.something; // object is still here with a reference count of one
end;
```

This is the cleanest and safest approach to memory management; and if you use *TInterfacedObject* it is handled automatically. If you do not follow this rule, your object can unexpectedly disappear, as demonstrated in the following code:

```
function test_func()
var
  x: TTest;
begin
```

```
    x := TTest.Create; // no count on the object yet
    beep(x as ITest); // count is incremented by the act of calling beep
    // and decremented when it returns
    x.something; // surprise, the object is gone
  end;
```

Note In the examples above, the *beep* procedure, as it is declared, increments the reference count (call *AddRef*) on the parameter, whereas the following declarations do not:

```
procedure beep(const x: ITest);
```

or

```
procedure beep(var x: ITest);
```

These declarations generate smaller, faster code.

One case where you cannot use reference counting, because it cannot be consistently applied, is if your object is a component or a control owned by another component. In that case, you can still use interfaces, but you should not use reference counting because the lifetime of the object is not dictated by its interfaces.

Not using reference counting

If your object is a CLX component or a control that is owned by another component, then your object is part of a different memory management system that is based in *TComponent*. You should not mix the object lifetime management approaches of CLX components and interface reference counting. If you want to create an object that supports interfaces, you can implement the *IInterface _AddRef* and *_Release* methods as empty functions to bypass the interface reference counting mechanism:

```
function TMyObject._AddRef: Integer;
begin
  Result := -1;
end;

function TMyObject._Release: Integer;
begin
  Result := -1;
end;
```

You would still implement *QueryInterface* as usual to provide dynamic querying on your object. Because you do implement *QueryInterface*, you can still use the **as** operator for interfaces on components, as long as you create an interface identifier (IID).

Note that *TComponent* already implements these methods in this way so you do not need to do this if your object is based on *TComponent* or any of its descendants.

Working with strings

Kylix has a number of different character and string types that have been introduced throughout the development of the Object Pascal language. This section is an overview of these types, their purpose, and usage. For language details, see the Object Pascal Language online Help on String types.

Character types

Kylix has three character types: *Char*, *AnsiChar*, and *WideChar*.

The *Char* character type came from standard Pascal, was used in Turbo Pascal, and then in Object Pascal. Later Object Pascal added *AnsiChar* and *WideChar* as specific character types that support standards for character representation. *AnsiChar* supports 8-bit characters, and *WideChar* supports a 16-bit Unicode standard. The name *WideChar* is used because Unicode characters are also known as wide characters. Wide characters are two bytes instead of one, so that the character set can represent many more different characters. When *AnsiChar* and *WideChar* were implemented, *Char* became the default character type representing the currently recommended implementation.

Note The Linux wchar_t widechar is 32 bits per character. The 16-bit Unicode standard that Object Pascal widechars support is a subset of the 32-bit UCS standard supported by Linux and the GNU libraries. Pascal widechar data must be widened to 32 bits per character before it can be passed to an OS function as wchar_t.

The following table summarizes these character types:

Table 4.2 Object Pascal character types

Type	Bytes	Contents	Purpose
Char	1	A single character	Default character type
AnsiChar	1	A single character	8-bit characters
WideChar	2	A single Unicode character	16-bit Unicode standard

For more information about using these character types, see the *Object Pascal Language Guide* online Help on Character types. For more information about Unicode characters, see the *Object Pascal Language Guide* online Help on About extended character sets.

String types

Kylix has three categories of string types:

- Character pointers
- String types
- String classes

This section summarizes string types, and discusses using them with character pointers. For information about using string classes, see the online Help on *TStrings*.

Kylix has three string implementations: short strings, long strings, and wide strings. Several different string types represent these implementations. In addition, the reserved word **string** defaults to the currently recommended string implementation.

Short strings

String was the first string type used in Turbo Pascal. **String** was originally implemented as a short string. Short strings are an allocation of between 1 and 256 bytes, of which the first byte contains the length of the string and the remaining bytes contain the characters in the string:

```
S: string[0..n]// the original string type
```

When long strings were implemented, **string** was changed to a long string implementation by default and *ShortString* was introduced as a backward compatibility type. *ShortString* is a predefined type for a maximum length string:

```
S: string[255]// the ShortString type
```

The size of the memory allocated for a *ShortString* is static, meaning that it is determined at compile time. However, the location of the memory for the *ShortString* can be dynamically allocated, for example if you use a *PShortString*, which is a pointer to a *ShortString*. The number of bytes of storage occupied by a short string type variable is the maximum length of the short string type plus one. For the *ShortString* predefined type the size is 256 bytes.

Both short strings, declared using the syntax **string**[0..n], and the *ShortString* predefined type exist primarily for backward compatibility with earlier versions of Kylix and Borland Pascal.

A compiler directive, $H, controls whether the reserved word **string** represents a short string or a long string. In the default state, {$H+}, **string** represents a long string. You can change it to a *ShortString* by using the {$H-} directive. The {$H-} state is mostly useful for using code from versions of Object Pascal that used short strings by default. However, short strings can be useful in data structures where you need a fixed-size component. You can locally override the meaning of string-type definitions to ensure generation of short strings. You can also change declarations of short string types to string[255] or *ShortString*, which are unambiguous and independent of the $H setting.

For details about short strings and the *ShortString* type, see the *Object Pascal Language Guide* online Help on Short strings.

Long strings

Long strings are dynamically allocated strings with a maximum length of 2 Gigabytes. Like short strings, long strings use 8-bit characters and have a length indicator. Unlike short strings, long strings have no zeroth element that contains the dynamic string length. To find the length of a long string you must use the *Length* standard function, and to set the length of a long string you must use the *SetLength* standard procedure. Long strings are also reference-counted and, like *PChars*, long strings are null-terminated. For details about the implementation of longs strings, see the *Object Pascal Language Guide* online Help on Long strings.

Long strings are denoted by the reserved word **string** and by the predefined identifier *AnsiString*. For new applications, it is recommended that you use the long string type. All components in the Visual Component Library are compiled in this state, typically using **string**. If you write components, they should also use long strings, as should any code that receives data from string-type properties. If you want to write specific code that always uses a long string, then you should use *AnsiString*. If you want to write flexible code that allows you to easily change the type as new string implementations become standard, then you should use **string**.

WideString

The *WideChar* type allows wide character strings to be represented as arrays of *WideChars*. Wide strings are strings composed of 16-bit Unicode characters. As with long strings, WideStrings are dynamically allocated with a maximum length of 2 Gigabytes and they are reference counted. The *WideString* type is denoted by the predefined identifier *WideString*.

For more information about WideStrings, see the *Object Pascal Language Guide* online Help on WideString.

PChar types

A *PChar* is a pointer to a null-terminated string of characters of the type *Char*. Each of the three character types also has a built-in pointer type:

- A *PChar* is a pointer to a null-terminated string of 8-bit characters.
- A *PAnsiChar* is a pointer to a null-terminated string of 8-bit characters.
- A *PWideChar* is a pointer to a null-terminated string of 16-bit characters.

PChars are, with short strings, one of the original Object Pascal string types.

OpenString

An *OpenString* is obsolete, but you may see it in older code. It is for 16-bit compatibility and is allowed only in parameters. *OpenString* was used, before long strings were implemented, to allow a short string of an unspecified length string to be passed as a parameter. For example, this declaration:

```
procedure a(v : openstring);
```

will allow any length string to be passed as a parameter, where normally the string length of the formal and actual parameters must match exactly. You should not have to use *OpenString* in any new applications you write.

Refer also to the {$P+/-} switch in “Compiler directives for strings” on page 4-34.

Runtime library string handling routines

The runtime library provides many specialized string handling routines specific to a string type. These are routines for wide strings, longs strings, and null-terminated strings (meaning *PChars*). Routines that deal with *PChar* types use the null-termination to determine the length of the string. For more details about null-terminated strings, see Working with null-terminated strings in the *Object Pascal Language Guide* online Help.

The runtime library also includes a category of string formatting routines. There are no categories of routines listed for *ShortString* types. However, some built-in compiler routines deal with the *ShortString* type. These include, for example, the *Low* and *High* standard functions.

Because wide strings and long strings are the commonly used types, the remaining sections discuss these routines.

Wide character routines

When working with strings you should make sure that the code in your application can handle the strings it will encounter in the various target locales. Sometimes you will need to use wide characters and wide strings. In fact, one approach to working with ideographic character sets is to convert all characters to a wide character encoding scheme such as Unicode. CLX components represent all string values as wide strings.

Using a wide character encoding scheme has the advantage that you can make many of the usual assumptions about strings that do not work for MBCS systems. There is a direct relationship between the number of bytes in the string and the number of characters in the string. You do not need to worry about cutting characters in half or mistaking the second half of a character for the start of a different character.

Commonly used long string routines

The long string handling routines cover several functional areas. Within these areas, some are used for the same purpose, the differences being whether or not they use a particular criteria in their calculations. The tables included later list these routines by these functional areas:

- Comparison
- Case conversion
- Modification
- Sub-string
- String handling

Where appropriate, the tables also provide columns indicating whether or not a routine satisfies the following criteria.

- Uses case sensitivity: If the locale settings are used, it determines the definition of case. If the routine does not use the locale settings, analyses are based upon the ordinal values of the characters. If the routine is case-insensitive, there is a logical merging of upper- and lowercase characters that is determined by a predefined pattern.
- Uses the locale settings: Locale settings allow you to customize your application for specific locales. In particular, for Asian language environments. Most locale settings consider lowercase characters to be less than the corresponding uppercase characters. This is in contrast to ASCII order, in which lowercase characters are greater than uppercase characters.
- Supports the multi-byte character set (MBCS): MBCSs are used when writing code for far eastern locales. Multi-byte characters are represented as a mix of one- to six-byte character codes, so the length in bytes does not necessarily correspond to the length of the string. The routines that support MBCS are written to parse characters. The *ByteType* and *StrByteType* determine whether a particular byte is

the lead byte of a two or more byte character. Be careful when using multi-byte characters not to truncate a string by cutting a multibyte character in half. Do not pass characters as a parameter to a function or procedure, since the size of a character cannot be predetermined. Pass, instead, a pointer to a to a character or string. For more information about MBCS, see "Enabling application code" on page 12-2 of Chapter 12, "Creating international applications."

Note Because of limitations in recent versions of glibc, avoid passing large strings (greater than 50K) to multibyte-enabled routines. This limitation should be removed in future versions of glibc.

Table 4.3 String comparison routines

Routine	Case-sensitive	Uses locale settings	Supports MBCS
AnsiCompareStr	yes	yes	yes
AnsiCompareText	no	yes	yes
AnsiCompareFileName	yes	yes	yes
CompareStr	yes	no	no
CompareText	no	no	no

Table 4.4 Case conversion routines

Routine	Uses locale settings	Supports MBCS
AnsiLowerCase	yes	yes
AnsiLowerCaseFileName	yes	yes
AnsiUpperCaseFileName	yes	yes
AnsiUpperCase	yes	yes
LowerCase	no	no
UpperCase	no	no

The routines used for string file names: *AnsiCompareFileName*, *AnsiLowerCaseFileName*, and *AnsiUpperCaseFileName* all use the locale settings. You should always use file names that are portable because the locale (character set) used for file names can differ from the default user interface. Most portable file names are those which are simple ASCII and contain no special characters. If application generated, file names should be made all lowercase. If user input, file names should be made exactly as the user typed them with regard to case.

Table 4.5 String modification routines

Routine	Case-sensitive	Supports MBCS
AdjustLineBreaks	NA	yes
AnsiQuotedStr	NA	yes
StringReplace	optional by flag	yes
Trim	NA	yes

Table 4.5 String modification routines (continued)

Routine	Case-sensitive	Supports MBCS
TrimLeft	NA	yes
TrimRight	NA	yes
WrapText	NA	yes

Table 4.6 Substring routines

Routine	Case-sensitive	Supports MBCS
AnsiExtractQuotedStr	NA	yes
AnsiPos	yes	yes
IsDelimiter	yes	yes
IsPathDelimiter	yes	yes
LastDelimiter	yes	yes
QuotedStr	no	no

Table 4.7 String handling routines

Routine	Case-sensitive	Supports MBCS
AnsiContainsText	no	yes
AnsiEndsText	no	no
AnsiIndexText	no	yes
AnsiMatchText	no	yes
AnsiResemblesText	no	no
AnsiStartsText	no	yes
IfThen	NA	yes
LeftStr	yes	no
RightStr	yes	no
SoundEx	NA	no
SoundExInt	NA	no
DecodeSoundExInt	NA	no
SoundExWord	NA	no
DecodeSoundExWord	NA	no
SoundExSimilar	NA	no
SoundExCompare	NA	no

Declaring and initializing strings

When you declare a long string:

```
S: string;
```

you do not need to initialize it. Long strings are automatically initialized to empty. To test a string for empty you can either use the *EmptyStr* variable:

```
S = EmptyStr;
```

or test against an empty string:

```
S = '';
```

An empty string has no valid data. Therefore, trying to index an empty string is like trying to access **nil** and will result in an access violation:

```
var
  S: string;
begin
  S[i];    // this will cause an access violation
  // statements
end;
```

Similarly, if you cast an empty string to a *PChar*, the result is a **nil** pointer. So, if you are passing such a *PChar* to a routine that needs to read or write to it, be sure that the routine can handle **nil**:

```
var
  S: string;   // empty string
begin
  proc(PChar(S));  // be sure that proc can handle nil
  // statements
end;
```

If it cannot, then you can either initialize the string:

```
S := 'No longer nil';
proc(PChar(S));// proc does not need to handle nil now
```

or set the length, using the *SetLength* procedure:

```
SetLength(S, 100);//sets the dynamic length of S to 100
proc(PChar(S));// proc does not need to handle nil now
```

When you use *SetLength*, existing characters in the string are preserved, but the contents of any newly allocated space is undefined. Following a call to *SetLength*, *S* is guaranteed to reference a unique string, that is a string with a reference count of one. To obtain the length of a string, use the *Length* function.

Remember when declaring a **string** that:

```
S: string[n];
```

implicitly declares a short string, not a long string of *n* length. To declare a long string of specifically *n* length, declare a variable of type **string** and use the *SetLength* procedure.

```
S: string;
SetLength(S, n);
```

Mixing and converting string types

Short, long, and wide strings can be mixed in assignments and expressions, and the compiler automatically generates code to perform the necessary string type conversions. However, when assigning a string value to a short string variable, be aware that the string value is truncated if it is longer than the declared maximum length of the short string variable.

Long strings are already dynamically allocated. If you use one of the built-in pointer types, such as *PAnsiString*, *PString*, or *PWideString*, remember that you are introducing another level of indirection. Be sure this is what you intend.

Additional functions (*CopyQStringListToTstrings, Copy TStringsToQStringList, QStringListToTStringList*) are provided for converting underlying Qt string types and Kylix string types. These functions are located in Qtypes.pas.

String to PChar conversions

Long string to *PChar* conversions are not automatic. Some of the differences between strings and *PChars* can make conversions problematic:

- Long strings are reference-counted, while *PChars* are not.
- Assigning to a string copies the data, while a *PChar* is a pointer to memory.
- Long strings are null-terminated and also contain the length of the string, while *PChars* are simply null-terminated.

Situations in which these differences can cause subtle errors are discussed in this section.

String dependencies

Sometimes you will need convert a long string to a null-terminated string, for example, if you are using a function that takes a *PChar*. If you must cast a string to a *PChar*, be aware that you are responsible for the lifetime of the resulting *PChar*. Because long strings are reference counted, typecasting a string to a *PChar* increases the dependency on the string by one, without actually incrementing the reference count. When the reference count hits zero, the string will be destroyed, even though there is an extra dependency on it. The cast *PChar* will also disappear, while the routine you passed it to may still be using it. For example:

```
procedure my_func(x: string);
begin
  // do something with x
  some_proc(PChar(x)); // cast the string to a PChar
  // you now need to guarantee that the string remains
  // as long as the some_proc procedure needs to use it
end;
```

Returning a PChar local variable

A common error when working with *PChars* is to store in a data structure, or return as a value, a local variable. When your routine ends, the *PChar* will disappear because it is simply a pointer to memory, and is not a reference counted copy of the string. For example:

```
function title(n: Integer): PChar;
var
  s: string;
begin
  s := Format('title - %d', [n]);
  Result := PChar(s); // DON'T DO THIS
end;
```

This example returns a pointer to string data that is freed when the *title* function returns.

Passing a local variable as a PChar

Consider that you have a local string variable that you need to initialize by calling a function that takes a *PChar*. One approach is to create a local **array of char** and pass it to the function, then assign that variable to the string:

```
// assume FillBuffer is a predefined function
function FillBuffer(Buf:PChar;Count:Integer):Integer
begin
  . . .
end;

// assume MAXSIZE is a predefined constant
var
  i: Integer;
  buf: array[0..MAX_SIZE] of char;
  S: string;
begin
  i := FillBuffer(0, @buf, SizeOf(buf));// treats @buf as a PChar
  S := buf;
  //statements
end;
```

This approach is useful if the size of the buffer is relatively small, since it is allocated on the stack. It is also safe, since the conversion between an **array of char** and a **string** is automatic. When *FillBuffer* returns, the *Length* of the string correctly indicates the number of bytes written to *buf*.

To eliminate the overhead of copying the buffer, you can cast the string to a *PChar* (if you are certain that the routine does not need the *PChar* to remain in memory). However, synchronizing the length of the string does not happen automatically, as it does when you assign an **array of char** to a **string**. You should reset the string *Length* so that it reflects the actual width of the string. If you are using a function that returns the number of bytes copied, you can do this safely with one line of code:

```
var
  S: string;
begin
```

```
    SetLength(S, MAX_SIZE);// when casting to a PChar, be sure the string is not empty
    SetLength(S, GetModuleFilename( 0, PChar(S), Length(S) ) );
    // statements
  end;
```

Compiler directives for strings

The following compiler directives affect character and string types.

Table 4.8 Compiler directives for strings

Directive	Description
{$H+/-}	A compiler directive, $H, controls whether the reserved word **string** represents a short string or a long string. In the default state, {$H+}, **string** represents a long string. You can change it to a *ShortString* by using the {$H-} directive.
{$P+/-}	The $P directive is meaningful only for code compiled in the {$H-} state, and is provided for backwards compatibility. $P controls the meaning of variable parameters declared using the string keyword in the {$H-} state. In the {$P-} state, variable parameters declared using the string keyword are normal variable parameters, but in the {$P+} state, they are open string parameters. Regardless of the setting of the $P directive, the OpenString identifier can always be used to declare open string parameters.

{$V+/-}	The $V directive controls type checking on short strings passed as variable parameters. In the {$V+} state, strict type checking is performed, requiring the formal and actual parameters to be of identical string types.
	In the {$V-} (relaxed) state, any short string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter. Be aware that this could lead to memory corruption. For example:
	`var S: string[3];`
	
	`procedure Test(var T: string);`
	`begin`
	`  T := ‘1234’;`
	`end;`
	
	`begin`
	`  Test(S);`
	`end.`

{$X+/-}	The {$X+} compiler directive enables Kylix's support for null-terminated strings by activating the special rules that apply to the built-in *PChar* type and zero-based character arrays. (These rules allow zero-based arrays and character pointers to be used with *Write*, *Writeln*, *Val*, *Assign*, and *Rename* from the System unit.)

Strings and characters: related topics

The following *Object Pascal Language Guide* topics discuss strings and character sets. Also see Chapter 12, “Creating international applications.”

- “About extended character sets” (Discusses international character sets.)
- “Working with null-terminated strings” (Contains information about character arrays.)

- "Character strings"
- "Character pointers"
- "String operators."

Working with files

This section describes working with files and distinguishes between manipulating files on disk, and input/output operations such as reading and writing to files. The first section discusses the runtime library and routines you would use for common programming tasks that involve manipulating files on disk. The next section is an overview of file types used with file I/O. The last section focuses on the recommended approach to working with file I/O, which is to use file streams.

Although the Object Pascal language is not case sensitive, the Linux operating system is. Be attentive to case when working with files on Linux.

Note Previous versions of the Object Pascal language performed operations on files themselves, rather than on the file name parameters commonly used now. With these file types, you had to locate a file and assign it to a file variable before you could, for example, rename the file.

Manipulating files

Several common file operations are built into Object Pascal's runtime library. The procedures and functions for working with files operate at a high level. For most routines, you specify the name of the file and the routine makes the necessary calls to the operating system for you. In some cases, you use file handles instead. A file handle is a number assigned by Kylix to identify something.

Object Pascal provides routines for most file manipulation. When it does not, alternative routines are discussed.

Caution Although the Object Pascal language is not case sensitive, the Linux operating system is. Be attentive to case when working with files on Linux.

Deleting a file

Deleting a file erases the file from the disk and removes the entry from the disk's directory. There is no corresponding operation to restore a deleted file, so applications should generally allow users to confirm deletions of files. To delete a file, pass the name of the file to the *DeleteFile* function:

```
DeleteFile(FileName);
```

DeleteFile returns *True* if it deleted the file and *False* if it did not (for example, if the file did not exist or if it was read-only). *DeleteFile* erases the file named by *FileName* from the disk.

Finding a file

There are three routines used for finding a file: *FindFirst*, *FindNext*, and *FindClose*. *FindFirst* searches for the first instance of a file name with a given set of attributes in a specified directory. *FindNext* returns the next entry matching the name and attributes specified in a previous call to *FindFirst*. *FindClose* releases memory allocated by *FindFirst*. You should always use *FindClose* to terminates a *FindFirst*/*FindNext* sequence. If you want to know if a file exists, a *FileExists* function returns *True* if the file exists, *False* otherwise.

The three file find routines take a *TSearchRec* as one of the parameters. *TSearchRec* defines the file information searched for by *FindFirst* or *FindNext*. The declaration for *TSearchRec* is:

```
type
  TFileName = string;
  TSearchRec = record
    Time: Integer;//Time contains the time stamp of the file.
    Size: Integer;//Size contains the size of the file in bytes.
    Attr: Integer;//Attr represents the file attributes of the file.
    Name: TFileName;//Name contains the file name and extension.
    ExcludeAttr: Integer;
    FindHandle: THandle;
    FindData: TWin32FindData;//FindData contains additional information such as
    //file creation time, last access time, long and short filenames.
  end;
```

If a file is found, the fields of the *TSearchRec* type parameter are modified to describe the found file. You can test *Attr* against the following attribute constants or values to determine if a file has a specific attribute:

Table 4.9 Attribute constants and values

Constant	Value	Description
faReadOnly	$00000001	Read-only files
faHidden	$00000002	Hidden files
faSysFile	$00000004	System files
faVolumeID	$00000008	Volume ID files
faDirectory	$00000010	Directory files
faArchive	$00000020	Archive files
faAnyFile	$0000003F	Any file

To test for an attribute, combine the value of the *Attr* field with the attribute constant with the **and** operator. If the file has that attribute, the result will be greater than 0. For example, if the found file is a hidden file, the following expression will evaluate to *True*: (*SearchRec.Attr* **and** *faHidden* > 0). Attributes can be combined by OR'ing their constants or values. For example, to search for read-only and hidden files in addition to normal files, pass (*faReadOnly* **or** *faHidden*) the *Attr* parameter.

Example: This example uses a label, a button named *Search*, and a button named *Again* on a form. When the user clicks the *Search* button, the first file in the specified path is found, and the name and the number of bytes in the file appear in the label's caption.

Each time the user clicks the *Again* button, the next matching file name and size is displayed in the label:

```
var
  SearchRec: TSearchRec;

procedure TForm1.SearchClick(Sender: TObject);
begin
  FindFirst('/usr/Kylix/bin/*.*', faAnyFile, SearchRec);
  Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in size';
end;

procedure TForm1.AgainClick(Sender: TObject);
begin
  if (FindNext(SearchRec) = 0)
    Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in size';
  else
    FindClose(SearchRec);
end;
```

Changing file attributes

Every file has various attributes stored by the operating system as bitmapped flags. File attributes include such items as whether a file is read-only or a hidden file. Changing a file's attributes requires three steps: reading, changing, and setting.

You may be limited as to what you may modify depending on your file access rights.

Reading file attributes: Operating systems store file attributes in various ways, generally as bitmapped flags. To read a file's attributes, pass the file name to the *FileGetAttr* function, which returns the file attributes of a file. The return value is a group of bitmapped file attributes, of type *Word*. The attributes can be examined by AND-ing the attributes with the constants defined in *TSearchRec*. A return value of -1 indicates that an error occurred.

Changing individual file attributes: Because Kylix represents file attributes in a bit flag, you can use normal logical operators to manipulate the individual attributes. Each attribute has a mnemonic name defined in the SysUtils unit. For example, to set a file's read-only attribute, you would do the following:

```
Attributes := Attributes or faReadOnly;
```

You can also set or clear several attributes at once. For example, the clear both the system-file and hidden attributes:

```
Attributes := Attributes and not (faSysFile or faHidden);
```

Setting file attributes: Kylix enables you to set the attributes for any file at any time. To set a file's attributes, pass the name of the file and the attributes you want to the *FileSetAttr* function. *FileSetAttr* sets the file attributes of a specified file.

You can use the reading and setting operations independently, if you only want to determine a file's attributes, or if you want to set an attribute regardless of previous settings. To change attributes based on their previous settings, however, you need to read the existing attributes, modify them, and write the modified attributes.

Renaming a file

To change a file name, simply use the *RenameFile* function:

```
function RenameFile(const OldFileName, NewFileName: string): Boolean;
```

which changes a file name, identified by *OldFileName*, to the name specified by *NewFileName*. If the operation succeeds, *RenameFile* returns *True*. If it cannot rename the file, for example, if a file called *NewFileName* already exists, it returns *False*. For example:

```
if not RenameFile('OLDNAME.TXT','NEWNAME.TXT') then
  ErrorMsg('Error renaming file!');
```

You cannot rename (move) a file across drives using *RenameFile*. You would need to first copy the file and then delete the old one.

File date-time routines

The *FileAge*, *FileGetDate*, and *FileSetDate* routines operate on operating system date-time values. *FileAge* returns the date-and-time stamp of a file, or -1 if the file does not exist. *FileSetDate* sets the date-and-time stamp for a specified file, and returns zero on success or an error code on failure. *FileGetDate* returns a date-and-time stamp for the specified file or -1 if the handle is invalid.

As with most of the file manipulating routines, *FileAge* uses a string file name. *FileGetDate* and *FileSetDate*, however, take a *Handle* type as a parameter. To get access to a *Handle,* instantiate *TFileStream*, *FileOpen*, or call an OS routine such as _open() to create or open a file. Then use the *Handle* property. See “File types with file I/O” on page 4-38 for more information.

File types with file I/O

You can use three file types when working with file I/O: Pascal file types, file handles, and file stream objects. The following table summarizes these types.

Table 4.10 File types for file I/O

File type	Description
Pascal file types	In the System unit. These types are used with file variables, usually of the format "F: Text: or "F: File". The files have three types: typed, text, and untyped. A number of Kylix file-handling routines, such as *AssignPrn* and *writeln,* use them. If you need to work with them, see the *Object Pascal Language Guide*.
File handles	In the Sysutils unit. A number of routines use a handle to identify the file. You get the handle when you open or create the file (using FileOpen or FileCreate). Once you have the handle, there are routines to work with the contents of the file given its handle (write a line, read text, and so on).
File streams	File streams are object instances of the *TFileStream* class used to access information in disk files. File streams are a portable and high-level approach to file I/O. *TFileStream* has a *Handle* property that lets you access the file handle. The next section discusses *TFileStream*.

Using file streams

TFileStream is a class that enables applications to read from and write to a file on disk. It is used for high-level object representations of file streams. *TFileStream* offers multiple functionality: persistence, interaction with other streams, and file I/O.

- *TFileStream* is a descendant of the stream classes. As such, one advantage of using file streams is that they inherit the ability to persistently store component properties. The stream classes work with the *TFiler* classes, *TReader*, and *TWriter* to stream objects out to disk. Therefore, when you have a file stream, you can use that same code for CLX streaming mechanism. For more information about using the streaming system, see the *CLX Reference* online Help on the *TStream*, *TFiler*, *TReader*, *TWriter*, and *TComponent* classes.
- *TFileStream* can interact easily with other stream classes. For example, if you want to dump a dynamic memory block to disk, you can do so using a *TFileStream* and a *TMemoryStream*.
- *TFileStream* provides the basic methods and properties for file I/O. The remaining sections focus on this aspect of file streams

Creating and opening files

To create or open a file and get access to a handle for the file, you simply instantiate a *TFileStream*. This opens or creates a named file and provides methods to read from or write to it. If the file cannot be opened, *TFileStream* raises an exception.

```
constructor Create(const filename: string; Mode: Word);
```

The *Mode* parameter specifies how the file should be opened when creating the file stream. The *Mode* parameter consists of an open mode and a share mode or'ed together. The open mode must be one of the following values:

Table 4.11 Open modes

Value	Meaning
fmCreate	TFileStream a file with the given name. If a file with the given name exists, open the file in write mode.
fmOpenRead	Open the file for reading only.
fmOpenWrite	Open the file for writing only. Writing to the file completely replaces the current contents.
fmOpenReadWrite	Open the file to modify the current contents rather than replace them.

The share mode can be one of the following values with the restrictions listed below:

Table 4.12 Share modes

Value	Meaning
fmShareCompat	Other applications cannot open the file for any reason.
fmShareExclusive	Other applications cannot open the file for any reason.
fmShareDenyWrite	Other applications can open the file for reading but not for writing.

Table 4.12 Share modes (continued)

Value	Meaning
fmShareDenyRead	Other applications can open the file for writing but not for reading.
fmShareDenyNone	No attempt is made to prevent other applications from reading from or writing to the file.

Note that which share mode you can use depends on which open mode you used. The following table shows shared modes that are available for each open mode.

Table 4.13 Shared modes available for each open mode

Open Mode	fmShareCompat	fmShareExclusive	fmShareDenyWrite	fmShareDenyRead	fmShareDenyNone
fmOpenRead	Can't use	Can't use	Available	Can't use	Available
fmOpenWrite	Available	Available	Can't use	Available	Available
fmOpenReadWrite	Available	Available	Available	Available	Available

The file open and share mode constants are defined in the SysUtils unit.

Using the file handle

When you instantiate *TFileStream* you get access to the file handle. The file handle is contained in the *Handle* property. *Handle* is read-only and indicates the mode in which the file was opened. If you want to change the attributes of the file *Handle*, you must create a new file stream object.

Some file manipulation routines take a window's file handle as a parameter. Once you have a file stream, you can use the *Handle* property in any situation in which you would use a window's file handle. Be aware that, unlike handle streams, file streams close file handles when the object is destroyed.

Reading and writing to files

TFileStream has several different methods for reading from and writing to files. These are distinguished by whether they perform the following:

- Return the number of bytes read or written.
- Require the number of bytes is known.
- Raise an exception on error.

Read is a function that reads up to *Count* bytes from the file associated with the file stream, starting at the current *Position*, into *Buffer*. *Read* then advances the current position in the file by the number of bytes actually transferred. The prototype for *Read* is

```
function Read(var Buffer; Count: Longint): Longint; override;
```

Read is useful when the number of bytes in the file is not known. *Read* returns the number of bytes actually transferred, which may be less than *Count* if the end of file marker is encountered.

Write is a function that writes *Count* bytes from the *Buffer* to the file associated with the file stream, starting at the current *Position*. The prototype for *Write* is:

```
function Write(const Buffer; Count: Longint): Longint; override;
```

After writing to the file, *Write* advances the current position by the number bytes written, and returns the number of bytes actually written, which may be less than *Count* if the end of the buffer is encountered.

The counterpart procedures are *ReadBuffer* and *WriteBuffer* which, unlike *Read* and *Write*, do not return the number of bytes read or written. These procedures are useful in cases where the number of bytes is known and required, for example when reading in structures. *ReadBuffer* and *WriteBuffer* raise an exception on error (*EReadError* and *EWriteError*) while the *Read* and *Write* methods do not. The prototypes for *ReadBuffer* and *WriteBuffer* are:

```
procedure ReadBuffer(var Buffer; Count: Longint);
```

```
procedure WriteBuffer(const Buffer; Count: Longint);
```

These methods call the *Read* and *Write* methods, to perform the actual reading and writing.

Reading and writing strings

If you are passing a string to a read or write function, you need to use the correct syntax. The *Buffer* parameters for the read and write routines are **var** and **const** types, respectively. These are untyped parameters, so the routine takes the address of a variable.

The most commonly used type when working with strings is a long string. However, passing a long string as the *Buffer* parameter does not produce the correct result. Long strings contain a size, a reference count, and a pointer to the characters in the string. Consequently, dereferencing a long string does not result in only the pointer element. What you need to do is first cast the string to a *Pointer* or *PChar,* and then dereference it. For example:

```
procedure caststring;
var
  fs: TFileStream;
const
  s: string = 'Hello';
begin
  fs := TFileStream.Create('temp.txt', fmCreate or fmOpenWrite);
  fs.Write(s, Length(s));// this will give you garbage
  fs.Write(PChar(s)^, Length(s));// this is the correct way
end;
```

Seeking a file

Most typical file I/O mechanisms have a process of seeking a file in order to read from or write to a particular location within it. For this purpose, *TFileStream* has a *Seek* method. The prototype for *Seek* is:

```
function Seek(Offset: Longint; Origin: Word): Longint; override;
```

The *Origin* parameter indicates how to interpret the *Offset* parameter. *Origin* should be one of the following values:

Value	Meaning
soFromBeginning	Offset is from the beginning of the resource. Seek moves to the position Offset. Offset must be >= 0.
soFromCurrent	Offset is from the current position in the resource. Seek moves to Position + Offset.
soFromEnd	Offset is from the end of the resource. Offset must be <= 0 to indicate a number of bytes before the end of the file.

Seek resets the current *Position* of the stream, moving it by the indicated offset. *Seek* returns the new value of the *Position* property, the new current position in the resource.

File position and size

TFileStream has properties that hold the current position and size of the file. These are used by the *Seek*, read, and write methods.

The *Position* property of *TFileStream* is used to indicate the current offset, in bytes, into the stream (from the beginning of the streamed data). The declaration for *Position* is:

```
property Position: Longint;
```

The *Size* property indicates the size in bytes of the stream. It is used as an end of file marker to truncate the file. The declaration for *Size* is:

```
property Size: Longint;
```

Size is used internally by routines that read and write to and from the stream.

Setting the *Size* property changes the size of the file. If the *Size* of the file cannot be changed, an exception is raised. For example, trying to change the *Size* of a file opened in *fmOpenRead* mode raises an exception.

Copying

CopyFrom copies a specified number of bytes from one (file) stream to another.

```
function CopyFrom(Source: TStream; Count: Longint): Longint;
```

Using *CopyFrom* eliminates the need to create, read into, write from, and free a buffer when copying data.

CopyFrom copies *Count* bytes from *Source* into the stream. *CopyFrom* then moves the current position by *Count* bytes, and returns the number of bytes copied. If *Count* is 0, *CopyFrom* sets *Source* position to 0 before reading and then copies the entire contents of *Source* into the stream. If *Count* is greater than or less than 0, *CopyFrom* reads from the current position in *Source*.

Object Pascal data types

Object Pascal has many predefined data types. You can use these predefined types to create new types that meet the specific needs of your application. For an overview of types, see the *Object Pascal Language Guide*.

Chapter

5

Building applications and shared objects

This chapter provides an overview of how to use Kylix to create applications and shared objects.

Creating applications

The main use of Kylix is designing and building the following types of applications:

- GUI applications
- Console applications

GUI applications

When you compile a project, an executable file is created. The executable usually provides the basic functionality of your program, and simple programs often consist of only an executable file. You can extend the application by calling shared object files, packages, and other support files from the executable.

Kylix offers two application UI models:

- Single document interface (SDI)
- Multiple document interface (MDI)

In addition to the implementation model of your applications, the design-time behavior of your project and the runtime behavior of your application can be manipulated by setting project options in the IDE.

User interface models

Any form can be implemented as a multiple document interface (MDI) or single document interface (SDI) form. In an MDI application, more than one document or child window can be opened within a single parent window. This is common in applications such as spreadsheets or word processors. An SDI application, in contrast, normally contains a single document view. To make your form an SDI application, set the *FormStyle* property of your *Form* object to *fsNormal*.

For more information on developing the UI for an application, see Chapter 6, "Developing the application user interface."

SDI Applications

To create a new SDI application,

1 Select File | New to bring up the New Items dialog.
2 Click on the Projects page and select SDI Application.
3 Click OK.

By default, the *FormStyle* property of your *Form* object is set to *fsNormal*, so Kylix assumes that all new applications are SDI applications.

MDI applications

To create a new MDI application,

1 Select File | New to bring up the New Items dialog.
2 Click on the Projects page and select MDI Application.
3 Click OK.

MDI applications require more planning and are somewhat more complex to design than SDI applications. MDI applications spawn child windows that reside within the client window; the main form contains child forms. Set the *FormStyle* property of the *TForm* object to specify whether a form is a child (*fsMDIForm*) or main form (*fsMDIChild*). It is a good idea to define a base class for your child forms and derive each child form from this class, to avoid having to reset the child form's properties.

Setting IDE, project, and compilation options

Use Project | Project Options to specify various options for your project. For more information, see the online Help.

To change the default options that apply to all future projects, set the options in the Project Options dialog box and check the Default box at the bottom right of the window. All new projects will use the current options.

Console applications

Console applications can operate directly in a console window without requiring the window management layer. They are 32-bit programs that run without a graphical interface. These applications typically don't require much user input and perform a limited set of functions.

To create a new console application,

1 Choose File | New and select Console Application from the New Items dialog box.

Kylix then creates a project file for this type of source file and displays the code editor.

Note When you create a new console application, the IDE does not create a new form. Only the code editor is displayed.

Creating packages and shared object files

Shared object files are modules of compiled code that work with an executable to provide functionality to an application.

Packages are special shared object files used by Kylix applications, the IDE, or both. There are two kinds of packages: runtime packages and design-time packages. Runtime packages provide functionality to a program while that program is running. Design-time packages extend the functionality of the IDE.

For more information on packages, see Chapter 11, "Working with packages and components."

Working with shared object libraries

Shared object libraries on Linux are similar to Windows DLLs. You can link with third-party shared objects using external function declarations, just as you would with DLL functions under Windows.

The Linux program loader ignores module name bindings when resolving external function references. If an application uses two .so libraries that both export a function named Foo, the loader binds all Foo references to the first Foo function it finds. (The search order is described in the section on "Shared Object Dependencies" in the ELF standard.) When naming conflicts are unavoidable, you can prevent unintended behavior by loading objects dynamically with dlopen().

When you build a library project, the compiler generates a shared object (.so file) instead of a regular executable. By default, the name of the generated file starts with "lib" (for a standard library) or "bpl" (for a package). For example, if your project file is called something.pas, the compiler generates a shared object called libsomething.so or bplsomething.so.

The following compiler directives can be placed in library project files:

Table 5.1 Compiler directives for libraries

Compiler Directive	Description
{$SOPREFIX 'string'}	Overrides the default 'lib' or 'bpl' prefix in the output file name. For example, you could specify {$SOPREFIX 'dcl'} for a design-time package, or use {$SOPREFIX ' '} to eliminate the prefix entirely.
{$SOSUFFIX 'string'}	Adds a specified suffix to the output file name before the .so extension. For example, use {$SOSUFFIX '-2.1.3'} in something.pas to generate libsomething-2.1.3.so.
{$SOVERSION 'string'}	Adds a second extension to the output file name after the .so extension. For example, use {$SOVERSION '2.1.3'} in something.pas to generate libsomething.so.2.1.3.
{$SONAME 'string'}	Specifies the internally coded library name in the shared object's dynamic string table. This directive does not change the name of the output file itself, but does cause the compiler to create a symbolic link in the output directory that points to the actual file. For example, if you place {$SONAME 'libsomething.so.1'} in something.pas, the compiler creates a shared-object file called libsomething.so--which represents itself internally as libsomething.so.1--and a symbolic link, also called libsomething.so.1, that points to libsomething.so.

When to use packages and shared objects

For most applications written in Kylix, packages provide greater flexibility and are easier to create than shared objects. However, there are several situations where shared objects are better suited to your projects than packages:

- If your code module is called from non-Kylix applications.
- If extending the functionality of a Web server.
- If creating a code module for third-party developers.

Writing database applications

One of Kylix's strengths is its support for creating advanced database applications. Kylix includes built-in tools that allow you to connect to InterBase, MySQL, or other servers while providing transparent data sharing between applications.

Database applications are built from user interface elements, components that represent database information (datasets), and components that connect to the database information itself. Kylix supports two kinds of datasets:

- dbExpress
- Client

Different kinds of datasets connect to the underlying database information in different ways. *dbExpress* provides fast access to database information, supports cross-platform development, but does not include many data manipulation functions. Client datasets can buffer data in memory but you can't connect a client dataset directly to a database server, because client datasets do not include any built-in database access mechanism. Instead, you need to connect the client dataset to another dataset that can handle data access. For details, refer to “Database architecture” on page 14-4.

Part II, “Developing database applications” in this manual provides details on how to use Kylix to design database applications.

Building distributed applications

Distributed applications are applications that are deployed to various machines and platforms and work together, typically over a network, to perform a set of related functions. For instance, an application for purchasing items and tracking those purchases for a nationwide company would require individual client applications for all the outlets, a main server that would process the requests of those clients, and an interface to a database that stores all the information regarding those transactions. By building a distributed client application (for instance, a web-based application), maintaining and updating the individual clients is vastly simplified.

Kylix provides options for implementing distributed applications:

- TCP/IP applications
- Database applications

Distributing applications using TCP/IP

TCP/IP is a communication protocol that allows you to write applications that communicate over networks. You can implement virtually any design in your applications. TCP/IP provides a transport layer, but does not impose any particular architecture for creating your distributed application.

The growth of the Internet has created an environment where most computers already have some form of TCP/IP access, which simplifies distributing and setting up the application.

Applications that use TCP/IP can be message-based distributed applications (such as Web server applications that service HTTP request messages) or distributed object applications (such as distributed database applications that communicate using sockets).

The most basic method of adding TCP/IP functionality to your applications is to use client or server sockets. Kylix also provides support for applications that extend Web servers by creating CGI scripts. In addition, Kylix provides support for TCP/IP-based database applications.

Using sockets in applications

Two classes, *TTCPClient* and *TTCPServer*, allow you to create TCP/IP socket connections to communicate with other remote applications. For more information on sockets, see Chapter 23, “Working with sockets.”

Creating Web server applications

To create a new Web server application, select File | New and select Web Server Application in the New Items dialog box. Then select the Web server application type:

- Apache
- CGI stand-alone

Common Gateway Interface (CGI) is a standard for running programs on a server from a web page. CGI programs, or scripts, are executable programs that run by themselves. CGI applications use more system resources on the server, so complex applications are better created as Apache applications. The Apache Web server is a flexible, HTTP/1.1 compliant Web server that implements the latest protocols. It is configurable and is multi-platform.

For more information on building Web server applications, see Chapter 22, "Creating Internet server applications."

Apache Web server applications

Selecting this type of application sets up your project as an Apache Web server. Information is processed and returned to the client by the Web server.

TApacheApplication provides the underlying functionality for Apache server applications. *TApacheApplication* starts running when the Apache server receives an HTTP request. It creates objects to represent the request message (*TApacheRequest*) and any response (*TApacheResponse*) that should be sent in return, passes these to the dispatcher so that the response can be filled in, and sends the response (if not already sent).

CGI stand-alone Web server applications

CGI Web server applications are console applications that receive requests from clients on standard input, processes those requests, and sends back the results to the server on standard output to be sent to the client.

Using data modules and remote data modules

A *data module* is like a special form that contains nonvisual components. All the components in a data module *could* be placed on ordinary forms alongside visual controls. But if you plan on reusing groups of database and system objects, or if you want to isolate the parts of your application that handle database connectivity and business rules, then data modules provide a convenient organizational tool.

There are two types of data module, standard data modules, and Web modules:

- Standard data modules allow you to organize and document your application more easily. They are particularly useful for single- and two-tiered database applications, but can be used to organize the nonvisual components in any application. For more information, see "Creating data modules" on page 5-7.

- Web modules form the basis of Web server applications. In addition to holding the components that create the content of HTTP response messages, they handle the dispatching of HTTP messages from client applications. See Chapter 22, "Creating Internet server applications" for more information about using Web modules.

Creating data modules

To create a data module, choose File | New and double-click on Data Module. Kylix opens an empty data module, displays the unit file for the new module in the Code editor, and adds the module to the current project. When you reopen an existing data module, Kylix displays its components in the data module window.

At design time, you can add nonvisual components to a data module by selecting them on the Component palette and clicking in the data module window. When a component is selected in the data module window, you can edit its properties in the Object Inspector just as you would if the component were on a form. The properties you set for components in a data module apply consistently to all forms in your application that use that module.

The name of the data module is displayed in the title bar of the data module window. The default name is DataModule*n*, where *n* is a number representing the lowest unused unit number in a project. For example, if you start a new project, add a module to it before doing any other application building, the name of the data module defaults to DataModule2. The corresponding unit file for DataModule2 is Unit2 (Unit1 is the form). You can rename the data module by selecting the data module window and editing the *Name* property in the Object Inspector.

Creating business rules in a data module

In a data module's unit file, you can write methods, including event handlers for the components in the module, as well as global routines that encapsulate business rules. For example, you might write a procedure to perform month-, quarter-, or year-end bookkeeping; you could call such a procedure from an event handler for a component in the module or from any unit that uses the module.

Accessing a data module from a form

To associate visual controls on a form with a data module, you must first add the data module to the form's **uses** clause. You can do this in several ways:

- In the Code editor, open the form's unit file and add the name of the data module to the **uses** clause in the **interface** section.
- Choose File | Use Unit, then enter the name of the module or pick it from the list box in the Use Unit dialog.
- Double-click on a table or query component in the data module to open the Fields editor. From the Fields editor, drag any fields onto your form. Kylix prompts you to confirm that you want to add the module to the form's **uses** clause, then creates controls (such as edit boxes) for the fields.

Programming templates

Programming templates are commonly used “skeleton“ structures that you can add to your source code and then fill in. For example, if you want to use a **for** loop in your code, you could insert the following template:

```
for :=  to  do
begin

end;
```

To insert a code template in the Code editor, press *Ctrl-j* and select the template you want to use. You can also add your own templates to this collection. To add a template:

1 Select Tools | Editor Options.
2 Click the Code Insight tab.
3 In the templates section click Add.
4 Choose a shortcut name and enter a brief description of the new template.
5 Add the template code to the Code text box.
6 Click OK.

Sharing code: Using the Object Repository

The Object Repository (Tools | Repository) makes it easy share forms, dialog boxes, frames, and data modules. It also provides templates for new projects and wizards that guide the user through the creation of forms and projects. The repository is maintained in the delphi60dro file (by default in the .borland directory), a text file that contains references to the items that appear in the Repository and New Items dialogs.

Sharing items within a project

You can share items *within* a project without adding them to the Object Repository. When you open the New Items dialog box (File | New), you'll see a page tab with the name of the current project. This page lists all the forms, dialog boxes, and data modules in the project. You can derive a new item from an existing item and customize it as needed.

Adding items to the Object Repository

You can add your own projects, forms, frames, and data modules to those already available in the Object Repository. To add an item to the Object Repository,

1 If the item is a project or is in a project, open the project.

2 For a project, choose Project | Add To Repository. For a form or data module, right-click the item and choose Add To Repository.

3 Type a description, title, and author.

4 Decide which page you want the item to appear on in the New Items dialog box, then type the name of the page or select it from the Page combo box. If you type the name of a page that doesn't exist, Kylix creates a new page.

5 Choose Browse to select an icon to represent the object in the Object Repository.

6 Choose OK.

Sharing objects in a team environment

You can share objects with your workgroup or development team by making a repository available over a network. To use a shared repository, all team members must select the same Shared Repository directory in the Environment Options dialog:

1 Choose Tools | Environment Options.

2 On the Preferences page, locate the Shared Repository panel. In the Directory edit box, enter the directory where you want to locate the shared repository. Be sure to specify a directory that's accessible to all team members.

The first time an item is added to the repository, Kylix creates a delphi60dro file in the Shared Repository directory if one doesn't exist already.

Note It is important that the access permissions on the Object Repository directory (objrepos) are set up correctly on the because if the user does not have write permissions to the directory, they cannot add items to it. Therefore, if you want multiple users to access a common Object Repository, you should create a group and give its members read-write access to the objrepos directory. For example, if you name the group "dev", you would set the permissions on the command line by entering

```
cd <install directory>
chmod -R 775 objrepos
chgrp -R dev objrepos
```

See the *group(5)* man page for more information.

Using an Object Repository item in a project

To access items in the Object Repository, choose File | New. The New Items dialog appears, showing all the items available. Depending on the type of item you want to use, you have up to three options for adding the item to your project:

- Copy
- Inherit
- Use

Copying an item

Choose Copy to make an exact copy of the selected item and add the copy to your project. Future changes made to the item in the Object Repository will not be reflected in your copy, and alterations made to your copy will not affect the original Object Repository item.

Copy is the only option available for project templates.

Inheriting an item

Choose Inherit to derive a new class from the selected item in the Object Repository and add the new class to your project. When you recompile your project, any changes that have been made to the item in the Object Repository will be reflected in your derived class, in addition to changes you make to the item in your project. Changes made to your derived class do not affect the shared item in the Object Repository.

Inherit is available for forms, dialog boxes, and data modules, but not for project templates. It is the *only* option available for reusing items within the same project.

Using an item

Choose Use when you want the selected item itself to become part of your project. Changes made to the item in your project will appear in all other projects that have added the item with the Inherit or Use option. Select this option with caution.

The Use option is available for forms, dialog boxes, and data modules.

Using project templates

Templates are predesigned projects that you can use as starting points for your own work. To create a new project from a template,

1. Choose File|New to display the New Items dialog box.
2. Choose the Projects tab.
3. Select the project template you want and choose OK.
4. In the Select Directory dialog, specify a directory for the new project's files.

Kylix copies the template files to the specified directory, where you can modify them. The original project template is unaffected by your changes.

Modifying shared items

If you modify an item in the Object Repository, your changes will affect all future projects that use the item as well as existing projects that have added the item with the Use or Inherit option. To avoid propagating changes to other projects, you have several alternatives:

- Copy the item and modify it in your current project only.
- Copy the item to the current project, modify it, then add it to the Repository under a different name.
- Create a component, shared object, component template, or frame from the item. If you create a component or shared object, you can share it with other developers.

Specifying a default project, new form, and main form

By default, when you choose File | New Application or File | New Form, Kylix displays a blank form. You can change this behavior by reconfiguring the Repository:

1 Choose Tools | Repository
2 If you want to specify a default project, select the Projects page and choose an item under Objects. Then select the New Project check box.
3 If you want to specify a default form, select a Repository page (such as Forms), them choose a form under Objects. To specify the default new form (File | New Form), select the New Form check box. To specify the default main form for new projects, select the Main Form check box.
4 Click OK.

Reusing components and groups of components

Kylix offers several ways to save and reuse work you've done with CLX components:

- Component templates provide a simple, quick way of configuring and saving groups of components. See "Creating and using component templates" on page 6-35.
- You can save forms, data modules, and projects in the Repository. This gives you a central database of reusable elements and lets you use form inheritance to propagate changes. See "Sharing code: Using the Object Repository" on page 5-14.
- You can save frames on the Component palette or in the repository. Frames use form inheritance and can be embedded into forms or other frames. See "Working with frames" on page 6-36.
- Creating a custom component is the most complicated way of reusing code, but it offers the greatest flexibility. See Chapter 25, "Overview of component creation."

Creating and using component templates

You can create templates that are made up of one or more components. After arranging components on a form, setting their properties, and writing code for them, save them as a *component template*. Later, by selecting the template from the Component palette, you can place the preconfigured components on a form in a single step; all associated properties and event-handling code are added to your project at the same time.

Once you place a template on a form, you can reposition the components independently, reset their properties, and create or modify event handlers for them just as if you had placed each component in a separate operation.

To create a component template,

1 Place and arrange components on a form. In the Object Inspector, set their properties and events as desired.

2 Select the components. The easiest way to select several components is to drag the mouse over all of them. Gray handles appear at the corners of each selected component.

3 Choose Component | Create Component Template.

4 Specify a name for the component template in the Component Name edit box. The default proposal is the component type of the first component selected in step 2 followed by the word "Template". For example, if you select a label and then an edit box, the proposed name will be "TLabelTemplate". You can change this name, but be careful not to duplicate existing component names.

5 In the Palette Page edit box, specify the Component palette page where you want the template to reside. If you specify a page that does not exist, a new page is created when you save the template.

6 Under Palette Icon, select a bitmap to represent the template on the palette. The default proposal will be the bitmap used by the component type of the first component selected in step 2. To browse for other bitmaps, click Change. The bitmap you choose must be no larger than 24 pixels by 24 pixels.

7 Click OK.

To remove templates from the Component palette, choose Component | Configure Palette.

Enabling Help in CLX applications

While CLX does not provide direct support for displaying Help, it provides a backbone through which Help requests, triggered by the F1 key, can be passed on to one of multiple external Help viewers (that is, Man, Info, or HyperHelp). To support this, an application developer must create a class that implements the *ICustomHelpViewer* interface (and, optionally, one of several interfaces descended from it), and then register an instance of that class with the global Help Manager.

The Help Manager maintains a list of registered viewers and passes requests to them in a two-phase process: it first asks each viewer if it can provide support for a particular Help keyword or context, and then it passes the Help request on to the viewer which says it can provide such support. (If more than one viewer supports the keyword, as would be the case in an application which had registered viewers for both Man and Info, the Help Manager can display a selection box through which the user of the application can determine which Help viewer to invoke. Otherwise, it displays the first responding Help system encountered).

Help system interfaces

The CLX Help system allows communication between your application and Help viewers through a series of interfaces. These interfaces are all defined in HelpIntfs.pas, which also contains the implementation of the Help Manager.

ICustomHelpViewer provides support for displaying Help based upon a provided keyword and for displaying a table of contents listing all Help available in a particular viewer.

IExtendedHelpViewer provides support for displaying Help based upon a numeric Help context and for displaying topics; in most Help systems, topics function as high-level keywords (for example, “IntToStr” might be a keyword in the Kylix Help system, but “String manipulation routines” could be the name of a topic).

*ISpecialWinHelpViewe*r provides support for responding to specialized WinHelp messages that an application running under Windows may receive and which are not easily generalizable. In general, only applications operating in the Windows environment need to implement this interface, and even then it is only required for applications that make extensive use of non-standard WinHelp messages.

IHelpManager provides a mechanism for the Help viewer to communicate back to the application’s Help Manager and request additional information. An *IHelpManager* is obtained at the time the Help viewer registers itself.

IHelpSystem provides a mechanism through which *TApplication* passes Help requests on to the Help system. *TApplication* obtains an instance of an object which implements both *IHelpSystem* and *IHelpManager* at application load time and exports that instance as a property; this allows other code within the application to file Help requests directly when appropriate.

IHelpSelector provides a mechanism through which the Help system can invoke the user interface to ask which Help viewer should be used in cases where more than one viewer is capable of handling a Help request, and to display a Table of Contents. This display capability is not built into the Help Manager directly to allow the Help Manager code to be identical regardless of which widget set or class library is in use.

Implementing ICustomHelpViewer

The *ICustomHelpViewer* interface contains three types of methods: methods used to communicate system-level information (for example, information not related to a particular Help request) with the Help Manager; methods related to showing Help based upon a keyword provided by the Help Manager; and methods for displaying a table of contents.

Communicating with the Help Manager

ICustomHelpViewer provides three functions that can be used to communicate system information with the Help Manager:

- *GetViewerName*
- *NotifyID*
- *ShutDown*

The HelpManager calls through these functions in the following circumstances:

- *ICustomHelpViewer.GetViewerName : String* is called when the Help Manager wants to know the name of the viewer (for example, if the application is asked to display a list of all registered viewers). This information is returned via a string, and is required to be logically static (that is, it cannot change during the operation of the application). Multibyte character sets are not supported.
- *ICustomHelpViewer.NotifyID(const ViewerID: Integer)* is called **immediately** following registration to provide the viewer with a unique cookie that identifies it. This information must be stored off for later use; if the viewer shuts down on its own (as opposed to in response to a notification from the Help Manager), it must provide the Help Manager with the identifying cookie so that the Help Manager can release all references to the viewer. (Failing to provide the cookie, or providing the wrong one, causes the Help Manager to potentially release references to the wrong viewer.)
- *ICustomHelpViewer.ShutDown* is called by the Help Manager to notify the Help viewer that the Manager is shutting down and that any resources the Help viewer has allocated should be freed. It is recommended that all resource freeing be delegated to this method.

Asking the Help Manager for information

Help viewers communicate with the Help Manager through the *IHelpManager* interface, an instance of which is returned to them when they register with the Help Manager. *IHelpManager* allows the Help viewer to communicate four things: a request for the window handle of the currently active control; a request for the name of the Help file which the Help Manager believes should contain help for the currently active control; a request for the path to that Help file; and a notification that the Help viewer is shutting itself down in response to something other than a request from the Help Manager that it do so.

IHelpManager.GetHandle : LongInt is called by the Help viewer if it needs to know the handle of the currently active control; the result is a window handle.

IHelpManager.GetHelpFile: String is called by the Help viewer if it wishes to know the name of the Help file which the currently active control believes contains its help.

IHelpManager.GetHelpPath: String is called by the Help viewer if it wishes to know the path of the Help files. This information is *not* provided by default because many external Help systems are able to determine the path via environment variables, etc., and so do not need explicit path references.

IHelpManager.Release is called to notify the Help Manager when a Help viewer is disconnecting. It should *never* be called in response to a request through *ICustomHelpViewer.ShutDown*; it is only used to notify the Help Manager of unexpected disconnects.

Displaying keyword-based Help

Help requests typically come through to the Help viewer as either *keyword-based* Help, in which case the viewer is asked to provide help based upon a particular string, or as *context-based* Help, in which case the viewer is asked to provide help based upon a particular numeric identifier. (Numeric help contexts are the default form of Help requests in applications running under Windows, which use the WinHelp system; while CLX supports them, they are not recommended for use in CLX applications because most Linux Help systems do not understand them.) *ICustomHelpViewer* implementations are required to provide support for keyword-based Help requests, while *IExtendedHelpViewer* implementations are required to support context-based Help requests.

ICustomHelpViewer provides three methods for handling keyword-based Help:

- *CanShowKeyword*
- *GetHelpStrings*
- *ShowHelp*

```
ICustomHelpViewer.CanShowKeyword(const HelpString: String): Integer
```

is the first of the three methods called by the Help Manager, which will call *each* registered Help viewer with the same string to ask if the viewer provides help for that string; the viewer is expected to respond with an integer indicating how many different Help pages it can display in response to that Help request. The viewer can use any method it wants to determine this — inside the Kylix IDE, the HyperHelp viewer maintains its own index and searches it; the Man page viewer, on the other hand, invokes the program man and asks it. If the viewer does not support help on this keyword, it should return zero. Negative numbers are currently interpreted as meaning zero, but this behavior is not guaranteed in future releases.

```
ICustomHelpViewer.GetHelpStrings(const HelpString: String): TStringList
```

is called by the Help Manager if more than one viewer can provide help on a topic. The viewer is expected to return a *TStringList*. The strings in the returned list should map to the pages available for that keyword, but the characteristics of that mapping can be determined by the viewer. In the case of the HyperHelp viewer, the string list always contains exactly one entry (HyperHelp provides its own indexing, and duplicating that elsewhere would be pointless duplication); in the case of the Man page viewer, the string list consists of multiple strings, one for each section of the manual which contains a page for that keyword.

```
ICustomHelpViewer.ShowHelp(const HelpString: String)
```

is called by the Help Manager if it needs the Help viewer to display help for a particular keyword. This is the last method call in the operation; it is guaranteed to never be called unless *CanShowKeyword* is invoked first.

Displaying tables of contents

ICustomHelpViewer provides two methods relating to displaying tables of contents:

- *CanShowTableOfContents*
- *ShowTableOfContents*

The theory behind their operation is similar to the operation of the keyword Help request functions: the Help Manager first queries all Help viewers by calling *ICustomHelpViewer.CanShowTableOfContents : Boolean* and then invokes a particular Help viewer by calling *ICustomHelpViewer.ShowTableOfContents*.

Note that it is perfectly reasonable for a particular viewer to refuse to support requests to support a table of contents. The Man page viewer does this, for example, because the concept of a table of contents does not map well to the way Man pages work; the HyperHelp viewer supports a table of contents, on the other hand, by passing the request to display a table of contents directly to HyperHelp. It is *not* reasonable, however, for an implementation of *ICustomHelpViewer* to respond to queries through *CanShowTableOfContents* with the answer *true*, and then ignore requests through *ShowTableOfContents*.

Implementing IExtendedHelpViewer

ICustomHelpViewer only provides direct support for keyword-based Help. Some Help systems (especially WinHelp) work by associating numbers (known as *context IDs*) with keywords in a fashion which is internal to the Help system and therefore not visible to the application. Such systems require that the application support context-based Help in which the application invokes the Help system with that context, rather than with a string, and the Help system translates the number itself.

Applications written in CLX can talk to systems requiring context-based Help by extending the object which implements *ICustomHelpViewer* to also implement *IExtendedHelpViewer*. *IExtendedHelpViewer* also provides support for talking to Help systems that allow you to jump directly to high-level topics instead of using keyword searches.

IExtendedHelpViewer exposes four functions. Two of them — *UnderstandsContext* and *DisplayHelpByContext* — are used to support context-based Help; the other two — *UnderstandsTopic* and *DisplayTopic* — are used to support topics.

When an application user presses F1, the Help Manager calls

```
IExtendedHelpViewer.UnderstandsContext(const ContextID: Integer;
const HelpFileName: String): Boolean
```

and the currently activated control supports context-based, rather than keyword-based Help. As with *ICustomHelpViewer.CanShowKeyword*, the Help Manager queries all registered Help viewers iteratively. Unlike the case with *ICustomHelpViewer.CanShowKeyword*, however, if more than one viewer supports a specified context, the *first* registered viewer with support for a given context is invoked.

The Help Manager calls

```
IExtendedHelpViewer.DisplayHelpByContext(const ContextID: Integer;
const HelpFileName: String)
```

after it has polled the registered Help viewers.

The topic support functions work the same way:

```
IExtendedHelpViewer.UnderstandsTopic(const Topic: String): Boolean
```

is used to poll the Help viewers asking if they support a topic;

```
IExtendedHelpViewer.DisplayTopic(const Topic: String)
```

is used to invoke the first registered viewer which reports that it is able to provide help for that topic.

Implementing IHelpSelector

IHelpSelector is a companion to *ICustomHelpViewer*. When more than one registered viewer claims to provide support for a given keyword, context, or topic, or provides a table of contents, the Help Manager must choose between them. In the case of contexts or topics, the Help Manager *always* selects the first Help viewer that claims to provide support. In the case of keywords or the table of context, the Help Manager will, by default, select the first Help viewer. This behavior can be overridden by an application.

To override the decision of the Help Manager in such cases, an application must register a class that provides an implementation of the *IHelpSelector* interface. *IHelpSelector* exports two functions: *SelectKeyword*, and *TableOfContents*. Both take as arguments a *TStrings* containing, one by one, either the possible keyword matches or the names of the viewers claiming to provide a table of contents. The implementor is required to return the index (in the *TStrings*) that represents the selected string.

Note The Help Manager may get confused if the strings are re-arranged; it is recommended that implementors of *IHelpSelector* refrain from doing this. The Help system only supports *one* HelpSelector; when new selectors are registered, any previously existing selectors are disconnected.

Registering Help system objects

For the Help Manager to communicate with them, objects that implement *ICustomHelpViewer, IExtendedHelpViewer, ISpecialWinHelpViewer*, and *IHelpSelector* must register with the Help Manager.

To register Help system objects with the Help Manager, you need to

- Register the Help viewer
- Register the Help Selector

Registering Help viewers

The unit that contains the object implementation must use HelpIntfs. An instance of the object must be declared in the **var** section of the implementing unit.

The initialization section of the implementing unit must assign the instance variable and pass it to the function *RegisterViewer*. *RegisterViewer* is a flat function exported by

HelpIntfs.pas which takes as an argument an *ICustomHelpViewer* and returns an *IHelpManager*. The *IHelpManager* should be stored for future use.

Registering Help selectors

The unit that contains the object implementation must use HelpIntfs and QForms. An instance of the object must be declared in the **var** section of the implementing unit.

The initialization section of the implementing unit must register the Help selector through the *HelpSystem* property of the global Application object:

```
Application.HelpSystem.AssignHelpSelector(myHelpSelectorInstance)
```

This procedure does not return a value.

Using Help in a CLX Application

The following sections explain how to use Help within a CLX application.

How TApplication processes Help

TApplication provides two methods that are accessible from application code:

- *ContextHelp*, which invokes the Help system with a request for context-based Help
- *KeywordHelp*, which invokes the Help system with a request for keyword-based Help

Both functions take as an argument the context or keyword being passed and forward the request on through a data member of *TApplication,* which represents the Help system. That data member is directly accessible through the read-only property *HelpSystem*.

How controls process Help

All controls that derive from *TControl* expose four properties which are used by the Help system: *HelpType*, *HelpFile*, *HelpContext*, and *HelpKeyword*. *HelpFile* is supposed to contain the name of the file in which the control's help is located; if the help is located in an external Help system that does not care about file names (say, for example, the Man page system), then the property should be left blank.

The *HelpType* property contains an instance of an enumerated type which determines if the control's designer expects help to be provided via keyword-based Help or context-based Help; the other two properties are linked to it. If the *HelpType* is set to *htKeyword*, then the Help system expects the control to use keyword-based Help, and the Help system only looks at the contents of the *HelpKeyword* property. Conversely, if the *HelpType* is set to *htContext*, the Help system expects the control to use context-based Help and only looks at the contents of the *HelpContext* property.

In addition to the properties, controls expose a single method, *InvokeHelp*, which can be called to pass a request to the Help system. It takes no parameters and calls the methods in the global Application object, which correspond to the type of help the control supports.

Help messages are automatically invoked when F1 is pressed because the *KeyDown* method of *TWidgetControl* calls *InvokeHelp*.

Calling the Help system directly

For additional Help system functionality not exposed by this mechanism, *TApplication* provides a read-only property that allows direct access to the Help system. This property is an instance of an implementation of the interface *IHelpSystem*. *IHelpSystem* and *IHelpManager* are implemented by the same object, but one interface is used to allow the application to talk to the Help Manager, and one is used to allow the Help viewers to talk to the Help Manager.

Using IHelpSystem

IHelpSystem allows the application to do three things:

- Provides path information to the Help Manager
- Provides a new Help selector
- Asks the Help Manager to display help

Providing path information is important because the Help Manager is platform-independent and Help system-independent and so is not able to ascertain the location of Help files. If an application expects help to be provided by an external Help system that is not able to ascertain file locations itself, it must provide this information through *IHelpSystem.ProvideHelpPath*, which allows the information to become available through *IHelpManager.GetHelpPath*. (This information propagates outward only if the Help viewer asks for it.)

Assigning a Help selector allows the Help Manager to delegate decision-making in cases where multiple external Help systems can provide help for the same keyword. For more information, see the section “Implementing IHelpSelector” on page 5-17.

IHelpSystem exports four functions used to request the Help Manager to display help:

- *ShowHelp*
- *ShowContextHelp*
- *ShowTopicHelp*
- *Hook*

Hook is intended entirely for WinHelp compatibility and should not be used in a CLX application; it allows processing of WM_HELP messages that cannot be mapped directly onto requests for keyword-based, context-based, or topic-based Help. The other methods each take two arguments: the keyword, context ID, or topic for which help is being requested, and the Help file in which it is expected that help can be found.

In general, unless you are asking for topic-based help, it is equally effective and more clear to pass help requests to the Help Manager through the *InvokeHelp* method of your control.

Customizing the IDE Help system

The Kylix IDE supports multiple Help viewers in exactly the same fashion that a CLX application does: it delegates Help requests to the Help Manager, which forwards them to registered Help viewers. The IDE comes with two Help viewers installed: the HyperHelp viewer, which allows Help requests to be forwarded to HyperHelp, an external WinHelp emulator under which the Kylix Help files are viewed, and the Man page viewer, which allows you to access the Man system installed on most Unix machines. Because it is necessary for Kylix Help to work, the HyperHelp viewer may not be removed; the Man page viewer ships in a separate package whose source is available in the examples directory.

To install a new Help viewer in the IDE, you do exactly what you would do in a CLX application, with one difference. You write an object that implements *ICustomHelpViewer* (and, if desired, *IExtendedHelpViewer*) to forward Help requests to the external viewer of your choice, and you register the *ICustomHelpViewer* with the IDE.

To register a custom Help viewer with the IDE,

1. Make sure that the unit implementing the Help viewer contains HelpIntfs.pas.
2. Build the unit into a design-time package registered with the IDE, and build the package with runtime packages turned on. (This is necessary to ensure that the Help Manager instance used by the unit is the same as the Help Manager instance used by the IDE.)
3. Make sure that the Help viewer exists as a global instance within the unit.
4. In the *Register()* function of the unit, make sure that the instance is passed to the function *RegisterHelpViewer.*

Chapter

6

Developing the application user interface

With Kylix, you design a user interface (UI) by selecting components from the Component palette and dropping them onto forms. You get it to do what you want by setting the components' properties and coding their event handlers.

Controlling application behavior

TApplication, *TScreen*, and *TForm* are the classes that form the backbone of all Kylix applications by controlling the behavior of your project. The *TApplication* class forms the foundation of an application by providing properties and methods that encapsulate the behavior of a standard Linux program. *TScreen* is used at runtime to keep track of forms and data modules that have been loaded as well as maintaining system-specific information, such as screen resolution and available display fonts. Instances of *TForm* are the building blocks of your application's user interface. The windows and dialog boxes in your application are based on *TForm*.

Using the main form

TForm is the key class for creating GUI applications. When you open Kylix displaying a default project or create a new project, a form is displayed for you to begin your UI design.

The first form you create and save in a project becomes, by default, the project's main form, which is the first form created at runtime. As you add forms to your projects, you might decide to designate a different form as your application's main form. Also, specifying a form as the main form is an easy way to test it at runtime, because unless you change the form creation order, the main form is the first form displayed in the running application.

To change the project main form,

1 Choose Project | Options and select the Forms page.

2 In the Main Form combo box, select the form you want to use as the project's main form and choose OK.

Now if you run the application, the form you selected as the main form is displayed first.

Adding forms

To add a form to your project, select File | New Form. You can see all your project's forms and their associated units listed in the Project Manager (View | Project Manager) and you can display a list of the forms alone by choosing View | Forms.

Linking forms

Adding a form to a project adds a reference to it in the project file, but not to any other units in the project. Before you can write code that references the new form, you need to add a reference to it in the referencing forms' unit files. This is called *form linking*.

A common reason to link forms is to provide access to the components in that form. For example, you'll often use form linking to enable a form that contains data-aware components to connect to the data-access components in a data module.

To link a form to another form,

1 Select the form that needs to refer to another.
2 Choose File | Use Unit.
3 Select the name of the form unit for the form to be referenced.
4 Choose OK.

Linking a form to another just means that the **uses** clauses of one form unit contains a reference to the other's form unit, meaning that the linked form and its components are now in scope for the linking form.

Avoiding circular unit references

When two forms must reference each other, it's possible to cause a "Circular reference" error when you compile your program. To avoid such an error, do one of the following:

- Place both **uses** clauses, with the unit identifiers, in the **implementation** parts of the respective unit files. (This is what the File | Use Unit command does.)
- Place one **uses** clause in an **interface** part and the other in an **implementation** part. (You rarely need to place another form's unit identifier in this unit's **interface** part.)

Do not place both **uses** clauses in the **interface** parts of their respective unit files. This will generate the "Circular reference" error at compile time.

Hiding the main form

You can prevent the main form from displaying when your application first starts. To do so, you must use the global *Application* variable (described in the next topic).

To hide the main form at startup,

1 Choose Project | View Source to display the main project file.

2 Add the following lines after the call to `Application.CreateForm` and before the call to `Application.Run`.

```
Application.ShowMainForm := False;
Form1.Visible := False; { the name of your main form may differ }
```

Note You can set the form's *Visible* property to *False* using the Object Inspector at design time rather than setting it at runtime as shown above.

Working at the application level

The global variable *Application*, of type *TApplication*, is in every CLX-based application. *Application* encapsulates your application as well as providing many functions that occur in the background of the program. For instance, *Application* would handle how you would call a help file from the menu of your program. Understanding how *TApplication* works is more important to a component writer than to developers of stand-alone applications, but you should set the options that *Application* handles in the Project | Options Application page when you create a project.

In addition, *Application* receives many events that apply to the application as a whole. For example, the *OnActivate* event lets you perform actions when the application first starts up, the *OnIdle* event lets you perform background processes when the application is not busy, the *OnEvent* event lets you intercept events, and so on. Although you can't use the IDE to examine the properties and events of the global *Application* variable, another component, *TApplicationEvents*, intercepts the events and lets you supply event-handlers using the IDE.

Setting up the look and feel of your application

You can use the *TApplication.Style* property to specify the general look and feel of an application's graphical elements. *TApplication.Style* holds an instance of *TApplicationStyle*, which is declared, along with its ancestor *TStyle*, in QStyle.pas. *TStyle.DefaultStyle* can be set to dsMotif or dsWindows. For example, the following code sets the general look of your application to a Motif style:

```
Application.Style.DefaultStyle := dsMotif;
```

Most of *TStyle*'s other properties are custom event handlers that control the drawing of buttons, track bars, splitters, check boxes, and so on. (You can also control the individual tabs of a tab control, individual button controls, and individual menu items.) The CLXStyle sample application shows how to implement custom drawing.

Handling the screen

A global variable of type *TScreen* called *Screen* is created when you create a project. Screen encapsulates the state of the screen on which your application is running. Common tasks performed by *Screen* include specifying

- the look of the cursor
- the size of the window in which your application is running
- a list of fonts available to the screen device

By default, applications create a screen component based on information about the current screen device and assign it to Screen.

Managing layout

At its simplest, you control the layout of your user interface by where you place controls in your forms. The placement choices you make are reflected in the control's *Top*, *Left*, *Width*, and *Height* properties. You can change these values at runtime to change the position and size of the controls in your forms.

Controls have a number of other properties, however, that allow them to automatically adjust to their contents or containers. This allows you to lay out your forms so that the pieces fit together into a unified whole.

Two properties affect how a control is positioned and sized in relation to its parent. The *Align* property lets you force a control to fit perfectly within its parent along a specific edge or filling up the entire client area after any other controls have been aligned. When the parent is resized, the controls aligned to it are automatically resized and remain positioned so that they fit against a particular edge.

Controls that have contents that can change in size have an *AutoSize* property that causes the control to adjust its size to its font or contained objects.

Using forms

When you create a form in Kylix from the IDE, Kylix automatically creates the form in memory by including code in the main entry point of your application. Usually, this is the desired behavior and you don't have to do anything to change it. That is, the main window persists through the duration of your program, so you would likely not change the default Kylix behavior when creating the form for your main window.

However, you may not want all your application's forms in memory for the duration of the program execution. That is, if you do not want all your application's dialogs in memory at once, you can create the dialogs dynamically when you want them to appear.

Forms can be modal or modeless. Modal forms are forms with which the user must interact before switching to another form (for example, a dialog box requiring user input). Modeless forms are windows that are displayed until they are either obscured by another window or until they are closed or minimized by the user.

Controlling when forms reside in memory

By default, Kylix automatically creates the application's main form in memory by including the following code in the application's project source unit:

```
Application.CreateForm(TForm1, Form1);
```

This function creates a global variable with the same name as the form. So, every form in an application has an associated global variable. This variable is a pointer to an instance of the form's class and is used to reference the form while the application is running. Any unit that includes the form's unit in its **uses** clause can access the form via this variable.

All forms created in this way in the project unit appear when the program is invoked and exist in memory for the duration of the application.

Displaying an auto-created form

If you choose to create a form at startup and do not want it displayed until sometime later during program execution, the form's event handler uses the *ShowModal* method to display the form that is already loaded in memory:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  ResultsForm.ShowModal;
end;
```

In this case, since the form is already in memory, there is no need to create another instance or destroy that instance.

Creating forms dynamically

You may not always want all your application's forms in memory at once. To reduce the amount of memory required at load time, you may want to create some forms only when you need to use them. For example, a dialog box needs to be in memory only during the time a user interacts with it.

To create a form at a different stage during execution using the IDE, you:

1 Select the File|New Form from the main menu to display the new form.
2 Remove the form from the Auto-create forms list of the Project|Options|Forms page.

  This removes the form's invocation at startup. As an alternative, you can manually remove the following line from the project source*:*

  ```
  Application.CreateForm(TResultsForm, ResultsForm);
  ```

3 Invoke the form when desired by using the form's *Show* method, if the form is modeless, or *ShowModal* method, if the form is modal.

An event handler for the main form must create an instance of the result form and destroy it. One way to invoke the result form is to use the global variable as follows. Note that *ResultsForm* is a modal form so the handler uses the *ShowModal* method.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  ResultsForm:=TResultForm.Create(self)
  ResultsForm.ShowModal;
  ResultsForm.Free;
end;
```

The event handler in the example deletes the form after it is closed, so the form would need to be recreated if you needed to use *ResultsForm* elsewhere in the application. If the form were displayed using *Show* you could not delete the form within the event handler because *Show* returns while the form is still open.

Note If you create a form using its constructor, be sure to check that the form is not in the Auto-create forms list on the Project Options|Forms page. Specifically, if you create the new form without deleting the form of the same name from the list, Kylix creates the form at startup and this event-handler creates a new instance of the form, overwriting the reference to the auto-created instance. The auto-created instance still exists, but the application can no longer access it. After the event-handler terminates, the global variable no longer points to a valid form. Any attempt to use the global variable will likely crash the application.

Creating modeless forms such as windows

You must guarantee that reference variables for modeless forms exist for as long as the form is in use. This means that these variables should have global scope. In most cases, you use the global reference variable that was created when you made the form (the variable name that matches the name property of the form). If your application requires additional instances of the form, declare separate global variables for each instance.

Using a local variable to create a form instance

A safer way to create a unique instance of a *modal form* is to use a local variable in the event handler as a reference to a new instance. If a local variable is used, it does not matter whether *ResultsForm* is auto-created or not. The code in the event handler makes no reference to the global form variable. For example:

```
procedure TMainForm.Button1Click(Sender: TObject);
var
  RF:TResultForm;
begin
  RF:=TResultForm.Create(self)
  RF.ShowModal;
  RF.Free;
end;
```

Notice how the global instance of the form is never used in this version of the event handler.

Typically, applications use the global instances of forms. However, if you need a new instance of a modal form, and you use that form in a limited, discrete section of the application, such as a single function, a local instance is usually the safest and most efficient way of working with the form.

Of course, you cannot use local variables in event handlers for modeless forms because they must have global scope to ensure that the forms exist for as long as the form is in use. *Show* returns as soon as the form opens, so if you used a local variable, the local variable would go out of scope immediately.

Passing additional arguments to forms

Typically, you create forms for your application from within the IDE. When created this way, the forms have a constructor that takes one argument, *Owner*, which is the owner of the form being created. (The owner is the calling application object or form object.) *Owner* can be **nil**.

To pass additional arguments to a form, create a separate constructor and instantiate the form using this new constructor. The example form class below shows an additional constructor, with the extra argument *whichButton*. This new constructor is added to the form class manually.

```
TResultsForm = class(TForm)
  ResultsLabel: TLabel;
  OKButton: TButton;
  procedure OKButtonClick(Sender: TObject);
private
public
  constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
end;
```

Here's the manually coded constructor that passes the additional argument, *whichButton*. This constructor uses the *whichButton* parameter to set the *Caption* property of a *Label* control on the form.

```
constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
begin
  case whichButton of
    1: ResultsLabel.Caption := 'You picked the first button.';
    2: ResultsLabel.Caption := 'You picked the second button.';
    3: ResultsLabel.Caption := 'You picked the third button.';
  end;
end;
```

When creating an instance of a form with multiple constructors, you can select the constructor that best suits your purpose. For example, the following *OnClick* handler for a button on a form calls creates an instance of *TResultsForm* that uses the extra parameter:

```
procedure TMainForm.SecondButtonClick(Sender: TObject);
var
  rf: TResultsForm;
begin
  rf := TResultsForm.CreateWithButton(2, self);
```

```
    rf.ShowModal;
    rf.Free;
  end;
```

Retrieving data from forms

Most real-world applications consist of several forms. Often, information needs to be passed between these forms. Information can be passed to a form by means of parameters to the receiving form's constructor, or by assigning values to the form's properties. The way you get information from a form depends on whether the form is modal or modeless.

Retrieving data from modeless forms

You can easily extract information from modeless forms by calling public member functions of the form or by querying properties of the form. For example, assume an application contains a modeless form called *ColorForm* that contains a listbox called *ColorListBox* with a list of colors ("Red", "Green", "Blue", and so on). The selected color name string in *ColorListBox* is automatically stored in a property called *CurrentColor* each time a user selects a new color. The class declaration for the form is as follows:

```
TColorForm = class(TForm)
  ColorListBox:TListBox;
  procedure ColorListBoxClick(Sender: TObject);
private
  FColor:String;
public
  property CurColor:String read FColor write FColor;
end;
```

The *OnClick* event handler for the listbox, *ColorListBoxClick*, sets the value of the *CurrentColor* property each time a new item in the listbox is selected. The event handler gets the string from the listbox containing the color name and assigns it to *CurrentColor*. The *CurrentColor* property uses the setter function, *SetColor,* to store the actual value for the property in the private data member *FColor*:

```
procedure TColorForm.ColorListBoxClick(Sender: TObject);
var
  Index: Integer;
begin
  Index := ColorListBox.ItemIndex;
  if Index >= 0 then
    CurrentColor := ColorListBox.Items[Index]
  else
    CurrentColor := '';
end;
```

Now suppose that another form within the application, called *ResultsForm*, needs to find out which color is currently selected on *ColorForm* whenever a button (called *UpdateButton)* on *ResultsForm* is clicked. The *OnClick* event handler for *UpdateButton* might look like this:

```
procedure TResultForm.UpdateButtonClick(Sender: TObject);
var
  MainColor: String;
begin
  if Assigned(ColorForm) then
  begin
    MainColor := ColorForm.CurrentColor;
    {do something with the string MainColor}
  end;
end;
```

The event handler first verifies that *ColorForm* exists using the *Assigned* function. It then gets the value of *ColorForm's CurrentColor* property.

Alternatively, if *ColorForm* had a public function named *GetColor*, another form could get the current color without using the *CurrentColor* property (for example, `MainColor := ColorForm.GetColor;`). In fact, there's nothing to prevent another form from getting the *ColorForm's* currently selected color by checking the listbox selection directly:

```
with ColorForm.ColorListBox do
  MainColor := Items[ItemIndex];
```

However, using a property makes the interface to *ColorForm* very straightforward and simple. All a form needs to know about *ColorForm* is to check the value of *CurrentColor*.

Retrieving data from modal forms

Just like modeless forms, modal forms often contain information needed by other forms. The most common example is form A launches modal form B. When form B is closed, form A needs to know what the user did with form B to decide how to proceed with the processing of form A. If form B is still in memory, it can be queried through properties or member functions just as in the modeless forms example above. But how do you handle situations where form B is deleted from memory upon closing? Since a form does not have an explicit return value, you must preserve important information from the form before it is destroyed.

To illustrate, consider a modified version of the *ColorForm* form that is designed to be a modal form. The class declaration is as follows:

```
TColorForm = class(TForm)
  ColorListBox:TListBox;
  SelectButton: TButton;
  CancelButton: TButton;
  procedure CancelButtonClick(Sender: TObject);
  procedure SelectButtonClick(Sender: TObject);
private
  FColor: Pointer;
public
  constructor CreateWithColor(Value: Pointer; Owner: TComponent);
end;
```

The form has a listbox called *ColorListBox* with a list of names of colors. When pressed, the button called *SelectButton* makes note of the currently selected color name in *ColorListBox* then closes the form. *CancelButton* is a button that simply closes the form.

Note that a user-defined constructor was added to the class that takes a *Pointer* argument. Presumably, this *Pointer* points to a string that the form launching *ColorForm* knows about. The implementation of this constructor is as follows:

```
constructor TColorForm(Value: Pointer; Owner: TComponent);
begin
  FColor := Value;
  String(FColor^) := '';
end;
```

The constructor saves the pointer to a private data member *FColor* and initializes the string to an empty string.

Note To use the above user-defined constructor, the form must be explicitly created. It cannot be auto-created when the application is started. For details, see "Controlling when forms reside in memory" on page 6-4.

In the application, the user selects a color from the listbox and presses *SelectButton* to save the choice and close the form. The *OnClick* event handler for *SelectButton* might look like this:

```
procedure TColorForm.SelectButtonClick(Sender: TObject);
begin
  with ColorListBox do
    if ItemIndex >= 0 then
      String(FColor^) := ColorListBox.Items[ItemIndex];
  end;
  Close;
end;
```

Notice that the event handler stores the selected color name in the string referenced by the pointer that was passed to the constructor.

To use *ColorForm* effectively, the calling form must pass the constructor a pointer to an existing string. For example, assume *ColorForm* was instantiated by a form called *ResultsForm* in response to a button called *UpdateButton* on *ResultsForm* being clicked. The event handler would look as follows:

```
procedure TResultsForm.UpdateButtonClick(Sender: TObject);
var
  MainColor: String;
begin
  GetColor(Addr(MainColor));
  if MainColor <> '' then
    {do something with the MainColor string}
  else
    {do something else because no color was picked}
end;

procedure GetColor(PColor: Pointer);
begin
  ColorForm := TColorForm.CreateWithColor(PColor, Self);
  ColorForm.ShowModal;
  ColorForm.Free;
end;
```

UpdateButtonClick creates a String called MainColor. The address of MainColor is passed to the *GetColor* function which creates *ColorForm*, passing the pointer to MainColor as an argument to the constructor. As soon as *ColorForm* is closed it is deleted, but the color name that was selected is still preserved in MainColor, assuming that a color was selected. Otherwise, MainColor contains an empty string which is a clear indication that the user exited *ColorForm* without selecting a color.

This example uses one string variable to hold information from the modal form. Of course, more complex objects can be used depending on the need. Keep in mind that you should always provide a way to let the calling form know if the modal form was closed without making any changes or selections (such as having MainColor default to an empty string).

Working with frames

A frame (*TFrame*), like a form, is a container for other components. It uses the same ownership mechanism as forms for automatic instantiation and destruction of the components on it, and the same parent-child relationships for synchronization of component properties.

In some ways, however, a frame is more like a customized component than a form. Frames can be saved on the Component palette for easy reuse, and they can be nested within forms, other frames, or other container objects. After a frame is created and saved, it continues to function as a unit and to inherit changes from the components (including other frames) it contains. When a frame is embedded in another frame or form, it continues to inherit changes made to the frame from which it derives.

Frames are useful to organize groups of controls that are used in multiple places in your application. For example, if you have a bitmap that is used on multiple forms, you can put it in a frame and only one copy of that bitmap is included in the resources of your application. You could also describe a set of edit fields that are intended to edit a table with a frame and use that whenever you want to enter data into the table.

Creating frames

To create an empty frame, choose File | New Frame, or choose File | New and double-click on Frame. You can then drop components (including other frames) onto your new frame.

It is usually best—though not necessary—to save frames as part of a project. If you want to create a project that contains only frames and no forms, choose File | New Application, close the new form and unit without saving them, then choose File | New Frame and save the project.

Note When you save frames, avoid using the default names *Unit1*, *Project1*, and so forth, since these are likely to cause conflicts when you try to use the frames later.

At design time, you can display any frame included in the current project by choosing View | Forms and selecting a frame. As with forms and data modules, you can toggle between the Form Designer and the frame's form file by right-clicking and choosing View as Form or View as Text.

Adding frames to the Component palette

Frames are added to the Component palette as component templates. To add a frame to the Component palette, open the frame in the Form Designer (you cannot use a frame embedded in another component for this purpose), right-click on the frame, and choose Add to Palette. When the Component Template Information dialog opens, select a name, palette page, and icon for the new template.

Using and modifying frames

To use a frame in an application, you must place it, directly or indirectly, on a form. You can add frames directly to forms, to other frames, or to other container objects such as panels and scroll boxes.

The Form Designer provides two ways to add a frame to an application:

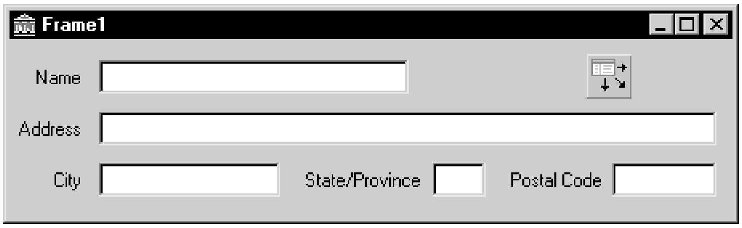
- Select a frame from the Component palette and drop it onto a form, another frame, or another container object. If necessary, the Form Designer asks for permission to include the frame's unit file in your project.
- Select *Frames* from the Standard page of the Component palette and click on a form or another frame. A dialog appears with a list of frames that are already included in your project; select one and click OK.

When you drop a frame onto a form or other container, Kylix declares a new class that descends from the frame you selected. (Similarly, when you add a new form to a project, Kylix declares a new class that descends from *TForm*.) This means that changes made later to the original (ancestor) frame propagate to the embedded frame, but changes to the embedded frame do not propagate backward to the ancestor.

Suppose, for example, that you wanted to assemble a group of data-access components and data-aware controls for repeated use, perhaps in more than one application. One way to accomplish this would be to collect the components into a component template; but if you started to use the template and later changed your mind about the arrangement of the controls, you would have to go back and manually alter each project where the template was placed.

If, on the other hand, you put your database components into a frame, later changes would need to be made in only one place; changes to an original frame automatically propagate to its embedded descendants when your projects are recompiled. At the same time, you are free to modify any embedded frame without affecting the original frame or other embedded descendants of it. The only limitation on modifying embedded frames is that you cannot add components to them.

Figure 6.1 A frame with data-aware controls and a data source component



In addition to simplifying maintenance, frames can help you to use resources more efficiently. For example, to use a bitmap or other graphic in an application, you might load the graphic into the *Picture* property of a *TImage* control. If, however, you use the same graphic repeatedly in one application, each *Imag*e object you place on a form will result in another copy of the graphic being added to the form's resource file. (This is true even if you set *TImage.Picture* once and save the *Image* control as a component template.) A better solution is to drop the *Image* object onto a frame, load your graphic into it, then use the frame where you want the graphic to appear. This results in smaller form files and has the added advantage of letting you change the graphic everywhere it occurs simply by modifying the *Image* on the original frame.

Sharing frames

You can share a frame with other developers in two ways:

- Add the frame to the Object Repository.
- Distribute the frame's unit (.pas) and form (.xfm) files.

To add a frame to the Repository, open any project that includes the frame, right-click in the Form Designer, and choose Add to Repository. For more information, see "Using the Object Repository" on page 2-35.

If you send a frame's unit and form files to other developers, they can open them and add them to the Component palette. If the frame has other frames embedded in it, the frame must be opened as part of a project in order to add it to the palette.

Using action lists

Action lists maintain a list of actions that your application can take in response to something a user does. By using action objects, you centralize the functions performed by your application from the user interface. This lets you share common code for performing actions (for example, when a toolbar button and menu item do the same thing), as well as providing a single, centralized way to enable and disable actions depending on the state of your application.

What is an action?

An action corresponds to one or more elements of the user interface, such as menu commands or toolbar buttons. Actions serve two functions: (1) they represent properties common to the user interface elements, such as whether a control is enabled or checked, and (2) they respond when a control fires, for example, when the application user clicks a button or chooses a menu item.

Actions are associated with other components:

- **Clients:** One or more clients use the action. The client most often represents a menu item or a button (for example, *TToolButton*, *TSpeedButton*, *TMenuItem*, *TButton*, *TCheckBox, TRadioButton*, and so on). When the client receives a user command (such as a mouse click), it initiates an associated action. Typically, a client's *OnClick* event is associated with its action's *Execute* event.
- **Target:** The action acts on the target. The target is usually a control, such as a memo or a data control. Component writers can create actions specific to the needs of the controls they design and use, and then package those units to create more modular applications. Not all actions use a target. For example, the standard help actions ignore the target and simply launch the help system.

  A target can also be a component. For example, data controls change the target to an associated dataset.

The client influences the action—the action responds when a client fires the action. The action also influences the client—action properties dynamically update the client properties. For example, if at runtime an action is disabled (by setting its *Enabled* property to *False*), every client of that action is disabled, appearing grayed.

You add, delete, and rearrange actions using the Action List editor (displayed by double-clicking an action list object). These actions are later connected to client controls.

Setting up action lists

Setting up action lists is fairly easy once you understand the basic steps involved:

- Create the action list.
- Add actions to the action list.
- Set properties on the actions.
- Attach clients to the action.

Here are the steps in more detail:

1. Drop a *TActionList* object onto your form or data module. (ActionList is on the Standard page of the component palette.)
2. Double-click the *TActionList* object to display the Action List editor.
   a. Use one of the predefined actions listed in the editor: right-click and choose New Standard Action.

  b The predefined actions are organized into categories (such as Dataset, Edit, Help, and Window) in the Standard Actions dialog box. Select all the standard actions you want to add to the action list and click OK.

  or

  c Create a new action of your own: right-click and choose New Action.

3 Set the properties of each action in the Object Inspector. (The properties you set affect every client of the action.)

The *Name* property identifies the action, and the other properties and events (*Caption*, *Checked*, *Enabled*, *HelpContext*, *Hint*, *ImageIndex*, *ShortCut*, *Visible,* and *Execute*) correspond to the properties and events of its client controls. The client's corresponding properties are typically, but not necessarily, the same name as the corresponding client property. For example, an action's *Enabled* property corresponds to a *TToolButton*'s *Enabled* property. However, an action's *Checked* property corresponds to a *TToolButton*'s *Down* property.

4 If you use the predefined actions, the action includes a standard response that occurs automatically. If creating your own action, you need to write an event handler that defines how the action responds when fired. See "What happens when an action fires" on page 6-15 for details.

5 Attach the actions in the action list to the clients that require them:

- Click on the control (such as the button or menu item) on the form or data module. In the Object Inspector, the *Action* property lists the available actions.
- Select the one you want.

The standard actions, such as *TEditDelete* or *TDataSetPost*, all perform the action you would expect. You can look at the online reference Help for details on how all of the standard actions work if you need to. If writing your own actions, you'll need to understand more about what happens when the action is fired.

What happens when an action fires

When an event fires, a series of events intended primarily for generic actions occurs. Then if the event doesn't handle the action, another sequence of events occurs.

Responding with events

When a client component or control is clicked or otherwise acted on, a series of events occurs to which you can respond. For example, the following code illustrates the event handler for an action that toggles the visibility of a toolbar when the action is executed:

```
procedure TForm1.Action1Execute(Sender: TObject);
begin
  { Toggle Toolbar1's visibility }
  ToolBar1.Visible := not ToolBar1.Visible;
end;
```

Note For general information about events and event handlers, see “Working with events and event handlers” on page 2-24.

You can supply an event handler that responds at one of three different levels: the action, the action list, or the application. This is only a concern if you are using a new generic action rather than a predefined standard action. You do not have to worry about this if using the standard actions because standard actions have built-in behavior that executes when these events occur.

The order in which the event handlers will respond to events is as follows:

- Action list
- Application
- Action

When the user clicks on a client control, Kylix calls the action's Execute method which defers first to the action list, then the Application object, then the action itself if neither action list nor Application handles it. To explain this in more detail, Kylix follows this dispatching sequence when looking for a way to respond to the user action:

1. If you supply an *OnExecute* event handler for the action list and it handles the action, the application proceeds.

   The action list’s event handler has a parameter called *Handled*, that returns *False* by default. If the handler is assigned and it handles the event, it returns *True*, and the processing sequence ends here. For example:

   ```
   procedure TForm1.ActionList1ExecuteAction(Action: TBasicAction; var Handled: Boolean);
   begin
     Handled := True;
   end;
   ```

   If you don’t set *Handled* to *True* in the action list event handler, then processing continues.

2. If you did not write an *OnExecute* event handler for the action list or if the event handler doesn’t handle the action, the application’s *OnActionExecute* event handler fires. If it handles the action, the application proceeds.

   The global *Application* object receives an *OnActionExecute* event if any action list in the application fails to handle an event. Like the action list’s *OnExecute* event handler, the *OnActionExecute* handler has a parameter *Handled* that returns *False* by default. If an event handler is assigned and handles the event, it returns *True*, and the processing sequence ends here. For example:

   ```
   procedure TForm1.ApplicationExecuteAction(Action: TBasicAction; var Handled: Boolean);
   begin
     { Prevent execution of all actions in Application }
     Handled := True;
   end;
   ```

3. If the application’s *OnExecute* event handler doesn’t handle the action, the action’s *OnExecute* event handler fires.

You can use built-in actions or create your own action classes that know how to operate on specific target classes (such as edit controls). When no event handler is found at any level, the application next tries to find a target on which to execute the action. When the application locates a target that the action knows how to address, it invokes the action. See "How actions find their targets" on page 6-17 for details on how the application locates a target that can respond to a predefined action class.

How actions find their targets

"What happens when an action fires" on page 6-15 describes the execution cycle that occurs when a user invokes an action. If no event handler is assigned to respond to the action, either at the action list, application, or action level, then the application tries to identify a target object to which the action can apply itself.

The application looks for the target using the following sequence:

1 Active control: The application looks first for an active control as a potential target.
2 Active form: If the application does not find an active control or if the active control can't act as a target, it looks at the screen's *ActiveForm*.
3 Controls on the form: If the active form is not an appropriate target, the application looks at the other controls on the active form for a target.

If no target is located, nothing happens when the event is fired.

Some controls can expand the search to defer the target to an associated component; for example, data-aware controls defer to the associated dataset component. Also, some predefined actions do not use a target; for example, the File Open dialog.

Updating actions

When the application is idle, the *OnUpdate* event occurs for every action that is linked to a control or menu item that is showing. This provides an opportunity for applications to execute centralized code for enabling and disabling, checking and unchecking, and so on. For example, the following code illustrates the *OnUpdate* event handler for an action that is "checked" when the toolbar is visible:

```
procedure TForm1.Action1Update(Sender: TObject);
begin
  { Indicate whether ToolBar1 is currently visible }
  (Sender as TAction).Checked := ToolBar1.Visible;
end;
```

Warning Do not add time-intensive code to the *OnUpdate* event handler. This executes whenever the application is idle. If the event handler takes too much time, it will adversely affect performance of the entire application.

Predefined action classes

The Action List editor lets you use predefined action classes that automatically perform actions. The predefined actions fall into the following categories:

Table 6.1 Action categories

Category	Description
Standard edit actions	Used with an edit control target. *TEditAction* is the base class for descendants that each override the *ExecuteTarget* method to implement copy, cut, and paste tasks by using the clipboard.
Standard window actions	Used with forms as targets in an MDI application. *TWindowAction* is the base class for descendants that each override the *ExecuteTarget* method to implement arranging, cascading, closing, tiling, and minimizing MDI child forms.
Standard Help actions	Used with any target. *THelpAction* is the base class for descendants that each override the *ExecuteTarget* method to pass the command onto a Help system.
DataSet actions	Used with a dataset component target. *TDataSetAction* is the base class for descendants that each override the *ExecuteTarget* and *UpdateTarget* methods to implement navigation and editing of the target. *TDataSetAction* introduces a *DataSource* property that ensures actions are performed on that dataset. If *DataSource* is nil, the currently focused data-aware control is used.
Dialog actions	Used with dialog components. *TDialogAction* implements the common behavior for actions that display a dialog when executed. Each descendant class represents a specific dialog.
File actions	Used with operations on files such as File Open or File Exit.
Search actions	Used with search options. *TSearchAction* implements the common behavior for actions that display a modeless dialog where the user can enter a search string for searching an edit control.

All of the action objects are described under the action object names in the online reference Help. Refer to the Help for details on how they work.

Writing action components

You can also create your own predefined action classes. When you write your own action classes, you can build in the ability to execute on certain target classes of object. Then, you can use your custom actions in the same way you use pre-defined action classes. That is, when the action can recognize and apply itself to a target class, you can simply assign the action to a client control, and it acts on the target with no need to write an event handler.

Component writers can use the classes in the QStdActns and DBActns units as examples for deriving their own action classes to implement behaviors specific to certain controls or components. The base classes for these specialized actions (*TEditAction*, *TWindowAction*, and so on) generally override *HandlesTarget, UpdateTarget,* and other methods to limit the target for the action to a specific class of objects. The descendant classes typically override *ExecuteTarget* to perform a specialized task. These methods are described here:

Method	Description
HandlesTarget	Called automatically when the user invokes an object (such as a toolbutton or menu item) that is linked to the action. The *HandlesTarget* method lets the action object indicate whether it is appropriate to execute at this time with the object specified by the *Target* parameter as a "target". See "How actions find their targets" on page 6-17 for details.
UpdateTarget	Called automatically when the application is idle so that actions can update themselves according to current conditions. Use in place of *OnUpdateAction*. See "Updating actions" on page 6-17 for details.
ExecuteTarget	Called automatically when the action fires in response to a user action in place of *OnExecute* (for example, when the user selects a menu item or presses a tool button that is linked to this action). See "What happens when an action fires" on page 6-15 for details.

Registering actions

When you write your own actions, you can register actions to enable them to appear in the Action List editor. You register and unregister actions by using the global routines in the Actnlist unit:

```
procedure RegisterActions(const CategoryName: string; const AClasses: array of
TBasicActionClass; Resource: TComponentClass);

procedure UnRegisterActions(const AClasses: array of TBasicActionClass);
```

When you call *RegisterActions*, the actions you register appear in the Action List editor for use by your applications. You can supply a category name to organize your actions, as well as a *Resource* parameter that lets you supply default property values.

For example, the following code registers the standard actions with the IDE:

```
{ Standard action registration }

RegisterActions('', [TAction], nil);

RegisterActions('Edit', [TEditCut, TEditCopy, TEditPaste], TStandardActions);

RegisterActions('Window', [TWindowClose, TWindowCascade, TWindowTileHorizontal,
TWindowTileVertical, TWindowMinimizeAll, TWindowArrange], TStandardActions);
```

When you call *UnRegisterActions*, the actions no longer appear in the Action List editor.

Creating and managing menus

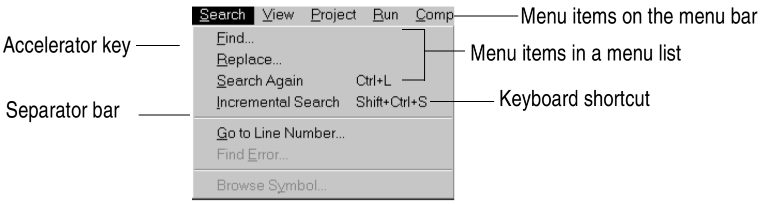
Menus provide an easy way for your users to execute logically grouped commands. The Menu Designer enables you to easily add a menu—either predesigned or custom tailored—to your form. You add a menu component to the form, open the Menu Designer, and type menu items directly into the Menu Designer window. You can add or delete menu items, or drag and drop them to rearrange them during design time.

You don't even need to run your program to see the results—your design is immediately visible in the form, appearing just as it will during runtime. Your code can also change menus at runtime, to provide more information or options to the user.

This chapter explains how to use the Menu Designer to design menu bars and pop-up (local) menus. It discusses the following ways to work with menus at design time and runtime:

- Opening the Menu Designer
- Building menus
- Editing menu items in the Object Inspector
- Using the Menu Designer context menu
- Using menu templates
- Saving a menu as a template
- Adding images to menu items

Figure 6.2 Menu terminology

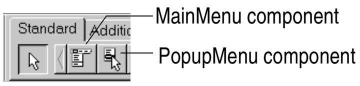


For information about hooking up menu items to the code that executes when they are selected, see “Associating menu events with event handlers” on page 2-26.

Designing menus

You design menus using the Menu Designer. Before you can start using the Menu Designer, first add either a *TMainMenu* or *TPopupMenu* component to your form. Both menu components are located on the Standard page of the Component palette.

Figure 6.3 MainMenu and PopupMenu components



A MainMenu component creates a menu that’s attached to the form’s title bar. A PopupMenu component creates a menu that appears when the user right-clicks in the form. Pop-up menus do not have a menu bar.

To open the Menu Designer, select a menu component on the form, and then either

- Double-click the menu component.

  or

- From the Properties page of the Object Inspector, select the *Items* property, and then either double-click [Menu] in the Value column, or click the ellipsis (...) button.

The Menu Designer is displayed with the first (blank) menu item highlighted in the Designer, and the *Caption* property selected in the Object Inspector.

Figure 6.4 Menu Designer for a pop-up menu

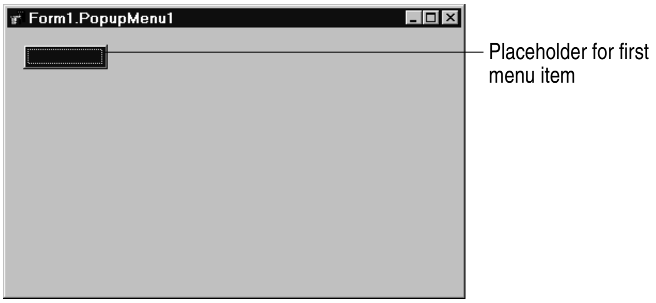
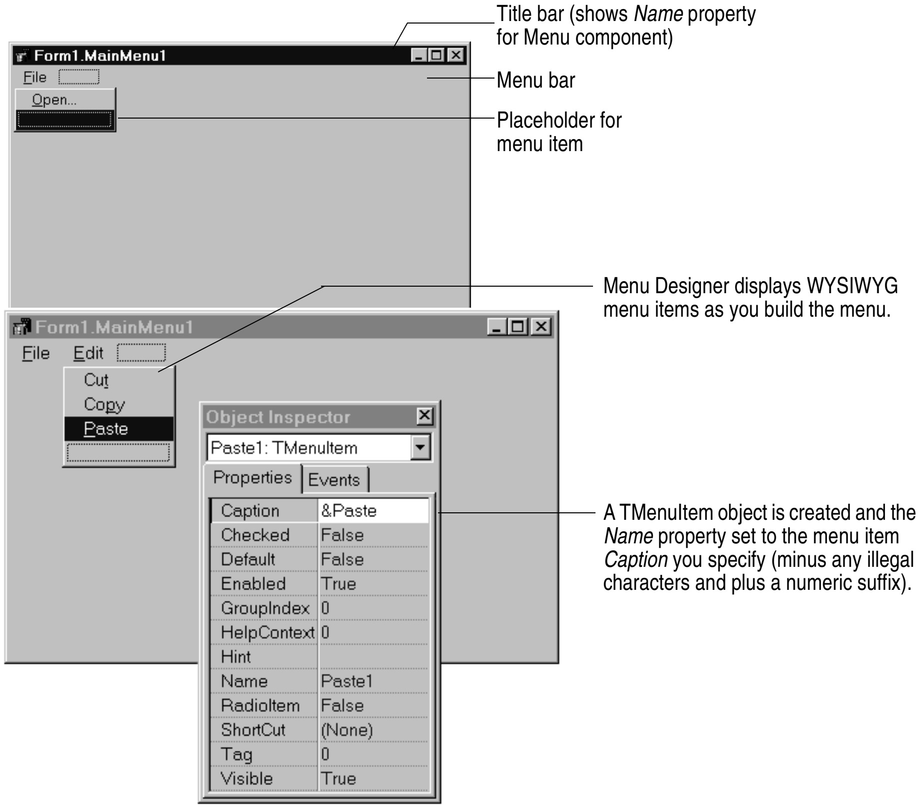


Figure 6.5 Menu Designer for a main menu


Form1.MainMenu1
File
Open...
Title bar (shows *Name* property for Menu component)
Menu bar
Placeholder for menu item
Menu Designer displays WYSIWYG menu items as you build the menu.
Form1.MainMenu1
File
Edit
Cut
Copy
Paste
Object Inspector
Paste1: TMenuItem
Properties
Events
A TMenuItem object is created and the *Name* property set to the menu item *Caption* you specify (minus any illegal characters and plus a numeric suffix).

Caption	&Paste
Checked	False
Default	False
Enabled	True
GroupIndex	0
HelpContext	0
Hint	
Name	Paste1
RadioItem	False
ShortCut	(None)
Tag	0
Visible	True

Building menus

You add a menu component to your form, or forms, for every menu you want to include in your application. You can build each menu structure entirely from scratch, or you can start from one of the predesigned menu templates.

This section discusses the basics of creating a menu at design time. For more information about menu templates, see "Using menu templates" on page 6-24.

Naming menus

As with all components, when you add a menu component to the form, Kylix gives it a default name; for example, *MainMenu1*. You can give the menu a more meaningful name that follows Object Pascal naming conventions.

Kylix adds the menu name to the form's type declaration, and the menu name then appears in the Component list.

Naming the menu items

In contrast to the menu component itself, you need to explicitly name menu items as you add them to the form. You can do this in one of two ways:

- Directly type the value for the *Name* property.
- Type the value for the *Caption* property first, and let Kylix derive the *Name* property from the caption.

  For example, if you give a menu item a *Caption* property value of *File*, Kylix assigns the menu item a *Name* property of *File1*. If you fill in the *Name* property before filling in the *Caption* property, Kylix leaves the *Caption* property blank until you type a value.

Note If you enter characters in the *Caption* property that are not valid for Object Pascal identifiers, Kylix modifies the *Name* property accordingly. For example, if you want the caption to start with a number, Kylix precedes the number with a character to derive the *Name* property.

The following table demonstrates some examples of this, assuming all menu items shown appear in the same menu bar.

Table 6.2 Sample captions and their derived names

Component caption	Derived name	Explanation
&File	File1	Removes ampersand
&File (2nd occurrence)	File2	Numerically orders duplicate items
1234	N12341	Adds a preceding letter and numerical order
1234 (2nd occurrence)	N12342	Adds a number to disambiguate the derived name
$@@@#	N1	Removes all non-standard characters, adding preceding letter and numerical order
- (hyphen)	N2	Numerical ordering of second occurrence of caption with no standard characters

As with the menu component, Kylix adds any menu item names to the form's type declaration, and those names then appear in the component list.

Adding, inserting, and deleting menu items

The following procedures describe how to perform the basic tasks involved in building your menu structure. Each procedure assumes you have the Menu Designer window open.

To add menu items at design time,

1. Select the position where you want to create the menu item.

   If you've just opened the Menu Designer, the first position on the menu bar is already selected.

2. Begin typing to enter the caption. Or enter the *Name* property first by specifically placing your cursor in the Object Inspector and entering a value. In this case, you then need to reselect the *Caption* property and enter a value.

3. Press *Enter*.

   The next placeholder for a menu item is selected.

   If you entered the *Caption* property first, use the arrow keys to return to the menu item you just entered. You'll see that Kylix has filled in the *Name* property based on the value you entered for the caption. (See "Naming the menu items" on page 6-18.)

4. Continue entering values for the *Name* and *Caption* properties for each new item you want to create, or press *Esc* to return to the menu bar.

   Use the arrow keys to move from the menu bar into the menu, and to then move between items in the list; press *Enter* to complete an action. To return to the menu bar, press *Esc*.

To insert a new, blank menu item,

1. Place the cursor on a menu item.
2. Press *Ins*.

   Menu items are inserted to the left of the selected item on the menu bar, and above the selected item in the menu list.

To delete a menu item or command,

1. Place the cursor on the menu item you want to delete.
2. Press *Del*.

Note You cannot delete the default placeholder that appears below the item last entered in a menu list, or next to the last item on the menu bar. This placeholder does not appear in your menu at runtime.

Adding separator bars

- Separator bars insert a line between menu items. You can use separator bars to indicate groupings within the menu list, or simply to provide a visual break in a list.

To make the menu item a separator bar, type a hyphen (-) for the caption.

Specifying accelerator keys and keyboard shortcuts

Accelerator keys enable the user to access a menu command from the keyboard by pressing *Alt*+ the appropriate letter, indicated in your code by the preceding ampersand. The letter after the ampersand appears underlined in the menu.

Kylix automatically checks for duplicate accelerators and adjusts them at runtime. This ensures that menus built dynamically at runtime contain no duplicate accelerators and that all menu items have an accelerator.

To specify an accelerator,

- Add an ampersand in front of the appropriate letter.

  For example, to add a Save menu command with the *S* as an accelerator key, type `&Save`.

Keyboard shortcuts enable the user to perform the action without using the menu directly, by typing in the shortcut key combination.

To specify a keyboard shortcut,

- Use the Object Inspector to enter a value for the *ShortCut* property, or select a key combination from the drop-down list.

  This list is only a subset of the valid combinations you can type in.

When you add a shortcut, it appears next to the menu item caption.

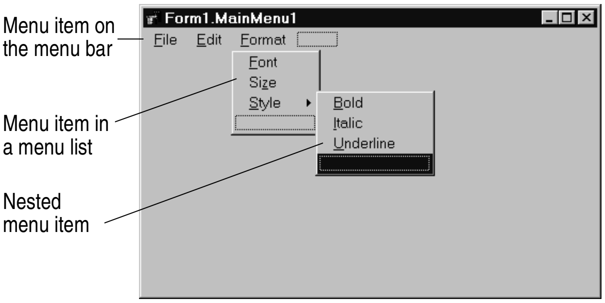
Caution Keyboard shortcuts, unlike accelerator keys, are not checked automatically for duplicates. You must ensure uniqueness yourself.

Creating submenus

Many application menus contain drop-down lists that appear next to a menu item to provide additional, related commands. Such lists are indicated by an arrow to the right of the menu item. Kylix supports as many levels of such submenus as you want to build into your menu.

Organizing your menu structure this way can save vertical screen space. However, for optimal design purposes you probably want to use no more than two or three menu levels in your interface design. (For pop-up menus, you might want to use only one submenu, if any.)

Figure 6.6 Nested menu structures



To create a submenu,

1 Select the menu item under which you want to create a submenu.
2 Press *Ctrl→* to create the first placeholder, or right-click and choose Create Submenu.
3 Type a name for the submenu item, or drag an existing menu item into this placeholder.
4 Press *Enter*, or ↓, to create the next placeholder.
5 Repeat steps 3 and 4 for each item you want to create in the submenu.
6 Press *Esc* to return to the previous menu level.

Creating submenus by demoting existing menus

You can create a submenu by inserting a menu item from the menu bar (or a menu template) between menu items in a list. When you move a menu into an existing menu structure, all its associated items move with it, creating a fully intact submenu. This pertains to submenus as well—moving a menu item into an existing submenu just creates one more level of nesting.

Moving menu items

During design time, you can move menu items simply by dragging and dropping. You can move menu items along the menu bar, or to a different place in the menu list, or into a different menu entirely.

The only exception to this is hierarchical: you cannot demote a menu item from the menu bar into its own menu; nor can you move a menu item into its own submenu. However, you can move any item into a *different* menu, no matter what its original position is.

While you are dragging, the cursor changes shape to indicate whether you can release the menu item at the new location. When you move a menu item, any items beneath it move as well.

To move a menu item along the menu bar,

1 Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new location.

2 Release the mouse button to drop the menu item at the new location.

To move a menu item into a menu list,

1 Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new menu.

   This causes the menu to open, enabling you to drag the item to its new location.

2 Drag the menu item into the list, releasing the mouse button to drop the menu item at the new location.

Adding images to menu items

Images can help users navigate in menus by matching glyphs and images to menu item action, similar to toolbar images. You can add single bitmaps to menu items, or you can organize images for your application into an image list and add them to a menu from the image list. If you're using several bitmaps of the same size in your application, it's useful to put them into an image list.

To add a single image to a menu or menu item, set its *Bitmap* property to reference the name of the bitmap to use on the menu or menu item.

To add images to menu items using an image list:

1 Drop a *TMainMenu* or *TPopupMenu* object on a form.

2 Drop a *TImageList* object on the form.

3 Open the ImageList editor by double clicking on the *TImageList* object.

4 Click Add to select the bitmap or bitmap group you want to use in the menu. Click OK.

5 Set the *TMainMenu* or *TPopupMenu* object's *Images* property to the ImageList you just created.

6 Create your menu items and submenu items as described previously.

7 Select the menu item you want to have an image in the Object Inspector and set the *ImageIndex* property to the corresponding number of the image in the *ImageList* (the default value for *ImageIndex* is -1, which doesn't display an image).

Note Use images that are 16 by 16 pixels for proper display in the menu. Although you can use other sizes for the menu images, alignment and consistency problems may result when using images greater than or smaller than 16 by 16 pixels.

Viewing the menu

You can view your menu in the form at design time without first running your program code. (Pop-up menu components are visible in the form at design time, but the pop-up menus themselves are not. Use the Menu Designer to view a pop-up menu at design time.)

To view the menu,

1 If the form is visible, click the form, or from the View menu, choose the form whose menu you want to view.

2 If the form has more than one menu, select the menu you want to view from the form's *Menu* property drop-down list.

The menu appears in the form exactly as it will when you run the program.

Editing menu items in the Object Inspector

This section has discussed how to set several properties for menu items—for example, the *Name* and *Caption* properties—by using the Menu Designer.

The section has also described how to set menu item properties, such as the *ShortCut* property, directly in the Object Inspector, just as you would for any component selected in the form.

When you edit a menu item by using the Menu Designer, its properties are still displayed in the Object Inspector. You can switch focus to the Object Inspector and continue editing the menu item properties there. Or you can select the menu item from the Component list in the Object Inspector and edit its properties without ever opening the Menu Designer.

To close the Menu Designer window and continue editing menu items,

1 Switch focus from the Menu Designer window to the Object Inspector by clicking the properties page of the Object Inspector.

2 Close the Menu Designer as you normally would.

The focus remains in the Object Inspector, where you can continue editing properties for the selected menu item. To edit another menu item, select it from the Component list.

Using the Menu Designer context menu

The Menu Designer context menu provides quick access to the most common Menu Designer commands, and to the menu template options. (For more information about menu templates, refer to "Using menu templates" on page 6-24.)

To display the context menu, right-click the Menu Designer window, or press *Alt+F10* when the cursor is in the Menu Designer window.

Commands on the context menu

The following table summarizes the commands on the Menu Designer context menu.

Table 6.3 Menu Designer context menu commands

Menu command	Action
Insert	Inserts a placeholder above or to the left of the cursor.
Delete	Deletes the selected menu item (and all its sub-items, if any).
Create Submenu	Creates a placeholder at a nested level and adds an arrow to the right of the selected menu item.

Table 6.3 Menu Designer context menu commands (continued)

Menu command	Action
Select Menu	Opens a list of menus in the current form. Double-clicking a menu name opens the designer window for the menu.
Save As Template	Opens the Save Template dialog box, where you can save a menu for future reuse.
Insert From Template	Opens the Insert Template dialog box, where you can select a template to reuse.
Delete Templates	Opens the Delete Templates dialog box, where you can choose to delete any existing templates.
Insert From Resource	Opens the Insert Menu from Resource file dialog box, where you can choose an .mnu file to open in the current form.

Switching between menus at design time

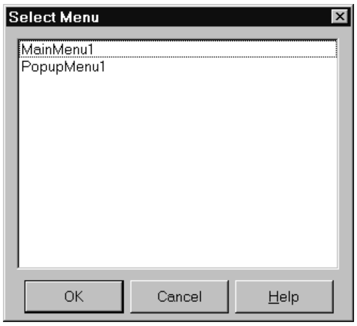
If you're designing several menus for your form, you can use the Menu Designer context menu or the Object Inspector to easily select and move among them.

To use the context menu to switch between menus in a form,

1 Right-click in the Menu Designer and choose Select Menu.

The Select Menu dialog box appears.

Figure 6.7 Select Menu dialog box



This dialog box lists all the menus associated with the form whose menu is currently open in the Menu Designer.

2 From the list in the Select Menu dialog box, choose the menu you want to view or edit.

To use the Object Inspector to switch between menus in a form,

1 Give focus to the form whose menus you want to choose from.

2 From the Component list, select the menu you want to edit.

3 On the Properties page of the Object Inspector, select the *Items* property for this menu, and then either click the ellipsis button, or double-click [Menu].

Using menu templates

Kylix provides several predesigned menus, or menu templates, that contain frequently used commands. You can use these menus in your applications without modifying them (except to write code), or you can use them as a starting point, customizing them as you would a menu you originally designed yourself. Menu templates do not contain any event handler code.

The menu templates shipped with Kylix are stored in delphi60dmt in the .borland directory in a default installation.

You can also save as a template any menu that you design using the Menu Designer. After saving a menu as a template, you can use it as you would any predesigned menu. If you decide you no longer want a particular menu template, you can delete it from the list.

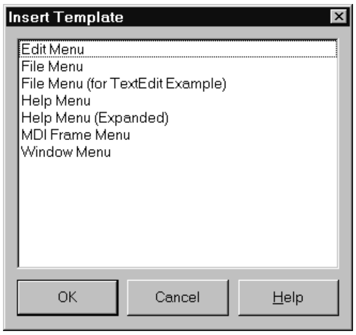
To add a menu template to your application,

1 Right-click the Menu Designer and choose Insert From Template.

(If there are no templates, the Insert From Template option appears dimmed in the context menu.)

The Insert Template dialog box opens, displaying a list of available menu templates.

Figure 6.8 Sample Insert Template dialog box for menus



2 Select the menu template you want to insert, then press *Enter* or choose OK.

This inserts the menu into your form at the cursor's location. For example, if your cursor is on a menu item in a list, the menu template is inserted above the selected item. If your cursor is on the menu bar, the menu template is inserted to the left of the cursor.

To delete a menu template,

1 Right-click the Menu Designer and choose Delete Templates.

(If there are no templates, the Delete Templates option appears dimmed in the context menu.)

The Delete Templates dialog box opens, displaying a list of available templates.

2 Select the menu template you want to delete, and press *Del*.

Kylix deletes the template from the templates list and from your hard disk.

Saving a menu as a template

Any menu you design can be saved as a template so you can use it again. You can use menu templates to provide a consistent look to your applications, or use them as a starting point which you then further customize.

The menu templates you save are stored in the .borland directory in the delphi60dmt file.

To save a menu as a template,

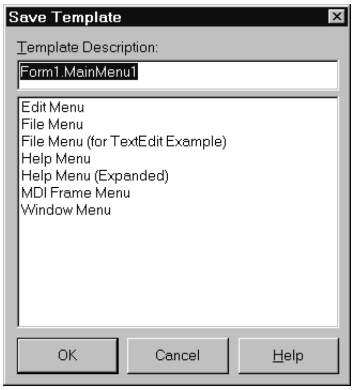
1 Design the menu you want to be able to reuse.

This menu can contain as many items, commands, and submenus as you like; everything in the active Menu Designer window will be saved as one reusable menu.

2 Right-click in the Menu Designer and choose Save As Template.

The Save Template dialog box appears.

Figure 6.9 Save Template dialog box for menus



3 In the Template Description edit box, type a brief description for this menu, and then choose OK.

The Save Template dialog box closes, saving your menu design and returning you to the Menu Designer window.

Note The description you enter is displayed only in the Save Template, Insert Template, and Delete Templates dialog boxes. It is not related to the *Name* or *Caption* property for the menu.

Naming conventions for template menu items and event handlers

When you save a menu as a template, Kylix does not save its *Name* property, since every menu must have a unique name within the scope of its owner (the form). However, when you insert the menu as a template into a new form by using the Menu Designer, Kylix then generates new names for it and all of its items.

For example, suppose you save a File menu as a template. In the original menu, you name it *MyFile*. If you insert it as a template into a new menu, Kylix names it *File1*. If you insert it into a menu with an existing menu item named *File1*, Kylix names it *File2*.

Kylix also does not save any *OnClick* event handlers associated with a menu saved as a template, since there is no way to test whether the code would be applicable in the new form. When you generate a new event handler for the menu template item, Kylix still generates the event handler name. You can associate items in the menu template with existing *OnClick* event handlers in the form.

Manipulating menu items at runtime

Sometimes you want to add menu items to an existing menu structure while the application is running, to provide more information or options to the user. You can insert a menu item by using the menu item's *Add* or *Insert* method, or you can alternately hide and show the items in a menu by changing their *Visible* property. The *Visible* property determines whether the menu item is displayed in the menu. To dim a menu item without hiding it, use the *Enabled* property.

For examples that use the menu item's *Visible* and *Enabled* properties, see "Disabling menu items" on page 7-4.

Designing toolbars

A *toolbar* is a panel, usually across the top of a form (under the menu bar), that holds buttons and other controls. You can put controls of any sort on a toolbar. In addition to buttons, you may want to put use color grids, scroll bars, labels, and so on.

There are several ways to add a toolbar to a form:

- Place a panel (*TPanel*) on the form and add controls (typically speed buttons) to it.
- Use a toolbar component (*TToolBar*) instead of *TPanel*, and add controls to it. *TToolBar* manages buttons and other controls, arranging them in rows and automatically adjusting their sizes and positions. If you use tool button (*TToolButton*) controls on the toolbar, *TToolBar* makes it easy to group the buttons functionally and provides other display options.

How you implement your toolbar depends on your application. The advantage of using the Panel component is that you have total control over the look and feel of the toolbar.

By using the toolbar component, you are ensuring that your application has a consistent look and feel. If these operating system controls change in the future, your application could change as well.

The following sections describe how to

- Add a toolbar and corresponding speed button controls using the panel component
- Add a toolbar and corresponding tool button controls using the Toolbar component
- Respond to clicks
- Add hidden toolbars
- Hide and show toolbars

Adding a toolbar using a panel component

To add a toolbar to a form using the panel component,

1 Add a panel component to the form (from the Standard page of the Component palette).

2 Set the panel's *Align* property to *alTop*. When aligned to the top of the form, the panel maintains its height, but matches its width to the full width of the form's client area, even if the window changes size.

3 Add speed buttons or other controls to the panel.

Speed buttons are designed to work on toolbar panels. A speed button usually has no caption, only a small graphic (called a *glyph*), which represents the button's function.

Speed buttons have three possible modes of operation. They can

- Act like regular pushbuttons
- Toggle on and off when clicked
- Act like a set of radio buttons

To implement speed buttons on toolbars, do the following:

- Add a speed button to a toolbar panel
- Assign a speed button's glyph
- Set the initial condition of a speed button
- Create a group of speed buttons
- Allow toggle buttons

Adding a speed button to a panel

To add a speed button to a toolbar panel, place the speed button component (from the Additional page of the Component palette) on the panel.

The panel, rather than the form, "owns" the speed button, so moving or hiding the panel also moves or hides the speed button.

The default height of the panel is 41, and the default height of speed buttons is 25. If you set the *Top* property of each button to 8, they'll be vertically centered. The default grid setting snaps the speed button to that vertical position for you.

Assigning a speed button's glyph

Each speed button needs a graphic image called a *glyph* to indicate to the user what the button does. If you supply the speed button only one image, the button manipulates that image to indicate whether the button is pressed, unpressed, selected, or disabled. You can also supply separate, specific images for each state if you prefer.

You normally assign glyphs to speed buttons at design time, although you can assign different glyphs at runtime.

To assign a glyph to a speed button at design time,

1 Select the speed button.

2 In the Object Inspector, select the *Glyph* property.

3 Double-click the Value column beside *Glyph* to open the Picture Editor and select the desired bitmap.

Setting the initial condition of a speed button

Speed buttons use their appearance to give the user clues as to their state and purpose. Because they have no caption, it's important that you use the right visual cues to assist users.

Table 6.4 lists some actions you can set to change a speed button's appearance:

Table 6.4 Setting speed buttons' appearance

To make a speed button:	Set the toolbar's:
Appear pressed	*GroupIndex* property to a value other than zero and its *Down* property to *True*.
Appear disabled	*Enabled* property to *False*.
Have a left margin	*Indent* property to a value greater than 0.

If your application has a default drawing tool, ensure that its button on the toolbar is pressed when the application starts. To do so, set its *GroupIndex* property to a value other than zero and its *Down* property to *True*.

Creating a group of speed buttons

A series of speed buttons often represents a set of mutually exclusive choices. In that case, you need to associate the buttons into a group, so that clicking any button in the group causes the others in the group to pop up.

To associate any number of speed buttons into a group, assign the same number to each speed button's *GroupIndex* property.

The easiest way to do this is to select all the buttons you want in the group, and, with the whole group selected, set *GroupIndex* to a unique value.

Allowing toggle buttons

Sometimes you want to be able to click a button in a group that's already pressed and have it pop up, leaving no button in the group pressed. Such a button is called a *toggle*. Use *AllowAllUp* to create a grouped button that acts as a toggle: click it once, it's down; click it again, it pops up.

To make a grouped speed button a toggle, set its *AllowAllUp* property to *True*.

Setting *AllowAllUp* to *True* for any speed button in a group automatically sets the same property value for all buttons in the group. This enables the group to act as a normal group, with only one button pressed at a time, but also allows every button to be up at the same time.

Adding a toolbar using the toolbar component

The toolbar component (*TToolBar*) offers button management and display features that panel components do not. To add a toolbar to a form using the toolbar component,

1. Add a toolbar component to the form. The toolbar automatically aligns to the top of the form.
2. Add tool buttons or other controls to the bar.

Tool buttons are designed to work on toolbar components. Like speed buttons, tool buttons can

- Act like regular pushbuttons
- Toggle on and off when clicked
- Act like a set of radio buttons

To implement tool buttons on a toolbar, do the following:

- Add a tool button
- Assign images to tool buttons
- Set the tool buttons' appearance
- Create a group of tool buttons
- Allow toggled tool buttons

Adding a tool button

To add a tool button to a toolbar, right-click on the toolbar and choose New Button.

The toolbar "owns" the tool button, so moving or hiding the toolbar also moves or hides the button. In addition, all tool buttons on the toolbar automatically maintain the same height and width. You can drop other controls from the Component palette onto the toolbar, and they will automatically maintain a uniform height. Controls will also wrap around and start a new row when they do not fit horizontally on the toolbar.

Assigning images to tool buttons

Each tool button has an *ImageIndex* property that determines what image appears on it at runtime. If you supply the tool button only one image, the button manipulates that image to indicate whether the button is disabled. To assign images to tool buttons at design time,

1 Select the toolbar on which the buttons appear.

2 In the Object Inspector, assign a *TImageList* object to the toolbar's *Images* property. An image list is a collection of same-sized icons or bitmaps.

3 Select a tool button.

4 In the Object Inspector, assign an integer to the tool button's *ImageIndex* property that corresponds to the image in the image list that you want to assign to the button.

You can also specify separate images to appear on the tool buttons when they are disabled and when they are under the mouse pointer. To do so, assign separate image lists to the toolbar's *DisabledImages* and *HotImages* properties.

Setting tool button appearance and initial conditions

Table 6.5 lists some actions you can set to change a tool button's appearance:

Table 6.5 Setting tool buttons' appearance

To make a tool button:	Set the toolbar's:
Appear pressed	(on the *TToolButton*) *Style* property to *tbsCheck* and *Down* property to *True*.
Appear disabled	*Enabled* property to *False*.
Have a left margin	*Indent* property to a value greater than 0.
Appear to have "pop-up" borders, thus making the toolbar appear transparent	*Flat* property to *True*.

To force a new row of controls after a specific tool button, Select the tool button that you want to appear last in the row and set its *Wrap* property to *True*.

To turn off the auto-wrap feature of the toolbar, set the toolbar's *Wrapable* property to *False*.

Creating groups of tool buttons

To create a group of tool buttons, select the buttons you want to associate and set their *Style* property to *tbsCheck*; then set their *Grouped* property to *True*. Selecting a grouped tool button causes other buttons in the group to pop up, which is helpful to represent a set of mutually exclusive choices.

Any unbroken sequence of adjacent tool buttons with *Style* set to *tbsCheck* and *Grouped* set to *True* forms a single group. To break up a group of tool buttons, separate the buttons with any of the following:

- A tool button whose *Grouped* property is *False*.

- A tool button whose *Style* property is not set to *tbsCheck*. To create spaces or dividers on the toolbar, add a tool button whose *Style* is *tbsSeparator* or *tbsDivider*.
- Another control besides a tool button.

Allowing toggled tool buttons

Use *AllowAllUp* to create a grouped tool button that acts as a toggle: click it once, it is down; click it again, it pops up. To make a grouped tool button a toggle, set its *AllowAllUp* property to *True*.

As with speed buttons, setting *AllowAllUp* to *True* for any tool button in a group automatically sets the same property value for all buttons in the group.

Responding to clicks

When the user clicks a control, such as a button on a toolbar, the application generates an *OnClick* event which you can respond to with an event handler. Since *OnClick* is the default event for buttons, you can generate a skeleton handler for the event by double-clicking the button at design time. For more information, see "Working with events and event handlers" on page 2-24 and "Generating a handler for a component's default event" on page 2-25.

Assigning a menu to a tool button

If you are using a toolbar (*TToolBar*) with tool buttons (*TToolButton*), you can associate menu with a specific button:

1 Select the tool button.
2 In the Object Inspector, assign a pop-up menu (*TPopupMenu*) to the tool button's *DropDownMenu* property.

If the menu's *AutoPopup* property is set to *True*, it will appear automatically when the button is pressed.

Adding hidden toolbars

Toolbars do not have to be visible all the time. In fact, it is often convenient to have a number of toolbars available, but show them only when the user wants to use them. Often you create a form that has several toolbars, but hide some or all of them.

To create a hidden toolbar,

1 Add a toolbar or panel component to the form.
2 Set the component's *Visible* property to *False*.

Although the toolbar remains visible at design time so you can modify it, it remains hidden at runtime until the application specifically makes it visible.

Hiding and showing toolbars

Often, you want an application to have multiple toolbars, but you do not want to clutter the form with them all at once. Or you may want to let users decide whether to display toolbars. As with all components, toolbars can be shown or hidden at runtime as needed.

To hide or show a toolbar at runtime, set its *Visible* property to *False* or *True*, respectively. Usually you do this in response to particular user events or changes in the operating mode of the application. To do this, you typically have a close button on each toolbar. When the user clicks that button, the application hides the corresponding toolbar.

You can also provide a means of toggling the toolbar. In the following example, a toolbar of pens is toggled from a button on the main toolbar. Since each click presses or releases the button, an *OnClick* event handler can show or hide the Pen toolbar depending on whether the button is up or down.

```
procedure TForm1.PenButtonClick(Sender: TObject);
begin
  PenBar.Visible := PenButton.Down;
end;
```

Chapter

7

Working with controls

Controls are visual components that the user of your application can interact with at runtime such as scrollbars, buttons, text boxes, list boxes, and so on. Generally, controls are objects that descend from *TControl* in the object hierarchy. This chapter describes some of the commonly used controls.

Working with text in controls

The following sections explain how to use various features of edit and memo controls. Some of these features work with edit controls as well.

- Setting text alignment
- Adding scrollbars at runtime
- Adding the clipboard object
- Selecting text
- Selecting all text
- Cutting, copying, and pasting text
- Deleting selected text
- Disabling menu items
- Providing a pop-up menu
- Handling the OnPopup event

Setting text alignment

In a memo component, text can be left- or right-aligned, or centered. To change text alignment, set the edit component's *Alignment* property. Alignment takes effect only if the *WordWrap* property is *True*; if word wrapping is turned off, there is no margin to align to. *WordWrap* turns wordwrapping on and off. When on, the *WrapMode*, *WrapBreak*, and *WrapAtValue* properties allow fine-grain control on how the wrapping is done.

You can also use the *HMargin* property to adjust the left and right margins in a memo control.

For example, the following code implements an OnClick event handler for the Character|Left menu item, then attaches the same event handler to both the Right and Center menu items on the Character menu.

```
procedure TForm1.AlignClick(Sender: TObject);
begin
  Left1.Checked := False;  { clear all three checks }
  Right1.Checked := False;
  Center1.Checked := False;
  with Sender as TMenuItem do Checked := True;  { check the item clicked }
  with Editor do  { then set Alignment to match }
    if Left1.Checked then
      Alignment := taLeftJustify
    else if Right1.Checked then
      Alignment := taRightJustify
    else if Center1.Checked then
      Alignment := taCenter;
end;
```

Adding scroll bars at runtime

Editing and memo components can contain horizontal or vertical scroll bars, or both, as needed. When word-wrapping is enabled, the component needs only a vertical scroll bar. If the user turns off word-wrapping, the component might also need a horizontal scroll bar, since text is not limited by the right side of the editor.

To add scroll bars at runtime,

1 Determine whether the text might exceed the right margin. In most cases, this means checking whether word wrapping is enabled. You might also check whether any text lines actually exceed the width of the control.
2 Set the edit or memo component's *ScrollBars* property to include or exclude scroll bars.

```
procedure TForm1.WordWrap1Click(Sender: TObject);
begin
  with Editor do
  begin
    WordWrap := not WordWrap;  { toggle word-wrapping }
    if WordWrap then
      ScrollBars := ssVertical  { wrapped requires only vertical }
    else
      ScrollBars := ssBoth;  { unwrapped might need both }
      WordWrap1.Checked := WordWrap;  { check menu item to match property }
  end;
end;
```

Note that the memo always shows scroll bars if they are enabled.

Adding a clipboard to an application

Most text-handling applications provide users with a way to move selected text between documents, including documents in different applications. The *TClipboard* object in Kylix provides a clipboard and includes methods for cutting, copying, and pasting text (and other formats, including graphics). The *TClipboard* object is declared in the *QClipbrd* unit.

To add a *TClipboard* object to an application,

1. Select the unit that will use the clipboard.
2. Search for the **`implementation`** reserved word.
3. Add *QClipbrd* to the **`uses`** clause below **`implementation`**.
   - If there is already a **`uses`** clause in the **`implementation`** part, add *QClipbrd* to the end of it.
   - If there is not already a **`uses`** clause, add one that says

     ```
     uses QClipbrd;
     ```

Selecting text

Before you can send any text to the clipboard, that text must be selected. Highlighting of selected text is built into the edit components. When the user selects text, it appears highlighted.

Table 7.1 lists properties commonly used to handle selected text.

Table 7.1 Properties of selected text

Property	Description
SelText	Contains a string representing the selected text in the component.
SelLength	Contains the length of a selected string.
SelStart	Contains the starting position of a string.

For example, the following *OnFind* event handler searches a Memo component for the text specified in the *FindText* property of a find dialog component. If found, the first occurrence of the text in Memo1 is selected.

```
procedure TForm1.FindDialog1Find(Sender: TObject);
var
  I, J, PosReturn, SkipChars: Integer;
begin
  for I := 0 to Memo1.Lines.Count do
  begin
    PosReturn := Pos(FindDialog1.FindText,Memo1.Lines[I]);
    if PosReturn <> 0 then {found!}
    begin
      Skipchars := 0;
      for J := 0 to I - 1 do
        Skipchars := Skipchars + Length(Memo1.Lines[J]);
```

```
          SkipChars := SkipChars + (I*2);
          SkipChars := SkipChars + PosReturn - 1;
          Memo1.SetFocus;
          Memo1.SelStart := SkipChars;
          Memo1.SelLength := Length(FindDialog1.FindText);
          Break;
        end;
      end;
    end;
```

Selecting all text

The *SelectAll* method selects the entire contents of the memo component. This is especially useful when the component's contents exceed the visible area of the component. In most other cases, users select text with either keystrokes or mouse dragging.

To select the entire contents of a memo control, call the control's *SelectAll* method.

For example,

```
procedure TMainForm.SelectAll(Sender: TObject);
begin
  Memo1.SelectAll;  { select all text in memo }
end;
```

Cutting, copying, and pasting text

Applications that use the *Clipbrd* unit can cut, copy, and paste text, graphics, and objects through the clipboard. The edit components that encapsulate the standard text-handling controls all have methods built into them for interacting with the clipboard. (See "Using the clipboard with graphics" on page 8-20 for information on using the clipboard with graphics.)

To cut, copy, or paste text with the clipboard, call the edit component's *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods, respectively.

For example, the following code attaches event handlers to the *OnClick* events of the Edit|Cut, Edit|Copy, and Edit|Paste commands, respectively:

```
procedure TForm1.CutToClipboard(Sender: TObject);
begin
  Editor.CutToClipboard;
end;
procedure TForm1.CopyToClipboard(Sender: TObject);
begin
  Editor.CopyToClipboard;
end;
procedure TForm1.PasteFromClipboard(Sender: TObject);
begin
  Editor.PasteFromClipboard;
end;
```

Deleting selected text

You can delete the selected text in an edit component without cutting it to the clipboard. To do so, call the *ClearSelection* method. For example, if you have a Delete item on the Edit menu, your code could look like this:

```
procedure TForm1.Delete(Sender: TObject);
begin
  Memo1.ClearSelection;
end;
```

Disabling menu items

It is often useful to disable menu commands without removing them from the menu. For example, in a text editor, if there is no text currently selected, the Cut, Copy, and Delete commands are inapplicable. An appropriate time to enable or disable menu items is when the user selects the menu. To disable a menu item, set its *Enabled* property to *False*.

In the following example, an event handler is attached to the *OnClick* event for the Edit item on a child form's menu bar. It sets *Enabled* for the Cut, Copy, and Delete menu items on the Edit menu based on whether *Memo1* has selected text. The Paste command is enabled or disabled based on whether any text exists on the clipboard.

```
procedure TForm1.Edit1Click(Sender: TObject);
var
  HasSelection: Boolean;  { declare a temporary variable }
begin
  Paste1.Enabled := Clipboard.Provides('text'); {Enable/disable paste menu item}
  ...
end;
```

The Provides method of the clipboard returns a Boolean value based on whether the clipboard contains objects, text, or images of a particular format. (In this case, the text is generic. You can specify the type of text using a subtype such as text/plain for plain text or text/html for html.) By calling Provides with the parameter text, you can determine whether the clipboard contains any text, and you can enable or disable the Paste item as appropriate.

Chapter 8, "Working with graphics" provides more information about using the clipboard with graphics.

Providing a pop-up menu

Pop-up, or local, menus are a common ease-of-use feature for any application. They enable users to minimize mouse movement by clicking the right mouse button in the application workspace to access a list of frequently used commands.

In a text editor application, for example, you can add a pop-up menu that repeats the Cut, Copy, and Paste editing commands. These pop-up menu items can use the same event handlers as the corresponding items on the Edit menu. You don't need to create accelerator or shortcut keys for pop-up menus because the corresponding regular menu items generally already have shortcuts.

A form's *PopupMenu* property specifies what pop-up menu to display when a user right-clicks any item on the form. Individual controls also have *PopupMenu* properties that can override the form's property, allowing customized menus for particular controls.

To add a pop-up menu to a form,

1 Place a pop-up menu component on the form.
2 Use the Menu Designer to define the items for the pop-up menu.
3 Set the *PopupMenu* property of the form or control that displays the menu to the name of the pop-up menu component.
4 Attach handlers to the *OnClick* events of the pop-up menu items.

Handling the OnPopup event

You may want to adjust pop-up menu items before displaying the menu, just as you may want to enable or disable items on a regular menu. With a regular menu, you can handle the *OnClick* event for the item at the top of the menu, as described in "Disabling menu items" on page 7-5.

With a pop-up menu, however, there is no top-level menu bar, so to prepare the pop-up menu commands, you handle the event in the menu component itself. The pop-up menu component provides an event just for this purpose, called *OnPopup*.

To adjust menu items on a pop-up menu before displaying them,

1 Select the pop-up menu component.
2 Attach an event handler to its *OnPopup* event.
3 Write code in the event handler to enable, disable, hide, or show menu items.

In the following code, the *Edit1Click* event handler described previously in "Disabling menu items" on page 7-5 is attached to the pop-up menu component's *OnPopup* event. A line of code is added to *Edit1Click* for each item in the pop-up menu.

```
procedure TForm1.Edit1Click(Sender: TObject);
var
  HasSelection: Boolean;
begin
  Paste1.Enabled := Clipboard.Provides('text');
  Paste2.Enabled := Paste1.Enabled;{Add this line}
  HasSelection := Editor.SelLength <> 0;
  Cut1.Enabled := HasSelection;
  Cut2.Enabled := HasSelection;{Add this line}
  Copy1.Enabled := HasSelection;
  Copy2.Enabled := HasSelection;{Add this line}
  Delete1.Enabled := HasSelection;
end;
```

Adding graphics to controls

Several controls let you customize the way the control is rendered. These include list boxes, combo boxes, menus, headers, tab controls, list views, status bars, tree views, and toolbars. Instead of using standard methods of drawing the control or its items, the control's owner (generally, the form) draws them at runtime. The most common use for owner-draw controls is to provide graphics instead of, or in addition to, text for items. For information on using owner-draw to add images to menus, see "Adding images to menu items" on page 6-26.

TComboBox and *TListBox* are owner-draw controls that contain lists of items. Usually, those lists are lists of strings that are displayed as text, or lists of objects that contain strings that are displayed as text. You can associate an object with each item in the list to make it easy to use that object when drawing items.

In general, creating an owner-draw control in Kylix involves these steps:

1 Indicating that a control is owner-drawn
2 Adding graphical objects to a string list
3 Drawing owner-draw items

Indicating that a control is owner-drawn

To customize the drawing of a control, you must supply event handlers that render the control's image when it needs to be painted. Some controls receive these events automatically. For example, list views, tree views, and tool bars all receive events at various stages in the drawing process without your having to set any properties. These events have names such as OnCustomDraw or OnAdvancedCustomDraw.

Other controls, however, require you to set a property before they receive owner-draw events. List boxes, combo boxes, header controls, and status bars have a property called *Style*. *Style* determines whether the control uses the default drawing (called the "standard" style) or owner drawing. Grids use a property called *DefaultDrawing* to enable or disable the default drawing. Tab controls have a property called *OwnerDraw* that enables or disabled the default drawing.

List boxes and combo boxes have additional owner-draw styles, called *fixed* and *variable*, as Table 7.2 describes. Other controls are always fixed, although the size of the item that contains the text may vary, the size of each item is determined before drawing the control.

Table 7.2 Fixed vs. variable owner-draw styles

Owner-draw style	Meaning	Examples
Fixed	Each item is the same height, with that height determined by the *ItemHeight* property.	*lbOwnerDrawFixed, csOwnerDrawFixed*
Variable	Each item might have a different height, determined by the data at runtime.	*lbOwnerDrawVariable, csOwnerDrawVariable*

Adding graphical objects to a string list

Every string list has the ability to hold a list of objects in addition to its list of strings. You can also add graphical object of varying sizes to a string list.

For example, in a file manager application, you may want to add bitmaps indicating the type of drive along with the letter of the drive. To do that, you need to add the bitmap images to the application, then copy those images into the proper places in the string list as described in the following sections.

Note that you can also organize graphical objects using an image list by creating a *TImageList*. However, these images must all be the same size. See “Adding images to menu items” on page 6-26 for an example of setting up an image list.

Adding images to an application

An image control is a nonvisual control that contains a graphical image, such as a bitmap. You use image controls to display graphical images on a form. You can also use them to hold hidden images that you'll use in your application. For example, you can store bitmaps for owner-draw controls in hidden image controls, like this:

1. Add image controls to the main form.
2. Set their *Name* properties.
3. Set the *Visible* property for each image control to *False*.
4. Set the *Picture* property of each image selecting the desired bitmap.

The image controls are invisible when you run the application. The image is stored with the form so it doesn't have to be loaded from a file at runtime.

Sizing owner-draw items

Before giving your application the chance to draw each item in a variable owner-draw control, a measure-item event (*TMeasureItemEvent*) is generated. The measure-item event tells the application where the item appears on the control.

Kylix determines the size the item (generally, it is just large enough to display the item's text in the current font). Your application can handle the event and change the rectangle chosen. For example, if you plan to substitute a bitmap for the item's text, change the rectangle to the size of the bitmap. If you want a bitmap and text, adjust the rectangle so it is large enough for both.

To change the size of an owner-draw item, attach an event handler to the measure-item event in the owner-draw control. List boxes and combo boxes use *OnMeasureItem*.

The sizing event has two important parameters: the index number of the item and the height of that item. The height is variable: the application can make it either smaller or larger. The positions of subsequent items depend on the size of preceding items.

For example, in a variable owner-draw list box, if the application sets the height of the first item to five pixels, the second item starts at the sixth pixel down from the top, and so on. In list boxes and combo boxes, the only aspect of the item the application can alter is the height of the item. The width of the item is always the width of the control.

The following code, attached to the *OnMeasureItem* event of an owner-draw list box, increases the height of each list item to accommodate its associated bitmap.

```
procedure TFMForm.DriveTabSetMeasureTab(Sender: TObject; Index: Integer;
  var TabWidth: Integer);  { note that TabWidth is a var parameter}
var
  BitmapWidth: Integer;
begin
  BitmapWidth := TBitmap(DriveTabSet.Tabs.Objects[Index]).Width;
  { increase tab width by the width of the associated bitmap plus two }
  Inc(TabWidth, 2 + BitmapWidth);
end;
```

Note You must typecast the items from the *Objects* property in the string list. *Objects* is a property of type *TObject* so that it can hold any kind of object. When you retrieve objects from the array, you need to typecast them back to the actual type of the items.

Drawing owner-draw items

When an application needs to draw or redraw an owner-draw control, Windows generates draw-item events for each visible item in the control. Depending on the control, the item may also receive draw events for the item as a whole or sub items.

To draw each item in an owner-draw control, attach an event handler to the draw-item event for that control.

The names of events for owner drawing typically start with one of the following:

- *OnDraw*, such as *OnDrawItem* or *OnDrawCell*
- *OnCustomDraw*, such as *OnCustomDrawItem*

The draw-item event contains parameters identifying the item to draw, the rectangle in which to draw, and usually some information about the state of the item (such as whether the item has focus). The application handles each event by rendering the appropriate item in the given rectangle.

For example, the following code shows how to draw items in a list box that has bitmaps associated with each string. It attaches this handler to the *OnDrawItem* event for the list box:

```
procedure TFMForm.DriveTabSetDrawTab(Sender: TObject; TabCanvas: TCanvas;
  R: TRect; Index: Integer; Selected: Boolean);
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap(DriveTabSet.Tabs.Objects[Index]);
  with TabCanvas do
  begin
  Draw(R.Left, R.Top + 4, Bitmap);  { draw bitmap }
  TextOut(R.Left + 2 + Bitmap.Width,  { position text }
    R.Top + 2, DriveTabSet.Tabs[Index]);  { and draw it to the right of the
                                            bitmap }
  end;
end;
```

Chapter

8

Working with graphics

Graphics elements can add polish to your applications. CLX offers a variety of ways to introduce graphics into your application. To add graphical elements, you can insert pre-drawn pictures at design time, create them using graphical controls at design time, or draw them dynamically at runtime.

Overview of graphics programming

CLX graphics components encapsulate the Qt graphics widgets, making it very easy to add graphics to your linux applications.

To draw graphics in a CLX application, you draw on an object's *canvas*, rather than directly on the object. The canvas is a property of the object, and is itself an object. A main advantage of the canvas object is that it handles resources effectively and manages the graphics context for you, so your programs can use the same methods regardless of whether you are drawing on the screen, to a printer, or on bitmaps or drawings. Canvases are available only at runtime, so you do all your work with canvases by writing code.

Note *TCanvas* is a wrapper resource manager around a Qt painter. The *Handle* property of the canvas *is* typed pointer to an instance of a Qt painter object. Having this instance pointer exposed allows you to use low-level Qt graphics library functions that require an instance pointer to a painter object.

How graphic images appear in your application depends on the type of object whose canvas you draw on. If you are drawing directly onto the canvas of a control, the picture is displayed immediately. However, if you draw on an offscreen image such as a *TBitmap* canvas, the image is not displayed until a control copies from the bitmap onto the control's canvas. That is, when drawing bitmaps and assigning them to an image control, the image appears only when the control has an opportunity to process its *OnPaint* event.

When working with graphics, you often encounter the terms *drawing* and *painting*:

- Drawing is the creation of a single, specific graphic element, such as a line or a shape, with code. In your code, you tell an object to draw a specific graphic in a specific place on its canvas by calling a drawing method of the canvas.
- Painting is the creation of the entire appearance of an object. Painting usually involves drawing. That is, in response to *OnPaint* events, an object generally draws some graphics. An edit box, for example, paints itself by drawing a rectangle and then drawing some text inside. A shape control, on the other hand, paints itself by drawing a single graphic.

The examples in the beginning of this chapter demonstrate how to draw various graphics, but they do so in response to *OnPaint* events. Later sections show how to do the same kind of drawing in response to other events.

Refreshing the screen

At certain times, the system determines that objects onscreen need to refresh their appearance. This is the case when a form or control is temporarily obscured, for example during window dragging, the form or control must repaint the obscured area when it is re-exposed. When this occurs a paint event is generated, which CLX routes to *OnPaint* events. CLX calls any *OnPaint* event handler that you have written for that object when you use the *Refresh* method. The default name generated for the *OnPaint* event handler in a form is *FormPaint*. You may want to use the *Refresh* method at times to refresh a component or form. For example, you might call *Refresh* in the form's *OnResize* event handler to redisplay any graphics.

If you use the *TImage* control to display a graphical image on a form, the painting and refreshing of the graphic contained in the *TImage* is handled automatically by CLX. The Picture property specifies the actual bitmap, drawing, or other graphic object that *TImage* displays. Drawing on a *TImage* creates a persistent image. Consequently, you do not need to do anything to redraw the contained image. In contrast, *TPaintBox*'s canvas maps directly onto the painter, so that anything drawn to the *PaintBox*'s canvas is transitory. This is true of nearly all controls, including the form itself. Therefore, if you draw or paint on a *TPaintBox* in its constructor, you will need to add that code to your *OnPaint* event handler in order for the image to be repainted each time the client area is invalidated.

Types of graphic objects

CLX provides the graphic objects shown in Table 8.1. These objects have methods to draw on the canvas, which are described in "Using Canvas methods to draw graphic objects" on page 8-9 and to load and save to graphics files, as described in "Using the clipboard with graphics" on page 8-20.

Table 8.1 Graphic object types

Object	Description
Picture	Used to hold any graphic image. To add additional graphic file formats, use the Picture *Register* method. Use this to handle arbitrary files such as displaying images in an image control.
Bitmap	A powerful graphics object used to create, manipulate (scale, scroll, rotate, and paint), and store images as files on a disk. Creating copies of a bitmap is fast since the *handle* is copied, not the image.
Clipboard	Represents the container for any text or graphics that are cut, copied, or pasted from or to an application. With the clipboard, you can get and retrieve data according to the appropriate format; handle reference counting, and opening and closing the clipboard; manage and manipulate formats for objects in the clipboard.
Icon	Represents the value loaded from an icon file.
Drawing	Contains a file, that records the operations required to construct an image, rather than contain the actual bitmap pixels of the image. Drawings are scalable without the loss of image detail and often require much less memory than bitmaps, particularly for high-resolution devices, such as printers. However, drawings do not display as fast as bitmaps. Use a drawing when versatility or precision is more important than performance.

Common properties and methods of Canvas

Table 8.2 lists the commonly used properties of the Canvas object. For a complete list of properties and methods, see the *TCanvas* component in online Help.

Table 8.2 Common properties of the Canvas object

Properties	Descriptions
Font	Specifies the font to use when writing text on the image. Set the properties of the TFont object to specify the font face, color, size, and style of the font.
Brush	Determines the color and pattern the canvas uses for filling graphical shapes and backgrounds. Set the properties of the TBrush object to specify the color and pattern or bitmap to use when filling in spaces on the canvas.
Pen	Specifies the kind of pen the canvas uses for drawing lines and outlining shapes. Set the properties of the TPen object to specify the color, style, width, and mode of the pen.
PenPos	Specifies the current drawing position of the pen.

These properties are described in more detail in "Using the properties of the Canvas object" on page 8-5.

Table 8.3 is a list of several methods you can use:

Table 8.3 Common methods of the Canvas object

Method	Descriptions
Arc	Draws an arc on the image along the perimeter of the ellipse bounded by the specified rectangle.
Chord	Draws a closed figure represented by the intersection of a line and an ellipse.
CopyRect	Copies part of an image from another canvas into the canvas.
Draw	Renders the graphic object specified by the Graphic parameter on the canvas at the location given by the coordinates (X, Y).
DrawFocusRect	Draws a stippled rectangle that is used to indicate keyboard focus with a pen whose Mode is automatically set to pmXor. This function only has an effect if the DefaultStyle member of the Style property in the application object is set to *dsWindows*.
DrawPoint	Draws a single point on the canvas using the current pen.
DrawPoints	Draws a series of points on the canvas using the current pen.
Ellipse	Draws the ellipse defined by a bounding rectangle on the canvas.
FillRect	Fills the specified rectangle on the canvas using the current brush.
GetClipRegion	Returns a pointer to the clipping region that is currently set.
LineTo	Draws a line on the canvas from PenPos to the point specified by X and Y, and sets the pen position to (X, Y).
MoveTo	Changes the current drawing position to the point (X,Y).
Pie	Draws a pie-shaped the section of the ellipse bounded by the rectangle (X1, Y1) and (X2, Y2) on the canvas.
PolyBezier	Draws a cubic Bézier curve using the current pen.
PolyBezierTo	Draws a set of Bézier curves and updates the value of PenPos.
Polygon	Draws a series of lines on the canvas connecting the points passed in and closing the shape by drawing a line from the last point to the first point.
Polyline	Draws a series of lines on the canvas with the current pen, connecting each of the points passed to it in Points.
Rectangle	Draws a rectangle on the canvas with its upper left corner at the point (X1, Y1) and its lower right corner at the point (X2, Y2). Use *Rectangle* to draw a box using Pen and fill it using Brush.
RoundRect	Draws a rectangle with rounded corners on the canvas.
StretchDraw	Draws a graphic on the canvas so that the image fits in the specified rectangle. The graphic image may need to change its magnitude or aspect ratio to fit.
TextHeight, TextWidth	Returns the height and width, respectively, of a string in the current font. Height includes leading between lines.
TextOut	Writes a string on the canvas, starting at the point (X,Y), and then updates the PenPos to the end of the string.
TextRect	Writes a string inside a region; any portions of the string that fall outside the region do not appear.
TiledDraw	Draws a tiled bitmap in a specified rectangle.

These methods are described in more detail in "Using Canvas methods to draw graphic objects" on page 8-9.

Using the properties of the Canvas object

With the Canvas object, you can set the properties of a pen for drawing lines, a brush for filling shapes, a font for writing text, and an array of pixels to represent the image.

This section describes

- Using pens
- Using brushes

Using pens

The *Pen* property of a canvas controls the way lines appear, including lines drawn as the outlines of shapes. Drawing a straight line is really just changing a group of pixels that lie between two points.

The pen itself has four properties you can change: *Color*, *Width*, *Style*, and *Mode*.

- Color property: Changes the pen color
- Width property: Changes the pen width
- Style property: Changes the pen style
- Mode property: Changes the pen mode

The values of these properties determine how the pen changes the pixels in the line. By default, every pen starts out black, with a width of 1 pixel, a solid style, and a mode called copy that overwrites anything already on the canvas.

Changing the pen color

You can set the color of a pen as you would any other *Color* property at runtime. A pen's color determines the color of the lines the pen draws, including lines drawn as the boundaries of shapes, as well as other lines and polylines. To change the pen color, assign a value to the *Color* property of the pen.

To let the user choose a new color for the pen, put a color grid on the pen's toolbar. A color grid can set both foreground and background colors. For a non-grid pen style, you must consider the background color, which is drawn in the gaps between line segments. Background color comes from the Brush color property.

Since the user chooses a new color by clicking the grid, this code changes the pen's color in response to the *OnClick* event:

```
procedure TForm1.PenColorClick(Sender: TObject);
begin
  Canvas.Pen.Color := PenColor.ForegroundColor;
end;
```

Changing the pen width

A pen's width determines the thickness, in pixels, of the lines it draws.

Note If you are developing a cross-platform application for deployment under Windows 95/98 and the thickness is greater than 1, Windows 95/98 always draws solid lines, regardless of the value of the pen's *Style* property.

To change the pen width, assign a numeric value to the pen's *Width* property.

Suppose you have a scroll bar on the pen's toolbar to set width values for the pen. And suppose you want to update the label next to the scroll bar to provide feedback to the user. Using the scroll bar's position to determine the pen width, you update the pen width every time the position changes.

This is how to handle the scroll bar's *OnChange* event:

```
procedure TForm1.PenWidthChange(Sender: TObject);
begin
  Canvas.Pen.Width := PenWidth.Position;{ set the pen width directly }
 PenSize.Caption := IntToStr(PenWidth.Position);{ convert to string for caption }
end;
```

Changing the pen style

A pen's *Style* property allows you to set solid lines, dashed lines, dotted lines, and so on.

Note If you are developing a cross-platform application for deployment under Windows 95/98, Windows 95/98 does not support dashed or dotted line styles for pens wider than one pixel and makes all larger pens solid, no matter what style you specify.

The task of setting the properties of pen is an ideal case for having different controls share same event handler to handle events. To determine which control actually got the event, you check the *Sender* parameter.

To create one click-event handler for six pen-style buttons on a pen's toolbar, do the following:

1. Select all six pen-style buttons and select the Object Inspector | Events | *OnClick* event and in the Handler column, type `SetPenStyle`.
2. The IDE generates an empty click-event handler called *SetPenStyle* and attaches it to the *OnClick* events of all six buttons.
3. Fill in the click-event handler by setting the pen's style depending on the value of *Sender*, which is the control that sent the click event:

```
procedure TForm1.SetPenStyle(Sender: TObject);
begin
  with Canvas.Pen do
  begin
    if Sender = SolidPen then Style := psSolid
   else if Sender = DashPen then Style := psDash
   else if Sender = DotPen then Style := psDot
   else if Sender = DashDotPen then Style := psDashDot
```

```
      else if Sender = DashDotDotPen then Style := psDashDotDot
      else if Sender = ClearPen then Style := psClear;
    end;
  end;
```

Changing the pen mode

A pen's *Mode* property lets you specify various ways to combine the pen's color with the color on the canvas. For example, the pen could always be black, be an inverse of the canvas background color, inverse of the pen color, and so on. See *TPen* in online Help for details.

Getting the pen position

The current drawing position—the position from which the pen begins drawing its next line—is called the pen position. The canvas stores its pen position in its *PenPos* property. Pen position affects the drawing of lines only; for shapes and text, you specify all the coordinates you need.

To set the pen position, call the *MoveTo* method of the canvas. For example, the following code moves the pen position to the upper left corner of the canvas:

```
Canvas.MoveTo(0, 0);
```

Note Drawing a line with the *LineTo* method also moves the current position to the endpoint of the line.

Using brushes

The *Brush* property of a canvas controls the way you fill areas, including the interior of shapes. Filling an area with a brush is a way of changing a large number of adjacent pixels in a specified way.

The brush has three properties you can manipulate:

- Color property: Changes the fill color
- Style property: Changes the brush style
- Bitmap property: Uses a bitmap as a brush pattern

The values of these properties determine the way the canvas fills shapes or other areas. By default, every brush starts out white, with a solid style and no pattern bitmap.

Changing the brush color

A brush's color determines what color the canvas uses to fill shapes. To change the fill color, assign a value to the brush's *Color* property. Brush is used for background color in text and line drawing so you typically set the background color property.

You can set the brush color just as you do the pen color, in response to a click on a color grid on the brush's toolbar (see "Changing the pen color" on page 8-5):

```
procedure TForm1.BrushColorClick(Sender: TObject);
begin
  Canvas.Brush.Color := BrushColor.ForegroundColor;
end;
```

Changing the brush style

A brush style determines what pattern the canvas uses to fill shapes. It lets you specify various ways to combine the brush's color with any colors already on the canvas. The predefined styles include solid color, no color, and various line and hatch patterns.

To change the style of a brush, set its *Style* property to one of the predefined values: *bsSolid*, *bsClear*, *bsHorizontal*, *bsVertical*, *bsFDiagonal*, *bsBDiagonal*, *bsCross*, *bsDiagCross*, *bsDense1*, *bsDense2*, *bsDense3*, *bsDense4*, *bsDense5*, *bsDense6*, or *bsDense7*.

This example sets brush styles by sharing a click-event handler for a set of fifteen brush-style buttons. All fifteen buttons are selected, the Object Inspector | Events | *OnClick* is set, and the *OnClick* handler is named *SetBrushStyle*. Here is the handler code:

```
procedure TForm1.SetBrushStyle(Sender: TObject);
begin
  with Canvas.Brush do
  begin
    if Sender = SolidBrush then Style := bsSolid
   else if Sender = ClearBrush then Style := bsClear
   else if Sender = HorizontalBrush then Style := bsHorizontal
   else if Sender = VerticalBrush then Style := bsVertical
   else if Sender = FDiagonalBrush then Style := bsFDiagonal
   else if Sender = BDiagonalBrush then Style := bsBDiagonal
   else if Sender = CrossBrush then Style := bsCross
   else if Sender = DiagCrossBrush then Style := bsDiagCross;
  end;
end;
```

Setting the Brush Bitmap property

A brush's *Bitmap* property lets you specify a bitmap image for the brush to use as a pattern for filling shapes and other areas.

The following example loads a bitmap from a file and assigns it to the Brush of the Canvas of Form1:

```
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create;
  try
    Bitmap.LoadFromFile('MyBitmap.bmp');
    Form1.Canvas.Brush.Bitmap := Bitmap;
    Form1.Canvas.FillRect(Rect(0,0,100,100));
  finally
    Form1.Canvas.Brush.Bitmap := nil;
```

```
      Bitmap.Free;
    end;
  end;
```

Note The brush does not assume ownership of a bitmap object assigned to its *Bitmap* property. You must ensure that the Bitmap object remains valid for the lifetime of the Brush, and you must free the Bitmap object yourself afterwards.

Using Canvas methods to draw graphic objects

This section shows how to use some common methods to draw graphic objects. It covers:

- Drawing lines and polylines
- Drawing shapes
- Drawing rounded rectangles
- Drawing polygons

Drawing lines and polylines

A canvas can draw straight lines and polylines. A straight line is just a line of pixels connecting two points. A polyline is a series of straight lines, connected end-to-end. The canvas draws all lines using its pen.

Drawing lines

To draw a straight line on a canvas, use the *LineTo* method of the canvas.

LineTo draws a line from the current pen position to the point you specify and makes the endpoint of the line the current position. The canvas draws the line using its pen.

For example, the following method draws crossed diagonal lines across a form whenever the form is painted:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 with Canvas do
 begin
   MoveTo(0, 0);
   LineTo(ClientWidth, ClientHeight);
   MoveTo(0, ClientHeight);
   LineTo(ClientWidth, 0);
 end;
end;
```

Drawing polylines

In addition to individual lines, the canvas can also draw polylines, which are groups of any number of connected line segments.

To draw a polyline on a canvas, call the *Polyline* method of the canvas.

The parameter passed to the *Polyline* method is an array of points. You can think of a polyline as performing a *MoveTo* on the first point and *LineTo* on each successive point. For drawing multiple lines, *Polyline* is faster than using the *MoveTo* method and the *LineTo* method because it eliminates a lot of call overhead.

The following method, for example, draws a rhombus in a form:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 with Canvas do
   Polyline([Point(0, 0), Point(50, 0), Point(75, 50), Point(25, 50), Point(0, 0)]);
end;
```

This example takes advantage of Kylix's ability to create an open-array parameter on-the-fly. You can pass any array of points, but an easy way to construct an array quickly is to put its elements in brackets and pass the whole thing as a parameter. For more information, see online Help.

Drawing shapes

Canvases have methods for drawing different kinds of shapes. The canvas draws the outline of a shape with its pen, then fills the interior with its brush. The line that forms the border for the shape is controlled by the current *Pen* object.

This section covers:

- Drawing rectangles and ellipses
- Drawing rounded rectangles
- Drawing polygons

Drawing rectangles and ellipses

To draw a rectangle or ellipse on a canvas, call the canvas's *Rectangle* method or *Ellipse* method, passing the coordinates of a bounding rectangle.

The *Rectangle* method draws the bounding rectangle; *Ellipse* draws an ellipse that touches all sides of the rectangle.

The following method draws a rectangle filling a form's upper left quadrant, then draws an ellipse in the same area:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Rectangle(0, 0, ClientWidth div 2, ClientHeight div 2);
 Canvas.Ellipse(0, 0, ClientWidth div 2, ClientHeight div 2);
end;
```

Drawing rounded rectangles

To draw a rounded rectangle on a canvas, call the canvas's *RoundRect* method.

The first four parameters passed to *RoundRect* are a bounding rectangle, just as for the *Rectangle* method or the *Ellipse* method. *RoundRect* takes two more parameters that indicate how to draw the rounded corners.

The following method, for example, draws a rounded rectangle in a form's upper left quadrant, rounding the corners as sections of a circle with a diameter of 10 pixels:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.RoundRect(0, 0, ClientWidth div 2, ClientHeight div 2, 10, 10);
end;
```

Drawing polygons

To draw a polygon with any number of sides on a canvas, call the *Polygon* method of the canvas.

Polygon takes an array of points as its only parameter and connects the points with the pen, then connects the last point to the first to close the polygon. After drawing the lines, *Polygon* uses the brush to fill the area inside the polygon.

For example, the following code draws a right triangle in the lower left half of a form:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Polygon([Point(0, 0), Point(0, ClientHeight),
   Point(ClientWidth, ClientHeight)]);
end;
```

Handling multiple drawing objects in your application

Various drawing methods (rectangle, shape, line, and so on) are typically available on the toolbar and button panel. Applications can respond to clicks on speed buttons to set the desired drawing objects. This section describes how to:

- Keep track of which drawing tool to use
- Changing the tool with speed buttons
- Using drawing tools

Keeping track of which drawing tool to use

A graphics program needs to keep track of what kind of drawing tool (such as a line, rectangle, ellipse, or rounded rectangle) a user might want to use at any given time. You could assign numbers to each kind of tool, but then you would have to remember what each number stands for. You can do that more easily by assigning mnemonic constant names to each number, but your code won't be able to distinguish which numbers are in the proper range and of the right type. Fortunately, Object Pascal provides a means to handle both of these shortcomings. You can declare an enumerated type.

An enumerated type is really just a shorthand way of assigning sequential values to constants. Since it's also a type declaration, you can use Object Pascal's type-checking to ensure that you assign only those specific values.

To declare an enumerated type, use the reserved work type, followed by an identifier for the type, then an equal sign, and the identifiers for the values in the type in parentheses, separated by commas.

For example, the following code declares an enumerated type for each drawing tool available in a graphics application:

```
type
  TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
```

By convention, type identifiers begin with the letter *T*, and groups of similar constants (such as those making up an enumerated type) begin with a 2-letter prefix (such as *dt* for “drawing tool”).

The declaration of the TDrawingTool type is equivalent to declaring a group of constants:

```
const
  dtLine = 0;
  dtRectangle = 1;
  dtEllipse = 2;
  dtRoundRect = 3;
```

The main difference is that by declaring the enumerated type, you give the constants not just a value, but also a type, which enables you to use Object Pascal's type-checking to prevent many errors. A variable of type TDrawingTool can be assigned only one of the constants dtLine..dtRoundRect. Attempting to assign some other number (even one in the range 0..3) generates a compile-time error.

In the following code, a field added to a form keeps track of the form’s drawing tool:

```
type
  TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
 TForm1 = class(TForm)
    ...{ method declarations }
 public
    Drawing: Boolean;
   Origin, MovePt: TPoint;
   DrawingTool: TDrawingTool;{ field to hold current tool }
  end;
```

Changing the tool with speed buttons

Each drawing tool needs an associated *OnClick* event handler. Suppose your application had a toolbar button for each of four drawing tools: line, rectangle, ellipse, and rounded rectangle. You would attach the following event handlers to the *OnClick* events of the four drawing-tool buttons, setting *DrawingTool* to the appropriate value for each:

```
procedure TForm1.LineButtonClick(Sender: TObject);{ LineButton }
begin
  DrawingTool := dtLine;
end;

procedure TForm1.RectangleButtonClick(Sender: TObject);{ RectangleButton }
begin
```

```
  DrawingTool := dtRectangle;
end;

procedure TForm1.EllipseButtonClick(Sender: TObject);{ EllipseButton }
begin
  DrawingTool := dtEllipse;
end;

procedure TForm1.RoundedRectButtonClick(Sender: TObject);{ RoundRectButton }
begin
  DrawingTool := dtRoundRect;
end;
```

Using drawing tools

Now that you can tell what tool to use, you must indicate how to draw the different shapes. The only methods that perform any drawing are the mouse-move and mouse-up handlers, and the only drawing code draws lines, no matter what tool is selected.

To use different drawing tools, your code needs to specify how to draw, based on the selected tool. You add this instruction to each tool's event handler.

This section describes

- Drawing shapes
- Sharing code among event handlers

Drawing shapes

Drawing shapes is just as easy as drawing lines: Each one takes a single statement; you just need the coordinates.

Here's a rewrite of the *OnMouseUp* event handler that draws shapes for all four tools:

```
procedure TForm1.FormMouseUp(Sender: TObject; Button TMouseButton; Shift: TShiftState;
                             X,Y: Integer);
begin
  case DrawingTool of
    dtLine:
      begin
        Canvas.MoveTo(Origin.X, Origin.Y);
       Canvas.LineTo(X, Y)
      end;
   dtRectangle: Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
   dtEllipse: Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
   dtRoundRect: Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                   (Origin.X - X) div 2, (Origin.Y - Y) div 2);
  end;
 Drawing := False;
end;
```

Of course, you also need to update the *OnMouseMove* handler to draw shapes:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin
```

```
    if Drawing then
    begin
      Canvas.Pen.Mode := pmNotXor;
     case DrawingTool of
        dtLine: begin
                  Canvas.MoveTo(Origin.X, Origin.Y);
                 Canvas.LineTo(MovePt.X, MovePt.Y);
                 Canvas.MoveTo(Origin.X, Origin.Y);
                 Canvas.LineTo(X, Y);
                end;
      dtRectangle: begin
                     Canvas.Rectangle(Origin.X, Origin.Y, MovePt.X, MovePt.Y);
                    Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
                   end;
      dtEllipse: begin
                   Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
                  Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
                 end;
      dtRoundRect: begin
                    Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                      (Origin.X - X) div 2, (Origin.Y - Y) div 2);
                   Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                      (Origin.X - X) div 2, (Origin.Y - Y) div 2);
                  end;
      end;
     MovePt := Point(X, Y);
    end;
   Canvas.Pen.Mode := pmCopy;
  end;
```

Typically, all the repetitious code that is in the above example would be in a separate routine. The next section shows all the shape-drawing code in a single routine that all mouse-event handlers can call.

Sharing code among event handlers

Any time you find that many your event handlers use the same code, you can make your application more efficient by moving the repeated code into a routine that all event handlers can share.

To add a method to a form,

1 Add the method declaration to the form object.

   You can add the declaration in either the **public** or **private** parts at the end of the form object's declaration. If the code is just sharing the details of handling some events, it's probably safest to make the shared method **private**.

2 Write the method implementation in the implementation part of the form unit.

The header for the method implementation must match the declaration exactly, with the same parameters in the same order.

The following code adds a method to the form called *DrawShape* and calls it from each of the handlers. First, the declaration of *DrawShape* is added to the form object's declaration:

```
type
  TForm1 = class(TForm)
    ...{ fields and methods declared here}
 public
    { Public declarations }
   procedure DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
  end;
```

Then, the implementation of *DrawShape* is written in the implementation part of the unit:

```
implementation
{$R *.FRM}
...{ other method implementations omitted for brevity }
procedure TForm1.DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
begin
  with Canvas do
  begin
    Pen.Mode := AMode;
   case DrawingTool of
      dtLine:
        begin
          MoveTo(TopLeft.X, TopLeft.Y);
         LineTo(BottomRight.X, BottomRight.Y);
        end;
     dtRectangle: Rectangle(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
     dtEllipse: Ellipse(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
     dtRoundRect: RoundRect(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y,
        (TopLeft.X - BottomRight.X) div 2, (TopLeft.Y - BottomRight.Y) div 2);
    end;
  end;
end;
```

The other event handlers are modified to call *DrawShape*.

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  DrawShape(Origin, Point(X, Y), pmCopy);{ draw the final shape }
 Drawing := False;
end;
procedure TForm1.FormMouseMove(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    DrawShape(Origin, MovePt, pmNotXor);{ erase the previous shape }
   MovePt := Point(X, Y);{ record the current point }
   DrawShape(Origin, MovePt, pmNotXor);{ draw the current shape }
  end;
end;
```

Drawing on a graphic

You don't need any components to manipulate your application's graphic objects. You can construct, draw on, save, and destroy graphic objects without ever drawing anything on screen. In fact, your applications rarely draw directly on a form. More often, an application operates on graphics and then uses a CLX image control component to display the graphic on a form.

Once you move the application's drawing to the graphic in the image control, it is easy to add printing, clipboard, and loading and saving operations for any graphic objects. graphic objects can be bitmap files, drawings, icons or whatever other graphics classes that have been installed such as jpeg graphics.

Note Because you are drawing on an offscreen image such as a *TBitmap* canvas, the image is not displayed until a control copies from a bitmap onto the control's canvas. That is, when drawing bitmaps and assigning them to an image control, the image appears only when the control has an opportunity to process its paint message. But if you are drawing directly onto the canvas property of a control, the picture object is displayed immediately.

Making scrollable graphics

The graphic need not be the same size as the form: it can be either smaller or larger. By adding a scroll box control to the form and placing the graphic image inside it, you can display graphics that are much larger than the form or even larger than the screen. To add a scrollable graphic first you add a *TScrollBox* component and then you add the image control.

Adding an image control

An image control is a container component that allows you to display your bitmap objects. You use an image control to hold a bitmap that is not necessarily displayed all the time, or which an application needs to use to generate other pictures.

Note “Adding graphics to controls” on page 7-7 shows how to use graphics in controls.

Placing the control

You can place an image control anywhere on a form. If you take advantage of the image control's ability to size itself to its picture, you need to set the top left corner only. If the image control is a nonvisible holder for a bitmap, you can place it anywhere, just as you would a nonvisual component.

If you drop the image control on a scroll box already aligned to the form's client area, this assures that the scroll box adds any scroll bars necessary to access offscreen portions of the image's picture. Then set the image control's properties.

Setting the initial bitmap size

When you place an image control, it is simply a container. However, you can set the image control's *Picture* property at design time to contain a static graphic. The control can also load its picture from a file at runtime, as described in "Using the clipboard with graphics" on page 8-20.

To create a blank bitmap when the application starts,

1 Attach a handler to the *OnCreate* event for the form that contains the image.

2 Create a bitmap object, and assign it to the image control's *Picture.Graphic* property.

In this example, the image is in the application's main form, *Form1*, so the code attaches a handler to *Form1*'s *OnCreate* event:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Bitmap: TBitmap;{ temporary variable to hold the bitmap }
begin
  Bitmap := TBitmap.Create;{ construct the bitmap object }
 Bitmap.Width := 200;{ assign the initial width... }
 Bitmap.Height := 200;{ ...and the initial height }
 Image.Picture.Graphic := Bitmap;{ assign the bitmap to the image control }
 Bitmap.Free; {We are done with the bitmap, so free it }
end;
```

Assigning the bitmap to the picture's *Graphic* property copies the bitmap to the picture object. However, the picture object does not take ownership of the bitmap, so after making the assignment, you must free it.

If you run the application now, you see that client area of the form has a white region, representing the bitmap. If you size the window so that the client area cannot display the entire image, you'll see that the scroll box automatically shows scroll bars to allow display of the rest of the image. But if you try to draw on the image, you don't get any graphics, because the application is still drawing on the form, which is now behind the image and the scroll box.

Drawing on the bitmap

To draw on a bitmap, use the image control's canvas and attach the mouse-event handlers to the appropriate events in the image control. Typically, you would use region operations (fills, rectangles, polylines, and so on). These are fast and efficient methods of drawing.

An efficient way to draw images when you need to access individual pixels is to use the bitmap *ScanLine* property. For general-purpose usage, you can set up the bitmap pixel format to 24 bits and then treat the pointer returned from *ScanLine* as an array of RGB. Otherwise, you will need to know the native format of the *ScanLine* property. This example shows how to use *ScanLine* to get pixels one line at a time.

```
procedure TForm1.Button1Click(Sender: TObject);
// This example shows drawing directly to the Bitmap
var
  x,y : integer;
```

```
    Bitmap : TBitmap;
    P : PByteArray;
  begin
    Bitmap := TBitmap.create;
    try
      if OpenDialog1.Execute then
      begin
        Bitmap.LoadFromFile(OpenDialog1.FileName);
      for y := 0 to Bitmap.height -1 do
        begin
           P := Bitmap.ScanLine[y];
           for x := 0 to Bitmap.width -1 do
              P[x] := y;
        end;
      end;
    canvas.draw(0,0,Bitmap);
    finally
      Bitmap.free;
    end;
  end;
```

Loading and saving graphics files

Graphic images that exist only for the duration of one running of an application are of very limited value. Often, you either want to use the same picture every time, or you want to save a created picture for later use. The image component makes it easy to load pictures from a file and save them again.

CLX components you use to load, save, and replace graphic images support many graphic formats including bitmap files, png files, xpms, Windows icons, and so on. They also support installable graphic classes.

The way to load and save graphics files is the similar to any other files and is described in the following sections:

- Loading a picture from a file
- Saving a picture to a file
- Replacing the picture

Loading a picture from a file

Your application should provide the ability to load a picture from a file if your application needs to modify the picture or if you want to store the picture outside the application so a person or another application can modify the picture.

To load a graphics file into an image control, call the *LoadFromFile* method of the image control's *Picture* object.

The following code gets a file name from an open picture file dialog box, and then loads that file into an image control named *Image*:

```
procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    CurrentFile := OpenDialog1.FileName;
    Image.Picture.LoadFromFile(CurrentFile);
  end;
end;
```

Saving a picture to a file

The picture object can load and save graphics in several formats, and you can create and register your own graphic-file formats so that picture objects can load and store them as well.

To save the contents of an image control in a file, call the *SaveToFile* method of the image control's *Picture* object.

The *SaveToFile* method requires the name of a file in which to save. If the picture is newly created, it might not have a file name, or a user might want to save an existing picture in a different file. In either case, the application needs to get a file name from the user before saving, as shown in the next section.

The following pair of event handlers, attached to the File|Save and File|Save As menu items, respectively, handle the resaving of named files, saving of unnamed files, and saving existing files under new names.

```
procedure TForm1.Save1Click(Sender: TObject);
begin
  if CurrentFile <> '' then
    Image.Picture.SaveToFile(CurrentFile){ save if already named }
 else SaveAs1Click(Sender);{ otherwise get a name }
end;
procedure TForm1.Saveas1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then{ get a file name }
  begin
    CurrentFile := SaveDialog1.FileName;{ save the user-specified name }
   Save1Click(Sender);{ then save normally }
  end;
end;
```

Replacing the picture

You can replace the picture in an image control at any time. If you assign a new graphic to a picture that already has a graphic, the new graphic replaces the existing one.

To replace the picture in an image control, assign a new graphic to the image control's *Picture* object.

Creating the new graphic is the same process you used to create the initial graphic (see "Setting the initial bitmap size" on page 8-17), but you should also provide a way for the user to choose a size other than the default size used for the initial graphic. An easy way to provide that option is to present a dialog box. With a dialog box in your project, add it to the uses clause in the unit for your main form. You can then attach an event handler to the File|New menu item's *OnClick* event. Here's an example:

```
procedure TForm1.New1Click(Sender: TObject);
var
  Bitmap: TBitmap;{ temporary variable for the new bitmap }
begin
  with NewBMPForm do
  begin
    ActiveControl := WidthEdit;{ make sure focus is on width field }
   WidthEdit.Text := IntToStr(Image.Picture.Graphic.Width);{ use current dimensions... }
   HeightEdit.Text := IntToStr(Image.Picture.Graphic.Height);{ ...as default }
   if ShowModal <> idCancel then{ continue if user doesn't cancel dialog box }
    begin
      Bitmap := TBitmap.Create;{ create fresh bitmap object }
     Bitmap.Width := StrToInt(WidthEdit.Text);{ use specified width }
     Bitmap.Height := StrToInt(HeightEdit.Text);{ use specified height }
     Image.Picture.Graphic := Bitmap;{ replace graphic with new bitmap }
     CurrentFile := '';{ indicate unnamed file }
     Bitmap.Free;
    end;
  end;
end;
```

Note Assigning a new bitmap to the picture object's *Graphic* property causes the picture object to copy the new graphic, but it does not take ownership of it. The picture object maintains its own internal graphic object. Because of this, the previous code frees the bitmap object after making the assignment.

Using the clipboard with graphics

You can use the clipboard to copy and paste graphics within your applications or to exchange graphics with other applications. The Clipboard object makes it easy to handle different kinds of information, including graphics.

Before you can use the Clipboard object in your application, you must add the QClipbrd unit to the uses clause of any unit that needs to access clipboard data.

Data that is stored on the clipboard is stored as a mime type with an associated *TStream* object. CLX provides the following predefined mime source and mime type string constants for the following CLX objects:

- TBitmap = 'image/delphi.bitmap'
- TComponent = 'application/delphi.component'
- TPicture = 'image/delphi.picture'
- TDrawing = 'image/delphi.drawing'

Copying graphics to the clipboard

You can copy any picture, including the contents of image controls, to the clipboard. Once on the clipboard, the picture is available to all applications.

To copy a picture to the clipboard, assign the picture to the clipboard object using the *Assign* method.

This code shows how to copy the picture from an image control named *Image* to the clipboard in response to a click on an Edit|Copy menu item:

```
procedure TForm1.Copy1Click(Sender: TObject);
begin
  Clipboard.Assign(Image.Picture)
end.
```

Cutting graphics to the clipboard

Cutting a graphic to the clipboard is exactly like copying it, but you also erase the graphic from the source.

To cut a graphic from a picture to the clipboard, first copy it to the clipboard, then erase the original.

In most cases, the only issue with cutting is how to show that the original image is erased. Setting the area to white is a common solution, as shown in the following code that attaches an event handler to the *OnClick* event of the Edit|Cut menu item:

```
procedure TForm1.Cut1Click(Sender: TObject);
var
  ARect: TRect;
begin
  Copy1Click(Sender);{ copy picture to clipboard }
 with Image.Canvas do
  begin
    CopyMode := cmWhiteness;{ copy everything as white }
   ARect := Rect(0, 0, Image.Width, Image.Height);{ get bitmap rectangle }
   CopyRect(ARect, Image.Canvas, ARect);{ copy bitmap over itself }
   CopyMode := cmSrcCopy;{ restore normal mode }
  end;
end;
```

Pasting graphics from the clipboard

If the clipboard contains a bitmapped graphic, you can paste it into any image object, including image controls and the surface of a form.

To paste a graphic from the clipboard,

1 Call the clipboard's *Provides* method to see whether the clipboard contains a graphic.

Provides is a Boolean function. It returns *True* if the clipboard contains an item of the type specified in the parameter.

2 Assign the clipboard to the destination.

This code shows how to paste a picture from the clipboard into an image control in response to a click on an Edit|Paste menu item:

```
procedure TForm1.PasteButtonClick(Sender: TObject);
var
  Bitmap: TBitmap;
begin
 if Clipboard.Provides(SDelphiBitmap) then { is there a bitmap on the clipboard? )
 begin
   Image1.Picture.Bitmap.Assign(Clipboard);
 end;
end;
```

The graphic on the clipboard could come from this application, or it could have been copied from another application. You do not need to check the clipboard format in this case because the paste menu should be disabled when the clipboard does not contain a supported format.

Rubber banding example

This section walks you through the details of implementing the “rubber banding” effect in an graphics application that tracks mouse movements as the user draws a graphic at runtime. The application draws lines and shapes on a window’s canvas in response to clicks and drags: pressing a mouse button starts drawing, and releasing the button ends the drawing.

To start with, the example code shows how to draw on the surface of the main form. Later examples demonstrate drawing on a bitmap.

The following topics describe the example:

- Responding to the mouse
- Adding a field to a form object to track mouse actions
- Refining line drawing

Responding to the mouse

Your application can respond to the mouse actions: mouse-button down, mouse moved, and mouse-button up. It can also respond to a click (a complete press-and-release, all in one place) that can be generated by some kinds of keystrokes (such as pressing *Enter* in a modal dialog box).

This section covers:

- What’s in a mouse event
- Responding to a mouse-down action
- Responding to a mouse-up action
- Responding to a mouse move

What’s in a mouse event?

A mouse event occurs when a user moves the mouse in the user interface of an application. CLX has three mouse events:

Table 8.4 Mouse events

Event	Description
OnMouseDown event	Occurs when the user presses a mouse button with the mouse pointer over a control.
OnMouseMove event	Occurs when the user moves the mouse while the mouse pointer is over a control.
OnMouseUp event	Occurs when the user releases a mouse button that was pressed with the mouse pointer over a component.

When a CLX application detects a mouse action, it calls whatever event handler you’ve defined for the corresponding event, passing five parameters. Use the information in those parameters to customize your responses to the events. The five parameters are as follows:

Table 8.5 Mouse-event parameters

Parameter	Meaning
Sender	The object that detected the mouse action
Button	Indicates which mouse button was involved: *mbLeft*, *mbMiddle*, or *mbRight*
Shift	Indicates the state of the *Alt*, *Ctrl*, and *Shift* keys at the time of the mouse action
X, *Y*	The coordinates where the event occurred

Most of the time, you need the coordinates returned in a mouse-event handler, but sometimes you also need to check *Button* to determine which mouse button caused the event.

Responding to a mouse-down action

Whenever the user presses a button on the mouse, an *OnMouseDown* event goes to the object the pointer is over. The object can then respond to the event.

To respond to a mouse-down action, attach an event handler to the *OnMouseDown* event.

CLX generates an empty handler for a mouse-down event on the form:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
 Shift: TShiftState; X, Y: Integer);
begin
end;
```

The following code displays the string 'Here!' at the location on a form clicked with the mouse:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
 Shift: TShiftState; X, Y: Integer);
begin
 Canvas.TextOut(X, Y, 'Here!');{ write text at (X, Y) }
end;
```

When the application runs, you can press the mouse button down with the mouse cursor on the form and have the string, "Here!" appear at the point clicked. This code sets the current drawing position to the coordinates where the user presses the button:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
 Shift: TShiftState; X, Y: Integer);
begin
 Canvas.MoveTo(X, Y);{ set pen position }
end;
```

Pressing the mouse button now sets the pen position, setting the line's starting point. To draw a line to the point where the user releases the button, you need to respond to a mouse-up event.

Responding to a mouse-up action

An *OnMouseUp* event occurs whenever the user releases a mouse button. The event usually goes to the object the mouse cursor is over when the user presses the button, which is not necessarily the same object the cursor is over when the button is released. This enables you, for example, to draw a line as if it extended beyond the border of the form.

To respond to mouse-up actions, define a handler for the *OnMouseUp* event.

Here's a simple *OnMouseUp* event handler that draws a line to the point of the mouse-button release:

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);{ draw line from PenPos to (X, Y) }
end;
```

This code lets a user draw lines by clicking, dragging, and releasing. In this case, the user cannot see the line until the mouse button is released.

Responding to a mouse move

An *OnMouseMove* event occurs periodically when the user moves the mouse. The event goes to the object that was under the mouse pointer when the user pressed the button. This allows you to give the user some intermediate feedback by drawing temporary lines while the mouse moves.

To respond to mouse movements, define an event handler for the *OnMouseMove* event. This example uses mouse-move events to draw intermediate shapes on a form while the user holds down the mouse button, thus providing some feedback to the user. The *OnMouseMove* event handler draws a line on a form to the location of the *OnMouseMove* event:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);{ draw line to current position }
end;
```

With this code, moving the mouse over the form causes drawing to follow the mouse, even before the mouse button is pressed.

Mouse-move events occur even when you haven't pressed the mouse button.

If you want to track whether there is a mouse button pressed, you need to add an object field to the form object.

Adding a field to a form object to track mouse actions

To track whether a mouse button was pressed, you must add an object field to the form object. When you add a component to a form, Kylix adds a field that represents that component to the form object, so that you can refer to the component by the name of its field. You can also add your own fields to forms by editing the type declaration in the form unit's header file.

In the following example, the form needs to track whether the user has pressed a mouse button. To do that, it adds a Boolean field and sets its value when the user presses the mouse button.

To add a field to an object, edit the object's type definition, specifying the field identifier and type after the **public** directive at the bottom of the declaration.

Kylix "owns" any declarations before the **public** directive: that's where it puts the fields that represent controls and the methods that respond to events.

The following code gives a form a field called *Drawing* of type Boolean, in the form object's declaration. It also adds two fields to store points *Origin* and *MovePt* of typeTPoint.

```
type
  TForm1 = class(TForm)
   procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
   procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
   procedure FormMouseMove(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
 public
    Drawing: Boolean;{ field to track whether button was pressed }
    Origin, MovePt: TPoint;{ fields to store points }
  end;
```

When you have a *Drawing* field to track whether to draw, set it to *True* when the user presses the mouse button, and *False* when the user releases it:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;{ set the Drawing flag }
 Canvas.MoveTo(X, Y);
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
```

```
    Canvas.LineTo(X, Y);
   Drawing := False;{ clear the Drawing flag }
  end;
```

Then you can modify the *OnMouseMove* event handler to draw only when *Drawing* is *True*:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then{ only draw if Drawing flag is set }
   Canvas.LineTo(X, Y);
end;
```

This results in drawing only between the mouse-down and mouse-up events, but you still get a scribbled line that tracks the mouse movements instead of a straight line.

The problem is that each time you move the mouse, the mouse-move event handler calls *LineTo*, which moves the pen position, so by the time you release the button, you've lost the point where the straight line was supposed to start.

Refining line drawing

With fields in place to track various points, you can refine an application's line drawing.

Tracking the origin point

When drawing lines, track the point where the line starts with the *Origin* field.

Origin must be set to the point where the mouse-down event occurs, so the mouse-up event handler can use *Origin* to place the beginning of the line, as in this code:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
 Canvas.MoveTo(X, Y);
 Origin := Point(X, Y);{ record where the line starts }
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }
 Canvas.LineTo(X, Y);
 Drawing := False;
end;
```

Those changes get the application to draw the final line again, but they do not draw any intermediate actions--the application does not yet support "rubber banding."

Tracking movement

The problem with this example as the *OnMouseMove* event handler is currently written is that it draws the line to the current mouse position from the last *mouse position,* not from the original position. You can correct this by moving the drawing position to the origin point, then drawing to the current point:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }
   Canvas.LineTo(X, Y);
  end;
end;
```

The above tracks the current mouse position, but the intermediate lines do not go away, so you can hardly see the final line. The example needs to erase each line before drawing the next one, by keeping track of where the previous one was. The *MovePt* field allows you to do this.

MovePt must be set to the endpoint of each intermediate line, so you can use *MovePt* and *Origin* to erase that line the next time a line is drawn:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
 Canvas.MoveTo(X, Y);
 Origin := Point(X, Y);
 MovePt := Point(X, Y);{ keep track of where this move was }
end;
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.Pen.Mode := pmNotXor;{ use XOR mode to draw/erase }
   Canvas.MoveTo(Origin.X, Origin.Y);{ move pen back to origin }
   Canvas.LineTo(MovePt.X, MovePt.Y);{ erase the old line }
   Canvas.MoveTo(Origin.X, Origin.Y);{ start at origin again }
   Canvas.LineTo(X, Y);{ draw the new line }
  end;
 MovePt := Point(X, Y);{ record point for next move }
 Canvas.Pen.Mode := pmCopy;
end;
```

Now you get a “rubber band” effect when you draw the line. By changing the pen’s mode to *pmNotXor*, you have it combine your line with the background pixels. When you go to erase the line, you’re actually setting the pixels back to the way they were. By changing the pen mode back to *pmCopy* (its default value) after drawing the lines, you ensure that the pen is ready to do its final drawing when you release the mouse button.

Chapter

9

Writing multi-threaded applications

CLX provides several objects that make writing multi-threaded applications easier. Multi-threaded applications are applications that include several simultaneous paths of execution. While using multiple threads requires careful thought, it can enhance your programs by

- **Avoiding bottlenecks.** With only one thread, a program must stop all execution when waiting for slow processes such as accessing files on disk, communicating with other machines, or displaying multimedia content. The CPU sits idle until the process completes. With multiple threads, your application can continue execution in separate threads while one thread waits for the results of a slow process.
- **Organizing program behavior.** Often, a program's behavior can be organized into several parallel processes that function independently. Use threads to launch a single section of code simultaneously for each of these parallel cases. Use threads to assign priorities to various program tasks so that you can give more CPU time to more critical tasks.
- **Multiprocessing.** If the system running your program has multiple processors, you can improve performance by dividing the work into several threads and letting them run simultaneously on separate processors.

Note Linux is a multiprocessing operating system with Intel MP architecture. Processes are separate tasks each with their own rights and responsibilities. Each individual process runs in its own virtual address space and is not capable of interacting with another process except through secure, kernel-managed mechanisms.

Defining thread objects

For most applications, you can use a thread object to represent an execution thread in your application. Thread objects simplify writing multi-threaded applications by encapsulating the most commonly needed uses of threads.

Thread objects do not allow you to control the security attributes or stack size of your threads. If you need to control these, you must use the *BeginThread* function. Even when using *BeginThread*, you can still benefit from some of the thread synchronization objects and methods described in “Coordinating threads” on page 9-6. For more information on using *BeginThread*, see the online help.

To use a thread object in your application, you must create a new descendant of *TThread*. To create a descendant of *TThread*, choose File | New from the main menu. In the new objects dialog box, select Thread Object. You are prompted to provide a class name for your new thread object. After you provide the name, Kylix creates a new unit file to implement the thread.

Note Unlike most dialog boxes in the IDE that require a class name, the New Thread Object dialog does not automatically prepend a ‘T’ to the front of the class name you provide.

The automatically generated unit file contains the skeleton code for your new thread object. If you named your thread *TMyThread*, it would look like the following:

```
unit Unit2;
interface
uses
  Classes;
type
  TMyThread = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;
implementation
{ TMyThread }
procedure TMyThread.Execute;
begin
  { Place thread code here }
end;
end.
```

You must fill in the code for the *Execute* method. These steps are described in the following sections.

Initializing the thread

If you want to write initialization code for your new thread class, you must override the Create method. Add a new constructor to the declaration of your thread class and write the initialization code as its implementation. This is where you can assign a default priority for your thread and indicate whether it should be freed automatically when it finishes executing.

Assigning a default priority

Priority indicates how much preference the thread gets when the operating system schedules CPU time among all the threads in your application. Use a high priority thread to handle time critical tasks, and a low priority thread to perform other tasks. To indicate the priority of your thread object, set the Priority property.

Priority values are integers in Linux where a lower number indicates a higher priority.

Warning Boosting the thread priority of a CPU intensive operation may "starve" other threads in the application. Only apply priority boosts to threads that spend most of their time waiting for external events.

The following code shows the constructor of a low-priority thread that performs background tasks which should not interfere with the rest of the application's performance:

```
constructor TMyThread.Create(CreateSuspended: Boolean);
begin
  inherited Create(CreateSuspended);
  Priority := tpIdle;
end;
```

Indicating when threads are freed

Usually, when threads finish their operation, they can simply be freed. In this case, it is easiest to let the thread object free itself. To do this, set the *FreeOnTerminate* property to *True*.

There are times, however, when the termination of a thread must be coordinated with other threads. For example, you may be waiting for one thread to return a value before performing an action in another thread. To do this, you do not want to free the first thread until the second has received the return value. You can handle this situation by setting *FreeOnTerminate* to *False* and then explicitly freeing the first thread from the second.

Writing the thread function

The *Execute* method is your thread function. You can think of it as a program that is launched by your application, except that it shares the same process space. Writing the thread function is a little trickier than writing a separate program because you must make sure that you don't overwrite memory that is used by other threads in your application. On the other hand, because the thread shares the same process space with other threads, you can use the shared memory to communicate between threads.

Using the main CLX thread

When you use objects from the CLX object hierarchy, their properties and methods are not guaranteed to be thread-safe. That is, accessing properties or executing methods may perform some actions that use memory which is not protected from the actions of other threads. Because of this, a main CLX thread is set aside for access of CLX objects.

If all objects access their properties and execute their methods within this single thread, you need not worry about your objects interfering with each other. To use the main CLX thread, create a separate routine that performs the required actions. Call this separate routine from within your thread's *Synchronize* method. For example:

```
procedure TMyThread.PushTheButton;
begin
  Button1.Click;
end;
⋮
procedure TMyThread.Execute;
begin
  ⋮
  Synchronize(PushTheButton);
  ⋮
end;
```

Synchronize waits for the main CLX thread to enter the message loop and then executes the passed method.

Note Because *Synchronize* uses the message loop, it does not work in console applications. You must use other mechanisms, such as critical sections, to protect access to CLX objects in console applications.

You do not always need to use the main CLX thread. Some objects are thread-aware. Omitting the use of the *Synchronize* method when you know an object's methods are thread-safe will improve performance because you don't need to wait for the CLX thread to enter its message loop. You do not need to use the *Synchronize* method in the following situations:

- Data access components are thread-safe as follows: for dbDirect, as long as the vendor client library is thread-safe, the dbDirect components will be thread-safe.

  When using data access components, you must wrap all calls that involve data-aware controls in the *Synchronize* method. Thus, for example, you need to synchronize calls that link a data control to a dataset by setting the *DataSet* property of the data source object, but you don't need to synchronize to access the data in a field of the dataset.
- VisualCLX objects are not thread-safe.
- DataCLX objects are thread-safe.
- Graphics objects are thread-safe. You do not need to use the main CLX thread to access *TFont*, *TPen*, *TBrush, TBitmap, TDrawing,* or *TIcon*. Canvas objects can be used outside the *Synchronize* method by locking them (see "Locking objects" on page 9-6).
- While list objects are not thread-safe, you can use a thread-safe version, *TThreadList*, instead of *TList*.

Call the *CheckSynchronize* routine periodically within the main thread of your application so that background threads can synchronize their execution with the main thread. The best place to call *CheckSynchronize* is when the application is idle (for example, from an *OnIdle* event handler). This ensures that it is safe to make method calls in the background thread.

Using thread-local variables

Your *Execute* method and any of the routines it calls have their own local variables, just like any other Object Pascal routines. These routines also can access any global variables. In fact, global variables provide a powerful mechanism for communicating between threads.

Sometimes, however, you may want to use variables that are global to all the routines running in your thread, but not shared with other instances of the same thread class. You can do this by declaring thread-local variables. Make a variable thread-local by declaring it in a **threadvar** section. For example,

```
threadvar
    x : integer;
```

declares an integer type variable that is private to each thread in the application, but global within each thread.

The threadvar section can only be used for global variables. Pointer and Function variables can't be thread variables. Types that use copy-on-write semantics, such as long strings don't work as thread variables either.

Checking for termination by other threads

Your thread begins running when the *Execute* method is called (see "Executing thread objects" on page 9-10) and continues until *Execute* finishes. This reflects the model that the thread performs a specific task, and then stops when it is finished. Sometimes, however, an application needs a thread to execute until some external criterion is satisfied.

You can allow other threads to signal that it is time for your thread to finish executing by checking the *Terminated* property. When another thread tries to terminate your thread, it calls the *Terminate* method. *Terminate* sets your thread's *Terminated* property to *True*. It is up to your *Execute* method to implement the *Terminate* method by checking and responding to the *Terminated* property. The following example shows one way to do this:

```
procedure TMyThread.Execute;
begin
  while not Terminated do
    PerformSomeTask;
end;
```

Handling exceptions in the thread function

The *Execute* method must catch all exceptions that occur in the thread. If you fail to catch an exception in your thread function, your application can cause access violations. This may not be obvious when you are developing your application, because the IDE catches the exception, but when you run your application outside of the debugger, the exception will cause a runtime error and the application will stop running.

To catch the exceptions that occur inside your thread function, add a **try**...**except** block to the implementation of the *Execute* method:

```
procedure TMyThread.Execute;
```

```
begin
  try
    while not Terminated do
      PerformSomeTask;
  except
    { do something with exceptions }
  end;
end;
```

Writing clean-up code

You can centralize the code that cleans up when your thread finishes executing. Just before a thread shuts down, an *OnTerminate* event occurs. Put any clean-up code in the *OnTerminate* event handler to ensure that it is always executed, no matter what execution path the *Execute* method follows.

The *OnTerminate* event handler is not run as part of your thread. Instead, it is run in the context of the main CLX thread of your application. This has two implications:

- You can't use any thread-local variables in an *OnTerminate* event handler (unless you want the main CLX thread values).
- You can safely access any components and CLX objects from the *OnTerminate* event handler without worrying about clashing with other threads.

For more information about the main CLX thread, see "Using the main CLX thread" on page 9-3.

Coordinating threads

When writing the code that runs when your thread is executed, you must consider the behavior of other threads that may be executing simultaneously. In particular, care must be taken to avoid two threads trying to use the same global object or variable at the same time. In addition, the code in one thread can depend on the results of tasks performed by other threads.

Avoiding simultaneous access

To avoid clashing with other threads when accessing global objects or variables, you may need to block the execution of other threads until your thread code has finished an operation. Be careful not to block other execution threads unnecessarily. Doing so can cause performance to degrade seriously and negate most of the advantages of using multiple threads.

Locking objects

Some objects have built-in locking that prevents the execution of other threads from using that object instance.

For example, canvas objects (*TCanvas* and descendants) have a *Lock* method that prevents other threads from accessing the canvas until the *Unlock* method is called.

CLX also includes a thread-safe list object, *TThreadList*. Calling *TThreadList.LockList* returns the list object while also blocking other execution threads from using the list until the *UnlockList* method is called. Calls to *TCanvas.Lock* or *TThreadList.LockList* can be safely nested. The lock is not released until the last locking call is matched with a corresponding unlock call in the same thread.

Using critical sections

If objects do not provide built-in locking, you can use a critical section. Critical sections work like gates that allow only a single thread to enter at a time. To use a critical section, create a global instance of *TCriticalSection. TCriticalSection* has two methods, *Acquire* (which blocks other threads from executing the section) and *Release* (which removes the block).

Each critical section is associated with the global memory you want to protect. Every thread that accesses that global memory should first use the *Acquire* method to ensure that no other thread is using it. When finished, threads call the *Release* method so that other threads can access the global memory by calling *Acquire*.

Warning Critical sections only work if every thread uses them to access the associated global memory. Threads that ignore the critical section and access the global memory without calling *Acquire* can introduce problems of simultaneous access.

For example, consider an application that has a global critical section variable, *LockXY*, that blocks access to global variables X and Y. Any thread that uses X or Y must surround that use with calls to the critical section such as the following:

```
LockXY.Acquire; { lock out other threads }
try
  Y := sin(X);
finally
  LockXY.Release;
end;
```

Using the multi-read exclusive-write synchronizer

When you use critical sections to protect global memory, only one thread can use the memory at a time. This can be more protection than you need, especially if you have an object or variable that must be read often but to which you very seldom write. There is no danger in multiple threads reading the same memory simultaneously, as long as no thread is writing to it.

When you have some global memory that is read often, but to which threads occasionally write, you can protect it using *TMultiReadExclusiveWriteSynchronizer*. This object acts like a critical section, but allows multiple threads to read the memory it protects as long as no thread is writing to it. Threads must have exclusive access to write to memory protected by *TMultiReadExclusiveWriteSynchronizer*.

To use a multi-read exclusive-write synchronizer, create a global instance of *TMultiReadExclusiveWriteSynchronizer* that is associated with the global memory you want to protect. Every thread that reads from this memory must first call the

BeginRead method. *BeginRead* ensures that no other thread is currently writing to the memory. When a thread finishes reading the protected memory, it calls the *EndRead* method. Any thread that writes to the protected memory must call *BeginWrite* first. *BeginWrite* ensures that no other thread is currently reading or writing to the memory. When a thread finishes writing to the protected memory, it calls the *EndWrite* method, so that threads waiting to read the memory can begin.

Warning Like critical sections, the multi-read exclusive-write synchronizer only works if every thread uses it to access the associated global memory. Threads that ignore the synchronizer and access the global memory without calling *BeginRead* or *BeginWrite* introduce problems of simultaneous access.

Other techniques for sharing memory

When using objects in CLX, use the main CLX thread to execute your code. Using the main CLX thread ensures that the object does not indirectly access any memory that is also used by CLX objects in other threads. See “Using the main CLX thread” on page 9-3 for more information on the main CLX thread.

If the global memory does not need to be shared by multiple threads, consider using thread-local variables instead of global variables. By using thread-local variables, your thread does not need to wait for or lock out any other threads. See “Using thread-local variables” on page 9-5 for more information about thread-local variables.

Waiting for other threads

If your thread must wait for another thread to finish some task, you can tell your thread to temporarily suspend execution. You can either wait for another thread to completely finish executing, or you can wait for another thread to signal that it has completed a task.

Waiting for a thread to finish executing

To wait for another thread to finish executing, use the *WaitFor* method of that other thread. *WaitFor* doesn’t return until the other thread terminates, either by finishing its own *Execute* method or by terminating due to an exception. For example, the following code waits until another thread fills a thread list object before accessing the objects in the list:

```
if ListFillingThread.WaitFor then
begin
  with ThreadList1.LockList do
  begin
    for I := 0 to Count - 1 do
      ProcessItem(Items[I]);
  end;
  ThreadList1.UnlockList;
end;
```

In the previous example, the list items were only accessed when the *WaitFor* method indicated that the list was successfully filled. This return value must be assigned by the *Execute* method of the thread that was waited for. However, because threads that call *WaitFor* want to know the result of thread execution, not code that calls *Execute*, the *Execute* method does not return any value. Instead, the *Execute* method sets the *ReturnValue* property. *ReturnValue* is then returned by the *WaitFor* method when it is called by other threads. Return values are integers. Your application determines their meaning.

Waiting for a task to be completed

Sometimes, you need to wait for a thread to finish some operation rather than waiting for a particular thread to complete execution. To do this, use an event object. Event objects (*TEvent*) should be created with global scope so that they can act like signals that are visible to all threads.

When a thread completes an operation that other threads depend on, it calls *TEvent.SetEvent*. *SetEvent* turns on the signal, so any other thread that checks will know that the operation has completed. To turn off the signal, use the *ResetEvent* method.

For example, consider a situation where you must wait for several threads to complete their execution rather than a single thread. Because you don’t know which thread will finish last, you can’t simply use the *WaitFor* method of one of the threads. Instead, you can have each thread increment a counter when it is finished, and have the last thread signal that they are all done by setting an event.

The following code shows the end of the *OnTerminate* event handler for all of the threads that must complete. *CounterGuard* is a global critical section object that prevents multiple threads from using the counter at the same time. *Counter* is a global variable that counts the number of threads that have completed.

```
procedure TDataModule.TaskThreadTerminate(Sender: TObject);
begin
  ⋮
  CounterGuard.Acquire; { obtain a lock on the counter }
  Dec(Counter);   { decrement the global counter variable }
  if Counter = 0 then
    Event1.SetEvent; { signal if this is the last thread }
  CounterGuard.Release; { release the lock on the counter }
  ⋮
end;
```

The main thread initializes the Counter variable, launches the task threads, and waits for the signal that they are all done by calling the *WaitFor* method. *WaitFor* waits for a specified time period for the signal to be set, and returns one of the values from Table 9.1.

Table 9.1 WaitFor return values

Value	Meaning
wrSignaled	The signal of the event was set.
wrTimeout	The specified time elapsed without the signal being set.
wrAbandoned	The event object was destroyed before the timeout period elapsed.
wrError	An error occurred while waiting.

The following shows how the main thread launches the task threads and then resumes when they have all completed:

```
Event1.ResetEvent; { clear the event before launching the threads }
for i := 1 to Counter do
  TaskThread.Create(False); { create and launch task threads }
if Event1.WaitFor(20000) <> wrSignaled then
  raise Exception;
{ now continue with the main thread. All task threads have finished }
```

Note If you do not want to stop waiting for an event after a specified time period, pass the *WaitFor* method a parameter value of INFINITE. Be careful when using INFINITE, because your thread will hang if the anticipated signal is never received.

Executing thread objects

Once you have implemented a thread class by giving it an *Execute* method, you can use it in your application to launch the code in the *Execute* method. To use a thread, first create an instance of the thread class. You can create a thread instance that starts running immediately, or you can create your thread in a suspended state so that it only begins when you call the *Resume* method. To create a thread so that it starts up immediately, set the constructor's *CreateSuspended* parameter to *False***.** For example, the following line creates a thread and starts its execution:

```
SecondProcess := TMyThread.Create(false); {create and run the thread }
```

Warning Do not create too many threads in your application. The overhead in managing multiple threads can impact performance. The recommended limit is 16 threads per process on single processor systems. This limit assumes that most of those threads are waiting for external events. If all threads are active, you will want to use fewer.

You can create multiple instances of the same thread type to execute parallel code. For example, you can launch a new instance of a thread in response to some user action, allowing each thread to perform the expected response.

Overriding the default priority

When the amount of CPU time the thread should receive is implicit in the thread's task, its priority is set in the constructor. This is described in "Initializing the thread" on page 9-2. However, if the thread priority varies depending on when the thread is executed, create the thread in a suspended state, set the priority, and then start the thread running:

```
SecondProcess :=  TMyThread.Create(True); { create but don't run }
SecondProcess.Priority := tpLower; { set the priority lower than normal }
SecondProcess.Resume; { now run the thread }
```

Starting and stopping threads

A thread can be started and stopped any number of times before it finishes executing. To stop a thread temporarily, call its *Suspend* method. When it is safe for the thread to resume, call its *Resume* method. *Suspend* increases an internal counter, so you can nest calls to *Suspend* and *Resume*. The thread does not resume execution until all suspensions have been matched by a call to *Resume*.

You can request that a thread end execution prematurely by calling the *Terminate* method. *Terminate* sets the thread's *Terminated* property to *True*. If you have implemented the *Execute* method properly, it checks the *Terminated* property periodically, and stops execution when *Terminated* is *True*.

Debugging multi-threaded applications

When debugging multi-threaded applications, it can be confusing trying to keep track of the status of all the threads that are executing simultaneously, or even to determine which thread is executing when you stop at a breakpoint. You can use the Thread Status box to help you keep track of and manipulate all the threads in your application. To display the Thread status box, choose View | Threads from the main menu.

When a debug event occurs (breakpoint, exception, paused), the Thread Status view indicates the status of each thread. Right-click the Thread Status box to access commands that locate the corresponding source location or make a different thread current. When a thread is marked as current, the next step or run operation is relative to that thread.

The Thread Status box lists all your application's execution threads by their thread ID. If you are using thread objects, the thread ID is the value of the *ThreadID* property. If you are not using thread objects, the thread ID for each thread is returned by the call to *BeginThread*.

For additional details on the Thread Status box, see online Help.

Chapter

10

Developing cross-platform applications

You can develop cross-platform 32-bit applications that run on both the Linux and Windows operating systems. To do this, you can start with an existing Windows application and modify it, or you can create a new application by following the recommended practices for writing platform-independent code.

This chapter describes how to port Kylix Windows applications to Linux and includes information on the differences between developing applications on Windows and Linux. It also provides guidelines for writing code that is portable between the different environments.

Note Most applications developed for Linux (with no Linux-specific API calls) will run on Linux and, if recompiled, can run on Windows when CLX is available on Windows.

Porting Windows applications to Linux

If you have Kylix applications that were written for the Windows environment, you can port them to a Linux environment. How easy it will be depends on the nature and complexity of the application and how many Windows dependencies there are.

The following sections describe some of the major differences between the Windows and Linux environments and provide guidelines on how to get started porting an application.

Porting techniques

The following are different approaches you can take to port an application from one platform to another:

Table 10.1 Porting techniques

Technique	Description
Platform-specific port	Targets an operating system and underlying APIs
Cross-platform port	Targets a cross-platform API
Windows emulation	Leave the code alone and port the API it uses

Platform-specific ports

Platform-specific ports tend to be time-consuming, expensive, and only produce a single targeted result. They create different code bases, which makes them particularly difficult to maintain. However, each port is designed for the specific operating system and can take advantage of platform-specific functionality. So, the application typically runs faster.

Cross-platform ports

Cross-platform ports generally provide the quickest technique and the ported applications target multiple platforms. In reality, the amount of work involved in developing cross-platform applications is highly dependent on the existing code. If code has been developed without regard for platform independence, you may run into scenarios where platform-independent "logic" and platform-dependent "implementation" are mixed together.

The cross-platform approach is the preferable approach because business logic is expressed in platform-independent terms. Some services are abstracted behind an internal interface that looks the same on all platforms, but has a specific implementation on each. Kylix's runtime library is an example of this: The interface is very similar on both platforms, although the implementation may be vastly different. You should separate cross-platform parts, then implement specific services on top. In the end, this approach is the least expensive solution, because of reduced maintenance costs due to a largely shared source base and an improved application architecture.

Windows emulation ports

Windows emulation is the most complex method and it can be very costly, but the resulting Linux application will look most similar to an existing Windows application. This approach involves implementing Windows functionality on Linux. From an engineering point of view, this is solution is very hard to maintain.

Porting your application

If you are porting an application to Linux that you want to run on Linux only, you may choose to remove Windows-specific features entirely. If, however, you are porting an application that you want to run on both platforms, you need to modify your code or use **$IFDEF**s to indicate sections of the code that apply specifically to Windows or Linux.

Follow these general steps to port your Windows application to Linux:

1 Move your Kylix Windows application source files and other project-related files onto your Linux computer. (You can share source files between Linux and Windows if you want the program to run on both platforms. Or you can transfer the files using a tool such as ftp using the ASCII mode.)

   Source files should include your unit files (.pas files), project file (.dpr file), and any package files (.dpk files). Project-related files include form files (.dfm files), resource files (.res files), and project options files (.dof files). If you want to compile your application from the command line only (rather than using the IDE), you'll need the configuration file (.cfg file).

2 If you plan to single-source the application for use on both Windows and Linux, copy .dfm files to .xfm files of the same name (for example, rename unit1.dfm to unit1.xfm). Rename (or **$IFDEF**) the reference to the .dfm file in the unit file(s) from `{$R *.dfm}` to `{$R *.xfm}`. (The .dfm file will work on Kylix but will be altered so it may not work on Delphi.)

3 Change (or **$IFDEF**) all **uses** clauses so they refer to the correct units in Kylix. (See "Kylix and Delphi unit comparison" on page 10-8 for information.)

4 Rewrite any code that does not require Windows dependencies making the code more platform-independent. Do this using the runtime library routines and constants. (See "Writing portable code" on page 10-15 for information.)

5 Find equivalent functionality for features that are different on Linux. Use **$IFDEF**s (sparingly) to delimit Windows-specific information. (See "Using conditional directives" on page 10-16 for information.)

   For example, you can **$IFDEF** platform-specific code in your source files:

```
[$IFDEF MSWINDOWS]
IniFile.LoadfromFile('c:\x.txt');
[$ENDIF]

[$IFDEF LINUX]
IniFile.LoadfromFile('/home/name/x.txt');
[$ENDIF]
```

6 Search for references to pathnames in all the project files.

   - Pathnames in Linux use a forward slash / as a delimiter (for example, /usr/lib) and files may be located in different directories on the Linux system. Use the PathDelim constant (in SysUtils) to specify the path delimiter that is appropriate for the system. Determine the correct location for any files on Linux.
   - Change references to drive letters (for example, C:\) and code that looks for drive letters by looking for a colon at position 2 in the string. Use the DriveDelim constant (in SysUtils) to specify the location in terms that are appropriate for the system.

- In places where you specify multiple paths, change the path separator from semicolon (;) to colon (:). Use the PathSep constant (in SysUtils) to specify the path separator that is appropriate for the system.
- Because file names are case-sensitive in Linux, make sure that your application doesn't change the case of file names or assume a certain case.

7 Compile the project on Linux. Review any error messages to see where additional changes need to be made.

CLX versus VCL

Kylix uses the Borland Component Library for Cross Platform (CLX) in place of the Visual Component Library (VCL). Within the VCL, many controls provide an easy way to access Windows controls. Similarly, CLX provides access to Qt widgets (from window + gadget) in the Qt shared libraries.

CLX looks much like the VCL. Most of the component names are the same, many properties have the same names. In addition, CLX, as well as the VCL, will be available on Windows (check the latest release of Delphi to determine availability).

CLX components can be grouped into the following parts:

Table 10.2 CLX parts

Part	Description
VisualCLX	Native cross-platform GUI components and graphics. The components in this area may differ on Linux and Windows.
DataCLX	Client data-access components. The components in this area are a subset of the local, client/server, and n-tier based on client datasets. The code is the same on Linux and Windows.
NetCLX	Internet components including Apache DSO and CGI WebBroker. These are the same on Linux and Windows.
BaseCLX	Runtime Library up to and including Classes.pas. The code is the same on Linux and Windows.

Widgets in VisualCLX replace Windows controls. In CLX, *TWidgetControl* replaces the VCL's *TWinControl*. Other components (such as *TScrollingWidget*) have corresponding names. However, you do not need to change occurrences of *TWinControl* to *TWidgetControl*. Type declarations, such as the following

```
TWinControl = TWidgetControl;
```

appear in the QControls.pas source file to simplify sharing of source code. *TWidgetControl* and its descendants all have a *Handle* property that is a reference to the Qt object; and a *Hooks* property, which is a reference to the hook objects that handle the event mechanism.

Unit names and locations of some classes are different for CLX. You will need to modify **uses** clauses to eliminate references to units that don't exist on Kylix and to change the names to Kylix units. (Most project files and the interface sections of most units contain a **uses** clause. The implementation section of a unit can also contain its own **uses** clause.)

What CLX does differently

Although much of CLX is implemented so that it is consistent with the VCL, some features are implemented differently. This section provides an overview of some of the differences between CLX and VCL implementations.

Look and feel

The visual environment in Linux looks somewhat different than it does in Windows. The look of dialogs may differ depending on which window manager is in use (for example, if using KDE or Gnome).

Styles

Application-wide “styles” can be used in addition to the *OwnerDraw* properties. You can use the *TApplication.Style* property to specify the look and feel of an application's graphical elements. Using styles, a widget or an application can take on a whole new look. You can still use owner draw on Linux but using styles is recommended.

Variants

All of the variant/safe array code that was in System is now in two new units:

- Variants.pas
- VarUtils.pas

The operating system dependent code is now isolated in VarUtils.pas, and it also contains generic versions of everything needed by Variants.pas. If you are porting code from Windows that included Windows calls, you need to replace these calls to calls into VarUtils.pas.

If you want to use variants, you must include the Variants unit to your **uses** clause.

VarIsEmpty does a simple test against *varEmpty* to see if a variant is clear, and on Linux you need to use the *VarIsClear* function to clear a variant.

Custom variant data handler

You can define custom data types for variants. This introduces operator overloading while the type is assigned to the variant. To create a new variant type, descend from the class, *TCustomVariantType*, and instantiate your new variant type.

For an example, see VarCmplx.pas. This unit implements complex mathematics support via custom variants. It supports the following variant operations: addition, subtraction, multiplication, division (not integer division), and negation. It also handles conversion to and from: SmallInt, Integer, Single, Double, Currency, Date, Boolean, Byte, OleStr, and String. Any of the float/ordinal conversion will lose any imaginary portion of the complex value.

No registry

Linux does not use a registry to store configuration information. Instead, you use text configuration files and environment variables instead of using the registry. System configuration files on Linux are often located in /etc, for example, /etc/hosts. Other user profiles are located in hidden files (preceded with a dot), such as .bashrc, which holds bash shell settings or .XDefaults, which is used to set defaults for X programs.

Registry-dependent code may be changed to using a local configuration text file instead stored, for example, in the same directory as the application. Writing a unit with all the registry functions but diverting all output to a local configuration file is one way you could handle a former dependency on the registry.

To place information in a global location, you could store a global configuration file in the root directory. This makes it so all of your applications can access the same configuration file. However, you must be sure that the file permissions and access rights are set up correctly.

You can also use ini files as in Windows. However, in Kylix, you need to use *TMemIniFile* instead of *TRegIniFile.*

Other differences

Kylix implementation also has some other differences that affect the way your application works. This section describes some of those differences.

ToggleButton doesn't get toggled by the Enter key. Pressing Enter doesn't simulate a click event on Kylix as it does in Delphi.

TColorDialog does not have a *TColorDialog.Options* property to set. Therefore, you cannot customize the appearance and functionality of the color selection dialog. Also, *TColordialog* is not always modal. You can manipulate the title bar of an application with a modal dialog on Kylix (that is, you can select the parent form of the color dialog and do things like maximizing it while the color dialog is open).

At runtime, combo boxes work differently on Kylix than they do in Delphi. On Kylix (but not on Delphi), you can add a item to a drop down by entering text and pressing Enter in the edit field of a combo box. You can turn this feature off by setting *InsertMode* to ciNone. It is also possible to add empty (no string) items to the list in the combo box. Also, if you keep pressing the down arrow key, it does not stop at the last item of the combo box list. It cycles around to the top again.

TCustomEdit does not implement *Undo*, *ClearUndo*, or *CanUndo*. So there is no way to programmatically undo edits. But application users can undo their edits in an edit box (*TEdit*) at runtime by right-clicking on the edit box and choosing the Undo command.

Missing in CLX

The following general features are missing in CLX:

- Bi-directional properties (*BidiMode*) for right-to-left text output or input

- Generic bevel properties on common controls (note that some objects still have bevel properties)
- Docking properties and methods
- Backward compatibility features such components on the Win3.1 tab and *Ctl3D*
- *DragCursor* and *DragKind* (but drag and drop is included)

Features that will not port

Some Windows-specific features supported on the Windows version of Kylix will not transport directly to Linux environments. Features, such as COM, ActiveX, OLE, BDE, and ADO are dependent on Windows technology and are not available in Kylix. The following table lists features that are different on the two platforms and lists the equivalent Kylix feature, if one is available.

Table 10.3 Changed or different features

Delphi/Windows feature	Kylix/Linux feature
Windows API calls	CLX methods, Qt calls, libc calls, or calls to other system libraries
COM components (including ActiveX)	Not available
ADO components	Regular database components available
Windows messaging	Qt events
Winsock	BSD sockets
Messaging Application Programming Interface (MAPI) includes a standard library of Windows messaging functions.	SMTP/POP3 let you send, receive, and save email messages
Legacy components (such as items on the Win 3.1 component palette tab)	Not available

The Kylix equivalent of Windows DLLs are shared object libraries (.so files), which contain position-independent code (PIC). This has the following consequences:

- Variables referring to an absolute address in memory (using the **absolute** directive) are not allowed.
- Global memory references and calls to external functions are made relative to the EBX register, which must be preserved across calls.

You only need to worry about global memory references and calls to external functions if using assembler—Kylix generates the correct code. (For information, see “Including inline assembler code” on page 10-18.)

Kylix library modules and packages are implemented using .so files.

Kylix and Delphi unit comparison

All of the objects in the VCL or CLX are defined in unit files (.pas source files). For example, you can find the implementation of *TObject* in the System unit, and the Classes unit defines the base *TComponent* class. When you drop an object onto a form or use an object within your application, the name of the unit is added to the **uses** clause which tells the compiler which units to link into the project.

This section provides tables that list the units that are in Kylix and the comparable unit in Delphi, list the units that are in Kylix only, and list the units that are in Delphi only. If a unit contained in Kylix will appear in future version of Delphi, that is stated in the Delphi unit column.

The following table lists Delphi units and the comparable Kylix units:

Table 10.4 Units in Kylix and Delphi

Delphi units	Kylix units
ActnList	QActnList
Buttons	QButtons
CheckLst	QCheckLst
Classes	Classes
Clipbrd	QClipbrd
ComCtrls	QComCtrls
Consts	Consts, QConsts, and RTLConsts
Contnrs	Contnrs
Controls	QControls
ConvUtils (future Delphi)	ConvUtils
DateUtils (future Delphi)	DateUtils
DB	DB
DBActns	QDBActns
DBClient	DBClient
DBCommon	DBCommon
DBConnAdmin (future Delphi)	DBConnAdmin
DBConsts	DBConsts
DBCtrls	QDBCtrls
DBGrids	QDBGrids
DBLocal	DBLocal
DBLocalS	DBLocalS
DBLogDlg	DBLogDlg
DBXpress	DBXpress
Dialogs	QDialogs
DSIntf	DSIntf

Table 10.4 Units in Kylix and Delphi (continued)

Delphi units	Kylix units
ExtCtrls	QExtCtrls
FMTBCD (future Delphi)	FMTBCD
Forms	QForms
Graphics	QGraphics
Grids	QGrids
HelpIntfs	HelpIntfs
ImgList	QImgList
IniFiles	IniFiles
Mask	QMask
MaskUtils	MaskUtils
Masks	Masks
Math	Math
Menus	QMenus
Midas	Midas
MidConst	MidConst
Printers	QPrinters
Provider	Provider
Qt (future Delphi)	Qt
Search (future Delphi)	QSearch
Sockets (future Delphi)	Sockets
StdActns	QStdActns
StdCtrls	QStdCtrls
SqlConst	SqlConst
SqlExpr	SqlExpr
SqlTimSt	SqlTimSt
StdConvs (future Delphi)	StdConvs
SyncObjs	SyncObjs
SysConst	SysConst
SysInit	SysInit
System	System
SysUtils	SysUtils
Types	Types and QTypes
TypInfo	TypInfo
VarCmplx	VarCmplx
Variants (future Delphi)	Variants
VarUtils (future Delphi)	VarUtils

The following units are in Kylix but not in Windows:

Table 10.5 Units in Kylix, not in Delphi

Unit	Description
DirSel	Directory selection
QStyle	GUI look and feel

The following Windows units are not included in Kylix mostly because they concern Windows-specific features that are not available on Linux such as ADO, COM, and the Borland Database Engine. The reason for the unit's exclusion is listed.

Table 10.6 Units in Delphi, not in Kylix

Unit	Reason for exclusion
ADOConst	No ADO feature
ADODB	No ADO feature
AppEvnts	No TApplicationEvent object
AxCtrls	No COM feature
BdeConst	No BDE feature
ComStrs	No COM feature
CorbaCon	No Corba feature
CorbaStd	No Corba feature
CorbaVCL	No Corba feature
CtlPanel	No Windows Control Panel support
DataBkr	May appear later in upsell
DBCGrids	No BDE feature
DBExcept	No BDE feature
DBInpReq	No BDE feature
DBLookup	Obsolete
DbOleCtl	No COM feature
DBPWDlg	No BDE feature
DBTables	No BDE feature
DdeMan	No DDE feature
DRTable	No BDE feature
ExtDlgs	No picture dialogs
FileCtrl	Obsolete
MConnect	No COM feature
Messages	Windows-specific area
MidasCon	Obsolete
MPlayer	Windows-specific media player
Mtsobj	No COM feature
MtsRdm	No COM feature
Mtx	No COM feature
mxConsts	No COM feature

Table 10.6 Units in Delphi, not in Kylix (continued)

Unit	Reason for exclusion
ObjBrkr	May appear later in upsell
OleConstMay	No COM feature
OleCtnrs	No COM feature
OleCtrls	No COM feature
OLEDB	No COM feature
OleServer	No COM feature
Outline	Obsolete
Registry	Windows-specific registry support
ScktCnst	Replaced by Sockets
ScktComp	Replaced by Sockets
SConnect	Unsupported connection protocols
SvcMgr	NT Services support
Tabnotbk	Obsolete
Tabs	Obsolete
ToolWin	No docking feature
VCLCom	No COM feature
WebConst	Windows-specific constants
Windows	Windows-specific virtual key codes

Differences in CLX object constructors

When a CLX object is created, either implicitly in the Forms Designer by placing that object on the form or explicitly in code by using the *Create* method of the object, an instance of the underlying associated widget is created also. The instance of the widget is owned by this CLX object. When the CLX object is deleted by calling the *Free* method or automatically deleted by the CLX object's parent container, the underlying widget is also deleted. This is the same type of functionality that you see in the VCL in Windows applications.

When you explicitly create a CLX object in code, by using the *Create(AHandle)* method of the object, you are passing the instance of an existing Qt widget to the CLX object to use during its construction. It is important to note that this CLX object does not own the Qt widget that is passed to it. Therefore, when you call the *Free* method after creating the object in this manner, only the CLX object is destroyed and not the underlying Qt widget instance. This is different from the VCL.

Some CLX objects let you assume ownership of the underlying widget using the *OwnHandle* method. After calling *OwnHandle*, if you delete the CLX object, the underlying widget is destroyed as well.

Sharing source files between Windows and Linux

If you want your application to run on both Windows and Linux, you can share the source files making them accessible to both operating systems. You can do this many ways such as placing the source files on a server that is accessible to both computers or by using Samba on the Linux machine to provide access to files through Microsoft network file sharing for both Linux and Windows. You can choose to keep the source on Linux and create a shared drive on Linux. Or you can keep the source on Windows and create a share on Windows for the Linux machine to access.

You can continue to develop and compile the file on Kylix using objects that are supported by both VCL and CLX. When you are finished, you can compile on both Linux and Windows.

Form files (.dfm files in Delphi) are called .xfm files in Kylix. If you want to single-source your code, you should copy the .dfm from Windows to an .xfm on Linux, maintaining both files. Otherwise, the .dfm file will be modified on Linux and may no longer work on Windows. If you plan to write cross-platform applications, the .xfm will work on versions of Delphi that support CLX.

Environmental differences between Windows and Linux

The following table lists some of the differences on Linux you need to be aware of if you're used to working in the Windows environment.

Table 10.7 Differences in the Linux operating environment

Difference	Description
File name case sensitivity	In Linux, a capital letter is *not* the same as a lowercase letter. The file Test.txt is *not* the same file as test.txt. You need to pay close attention to capitalization of file names on Linux.
Line ending characters	On Windows, lines of text are terminated by CR/LF (that is, ASCII 13 + ASCII 10), but on Linux it is LF. While the code editor in Kylix can handle the difference, you should be aware of this when importing code from Windows.
End of file character	In DOS and Windows, the character value #26 (Ctrl-Z) is treated as the end of the text file, even if there is data in the file after that character. Linux has no special end of file character; the text data ends at the end of the file.
Batch files/shell scripts	The Linux equivalent of .bat files are shell scripts. A script is a text file containing instructions, saved and made executable with the command, `chmod +x <scriptfile>`. To execute it, type its name. (The scripting language depends on the shell you are using on Linux. Bash is commonly used.)
Command confirmation	In DOS or Windows, if you try to delete a file or folder, it asks for confirmation ("Are you sure you want to do that?"). Generally, Linux won't ask; it will just do it. This makes it easy to accidentally destroy a file or the entire file system. There is no way to undo a deletion on Linux unless a file is backed up on another media.
Command feedback	If a command succeeds on Linux, it redisplays the command prompt without a status message.

Table 10.7 Differences in the Linux operating environment (continued)

Difference	Description
Command switches	Linux uses a dash (-) to indicate command switches or a double dash (--) for multiple character options where DOS uses a slash (/) or dash (-).
Configuration files	On Windows, configuration is done in the registry or in files such as autoexec.bat. On Linux, configuration files are created as hidden files starting with a dot (.). Many are placed in the /etc directory and your home directory. Linux also uses environment variables such as LD_LIBRARY_PATH (search path for libraries). Other important environment variables: HOME Your home directory (/home/sam) TERM Terminal type (xterm, vt100, console) SHELL Path to your shell (/bin/bash) USER Your login name (sfuller) PATH List to search for programs They are specified in the shell or in rc files such as the .bashrc.
DLLs	On Linux, you use shared object files (.so). In Windows, these are dynamic link libraries (DLLs).
Drive letters	Linux doesn't have drive letters. An example Linux pathname is /lib/security. See DriveDelim in the runtime library.
Exceptions	Operating system exceptions are called signals on Linux.
Executable files	On Linux, executable files require no extension. On Windows, executable files have an exe extension.
File name extensions	Linux does not use file name extensions to identify file types or to associate files with applications.
File permissions	On Linux, files (and directories) are assigned read, write, and execute permissions for the file owner, group, and others. For example, `-rwxr-xr-x` means, from left to right: `-` is the file type (`-` = ordinary file, `d` = directory, `l` = link); `rwx` are the permissions for the file owner (read, write, execute); `r-x` are the permissions for the group of the file owner (read, execute); and `r-x` are the permissions for all other users (read, execute). The root user (superuser) can override these permissions. You need to make sure that your application runs under the correct user and has proper access to required files.
Make utility	Borland's make utility is not available on the Linux platform. Instead, you can use Linux's own GNU make utility.
Multitasking	Linux fully supports multitasking. You can run several programs (in Linux, called processes) at the same time. You can launch processes in the background (using & after the command) and continue working straight away. Linux also lets you have several sessions.
Pathnames	Linux uses a forward slash (/) wherever DOS uses a backslash (\). A PathDelim constant can be used to specify the appropriate character for the platform. See PathDelim in the runtime library.

Table 10.7 Differences in the Linux operating environment (continued)

Difference	Description
Search path	When executing programs, Windows always checks the current directory first, then looks at the PATH environment variable. Linux never looks in the current directory but searches only the directories listed in PATH. To run a program in the current directory, you usually have to type ./ before it. You can also modify your PATH to include ./ as the first path to search.
Search path separator	Windows uses the semicolon as a search path separator. Linux uses a colon. See PathDelim in the runtime library.
Symbolic links	On Linux, a symbolic link is a special file that points to another file on disk. Place symbolic links in the global bin directory that points to your application's main files and you don't have to modify the system search path. A symbolic link is created with the ln (link) command. Windows has shortcuts for the GUI desktop. To make a program available at the command line, Windows install programs typically modify the system search path.

Directory structure on Linux

Directories are different in Linux. Any file or device can be mounted anywhere on the file system.

Note Linux pathnames use forward slashes as opposed to Windows use of backslashes. The initial slash stands for the root directory.

Following are some commonly used directories in Linux.

Table 10.8 Common Linux directories

Directory	Contents
/	The root or top directory of the entire Linux file system
/root	The root file system; the Superuser's home directory
/bin	Commands, utilities
/sbin	System utilities
/dev	Devices shown as files
/lib	Libraries
/home/username	Files owned by the user where username is the user's login name.
/opt	Optional
/boot	Kernel that gets called when the system starts up
/etc	Configuration files
/usr	Applications, programs. Usually includes directories like /usr/spool, /usr/man, /usr/include, /usr/local
/mnt	Other media mounted on the system such as a CD or a floppy disk drive
/var	Logs, messages, spool files
/proc	Virtual file system and reporting system statistics
/tmp	Temporary files

Note Different distributions of Linux sometimes place files in different locations. A utility program may be placed in /bin in a Red Hat distribution but in /usr/local/bin in a Debian distribution.

Refer to www.pathname.com for additional details on the organization of the hierarchical file system and to read the *Filesystem Hierarchy Standard*.

Writing portable code

If you are writing cross-platform applications that are meant to run on both operating systems, you can write code that compiles under different conditions. Using conditional compilation, you can maintain your Windows coding, yet also make allowances for Linux operating system differences.

To create applications that are easily portable between Windows and Linux, remember to

- reduce or isolate calls to platform-specific (Win32 or Linux) APIs; use CLX methods instead.
- eliminate Windows messaging (PostMessage, SendMessage) constructs within an application.
- use *TMemIniFile* instead of *TRegIniFile*.
- observe and preserve case-sensitivity in file and directory names.
- port any external assembler TASM code. The GNU assembler, “as,” does not support the TASM syntax. (See “Including inline assembler code” on page 10-18.)

Try to write the code to use platform-independent runtime library routines and use constants found in System, SysUtils, and other runtime library units. For example, use the PathDelim constant to insulate your code from ‘/’ versus ‘\’ platform differences.

Another example involves the use of multibyte characters on both platforms. Windows code traditionally expects only 2 bytes per multibyte character. In Linux, multibyte character encoding can have many more bytes per char (up to 6 bytes for UTF-8). Both platforms can be accommodated using the StrNextChar function in SysUtils. Existing Windows code such as the following

```
while p^ <> #0 do
begin
   if p^ in LeadBytes then
      inc(p);
   inc(p);
end;
```

can be replaced with platform-independent code like this:

```
while p^ <> #0 do
begin
  if p^ in LeadBytes then
    p := StrNextChar(p)
  else
    inc(p);
end;
```

This example is platform portable and supports multibyte characters longer than 2 bytes, but still avoids the performance cost of a procedure call for non-multibyte locales.

If using runtime library functions is not a workable solution, try to isolate the platform-specific code in your routine into one chunk or into a subroutine. Try to limit the number of **$IFDEF** blocks to maintain source code readability and portability. The conditional symbol WIN32 is not defined on Linux. The conditional symbol LINUX is defined, indicating the source code is being compiled for the Linux platform.

Using conditional directives

Using **$IFDEF** compiler directives is a reasonable way to conditionalize your code for the Windows and Linux platforms. However, because **$IFDEF**s make source code harder to understand and maintain, you need to understand when it is reasonable to use **$IFDEF**s. When considering the use of **$IFDEF**s, the top questions should be "Why does this code require an **$IFDEF**?" and "Can this be written without an **$IFDEF**?"

Follow these guidelines for using **$IFDEF**s within cross-platform applications:

- Try not to use **$IFDEF**s unless absolutely necessary. **$IFDEF**s in a source file are only evaluated when source code is compiled. Unlike C/C++, Kylix does not require unit sources (header files) to compile a project. Full rebuilds of all source code is an uncommon event for most Kylix projects.
- Do not use **$IFDEF**s in package files (.dpk). Limit their use to source files. Component writers need to create two design-time packages when doing cross-platform development, not one package using **$IFDEF**s.
- In general, use **$IFDEF** MSWINDOWS to test for any Windows platform including WIN32. Reserve the use of **$IFDEF** WIN32 for distinguishing between specific Windows platforms, such as 32-bit versus 64-bit Windows. Don't limit your code to WIN32 unless you know for sure that it will not work in WIN64.
- Avoid negative tests like **$IFNDEF** unless absolutely required. **$IFNDEF** LINUX is *not* equivalent to **$IFDEF** MSWINDOWS.
- Avoid **$IFNDEF/$ELSE** combinations. Use a positive test instead (**$IFDEF**) for better readability.
- Avoid **$ELSE** clauses on platform-sensitive **$IFDEF**s. Use separate **$IFDEF** blocks for LINUX- and MSWINDOWS-specific code instead of **$IFDEF** LINUX/**$ELSE** or **$IFDEF** MSWINDOWS/**$ELSE**.

  For example, old code may contain

```
{$IFDEF WIN32}
  (32-bit Windows code)
{$ELSE}
  (16-bit Windows code)   //!! By mistake, Linux could fall into this code.
{$ENDIF}
```

For any non-portable code in **$IFDEF**s, it is better for the source code to fail to compile than to have the platform fall into an **$ELSE** clause and fail mysteriously at runtime. Compile failures are easier to find than runtime failures.

- Use the **$IF** syntax for complicated tests. Replace nested **$IFDEF**s with a boolean expression in an **$IF** directive. You should terminate the **$IF** directive using **$IFEND**, not **$ENDIF**. This allows you to place **$IF** expressions within **$IFDEF**s to hide the new **$IF** syntax from previous compilers.

All of the conditional directives are documented in the online Help. Also see, the topic "Conditional Compilation" in Help for more information.

Terminating conditional directives

Use the **$IFEND** directive to terminate **$IF** and **$ELSEIF** conditional directives. This allows **$IF/$IFEND** blocks to be hidden from older compilers inside of using **$IFDEF/$ENDIF**. Older compilers won't recognize the **$IFEND** directive. **$IF** can only be terminated with **$IFEND**. You can only terminate old-style directives (**$IFDEF, $IFNDEF, $IFOPT**) with **$ENDIF**.

Note When nesting an **$IF** inside of **$IFDEF/$ENDIF**, do not use **$ELSE** with the **$IF**. Older compilers will see the **$ELSE** and think it is part of the **$IFDEF**, producing a compile error down the line. You can use {**$ELSEIF** True} as a substitute for {**$ELSE**} in this situation, since the **$ELSEIF** won't be taken if the **$IF** is taken first, and the older compilers won't know **$ELSEIF**. Hiding **$IF** for backwards compatibility is primarily an issue for third party vendors and application developers who want their code to run on several different versions.

$ELSEIF is a combination of **$ELSE** and **$IF**. The **$ELSEIF** directive allows you to write multi-part conditional blocks where only one of the conditional blocks will be taken. For example:

```
{$IFDEF doit}
   do_doit
{$ELSEIF  RTLVersion >= 14}
    goforit
{$ELSEIF  somestring = 'yes'}
    beep
{$ELSE}
    last chance
{$IFEND}
```

Of these four cases, only one is taken. If none of the first three conditions is true, the **$ELSE** clause is taken. **$ELSEIF** must be terminated by **$IFEND**. **$ELSEIF** cannot appear after **$ELSE**. Conditions are evaluated top to bottom like a normal **$IF...$ELSE** sequence. In the example, if doit is not defined, RTLVersion is 15, and somestring = 'yes', only the "goforit" block will be taken not the "beep" block, even though the conditions for both are true.

If you forget to use an **$ENDIF** to end one of your **$IFDEF**s, the compiler reports the following error message at the end of the source file:

```
Missing ENDIF
```

If you have more than a few **$IF/$IFDEF** directives in your source file, it can be difficult to determine which one is causing the problem. Kylix reports the following error message on the source line of the last **$IF/$IFDEF** compiler directive with no matching **$ENDIF/$IFEND**:

```
Unterminated conditional directive
```

You can start looking for the problem at that location.

Emitting messages

The **$MESSAGE** compiler directive allows source code to emit hints, warnings, and errors just as the compiler does.

```
{$MESSAGE  HINT|WARN|ERROR|FATAL 'text string' }
```

The message type is optional. If no message type is indicated, the default is HINT. The text string is required and must be enclosed in single quotes.

Examples:

`{$MESSAGE 'Boo!'}` emits a hint.

`{$Message Hint 'Feed the cats'}` emits a hint.

`{$Message Warn 'Looks like rain.'}` emits a warning.

`{$Message Error 'Not implemented'}` emits an error, continues compiling.

`{$Message Fatal 'Bang.  Yer dead.'}` emits an error, terminates the compiler.

Including inline assembler code

If you include inline assembler code in your Windows applications, you may not be able to use the same code on Linux because of position-independent code (PIC) requirements on Linux. Linux shared object libraries (DLL equivalents) require that all code be relocatable in memory without modification. This primarily affects inline assembler routines that use global variables or other absolute addresses, or that call external functions.

For units that contain only Object Pascal code, the compiler automatically generates PIC when required. PIC units have a .dpu extension (instead of .dcu). It's a good idea to compile every Pascal unit source file into both PIC and non-PIC formats; use the -p compiler switch to generate PIC. Precompiled units are available in both forms.

You may want to code assembler routines differently depending on whether you'll be compiling to an executable or a shared library; use {**$IFDEF** PIC} to branch the two versions of your assembler code. Or you can consider rewriting the routine in Object Pascal to avoid the issue.

Following are the PIC rules for inline assembler code:

- PIC requires all memory references be made relative to the EBX register, which contains the current module's base address pointer (in Linux called the Global Offset Table or GOT). So, instead of

  ```
  MOV EAX,GlobalVar
  ```

use

```
MOV EAX,[EBX].GlobalVar
```

- PIC requires that you preserve the EBX register across calls into your assembly code (same as on Win32), and also that you restore the EBX register *before* making calls to external functions (different from Win32).
- While PIC code will work in base executables, it may slow the performance and generate more code. You don't have any choice in shared objects, but in executables you probably still want to get the highest level of performance that you can.

Messages and system events

Message loops and events work differently on Linux, but this primarily affects component writing. Most component and property editors port easily. *TObject.Dispatch* and message method syntax on classes work fine on Linux; under Linux, however, operating system notifications are handled using system events rather than messages.

To create an event handler, you can override one of the methods described in Table 10.9 to write your own custom message instead of responding to Windows messages. In the override, call the inherited method so any default processes still takes place.

Table 10.9 TWidgetControl protected methods for responding to system events

Method	Description
ChangeBounds	Used when a *TWidgetControl* is resized. Roughly analogous to WM_SIZE or WM_MOVE in Windows. Qt sets the "geometry" of a widget based on the client area, VCL uses the entire control size, which includes what Qt refers to as the frame.
ChangeScale	Called automatically when resizing controls. Used to change the scale of a form and all its controls for a different screen resolution or font size. Because ChangeScale modifies the control's Top, Left, Width, and Height properties, it changes the position of the control and its children as well as their size.
ColorChanged	Called when the color of the control has been changed.
CursorChanged	Called when the cursor changes shape. The mouse cursor assumes this shape when it's over this widget.
EnabledChanged	Called when an application changes the enabled state of a window or control.
FontChanged	Called when the collection of font resources changed. It sets the font for the widget and informs all children about the change. Roughly analogous to the WM_FONTCHANGE message.
PaletteChanged	Called when the system palette has been changed. Roughly analogous to the WM_PALETTECHANGED message.
ShowHintChanged	Called when Help hints are displayed or hidden on a control.
StyleChanged	Called when the window or control's GUI styles have changed. Roughly analogous to the WM_STYLECHANGED message.
TabStopChanged	Called when the tab order on the form has been changed.

Table 10.9 TWidgetControl protected methods for responding to system events (continued)

Method	Description
VisibleChanged	Called when a control is hidden or shown.
WidgetDestroyed	Called when a widget underlying a control is destroyed.

Qt is a C++ toolkit, so all of its widgets are C++ objects. CLX is written in Object Pascal, and Object Pascal does not interact directly with C++ objects. In addition, Qt uses multiple inheritance in a few places. So Kylix includes an interface layer that converts all of the Qt classes to a series of straight C functions. These are then wrapped in a shared object in Linux and a DLL in Windows.

Every *TWidgetControl* has *CreateWidget*, *InitWidget*, and *HookEvents* virtual methods that almost always have to be overridden. *CreateWidget* creates the Qt widget, and assigns the Handle to the FHandle private field variable. *InitWidget* gets called after the widget is constructed, and the Handle is valid.

Some property assignments on Linux have moved from the Create constructor to *InitWidget*. This will allow delayed construction of the Qt object until it's really needed. For example, say you have a property named *Color*. In SetColor, you can check with HandleAllocated to see if you have a Qt handle. If the Handle is allocated, you can make the proper call to Qt to set the color. If not, you can store the value in a private field variable, and, in *InitWidget*, you set the property.

Linux supports two types of events: Widget and System. *HookEvents* is a virtual method that hooks the CLX controls event methods to a special hook object that communicates with the Qt object. The hook object is really just a set of method pointers. System events on Linux go through *EventHandler*, which is basically a replacement for *WndProc*.

Programming differences on Linux

The Linux wchar_t widechar is 32 bits per character. The 16-bit Unicode standard that Object Pascal widechars support is a subset of the 32-bit UCS standard supported by Linux and the GNU libraries. Pascal widechar data must be widened to 32 bits per character before it can be passed to an OS function as wchar_t.

In Linux, widestrings are reference counted like long strings (in Windows, they're not).

Multibyte handling differs in Linux. In Windows, multibyte characters (MBCS) are represented as 1- and 2-byte char codes. In Linux, they are represented in 1 to 6 bytes.

AnsiStrings can carry multibyte character sequences, dependent upon the user's locale settings. The Linux encoding for multibyte characters such as Japanese, Chinese, Hebrew, and Arabic may not be compatible with the Windows encoding for the same locale. Unicode is portable, whereas multibyte is not.

In Linux, you cannot use variables on absolute addresses. The syntax `var X: Integer absolute $1234;` is not supported in PIC and will be disallowed in all versions of Kylix that include CLX.

Cross-platform database applications

On Windows, Delphi provides several choices for how to access database information. These include using ADO, the Borland Database Engine (BDE), and InterBase Express. These three choices are not available on Kylix, however. Instead, you can use *dbExpress*, a new, cross-platform data access technology, which is also available on Windows, starting with Delphi version 6.

Before you port a database application to *dbExpress* so that it will run on Linux, you should understand the differences between using *dbExpress* and the data access mechanism you were using. These differences occur at different levels.

- At the lowest level, there is a layer that communicates between your application and the database server. This could be ADO, the BDE, or the InterBase client software. This layer is replaced by *dbExpress*, which is a set of lightweight drivers for dynamic SQL processing.
- The low-level data access is wrapped in a set of components that you add to data modules or forms. These components include database connection components, which represent the connection to a database server, and datasets, which represent the data fetched from the server. Although there are some very important differences, due to the unidirectional nature of *dbExpress* cursors, the differences are less pronounced at this level, because datasets all share a common ancestor, as do database connection components.
- At the user-interface level, there are the fewest differences. CLX data-aware controls are designed to be as similar as possible to the corresponding Windows controls. The major differences at the user interface level arise from changes needed to accommodate the use of cached updates.

For information on porting existing database applications to *dbExpress*, see “Porting database applications to Linux” on page 10-23. For information on designing new *dbExpress* applications, see Chapter 14, “Designing database applications”.

dbExpress differences

On Linux, *dbExpress* manages the communication with database servers. *dbExpress* consists of a set of lightweight drivers that implement a set of common interfaces. Each driver is a shared object (.so file) that must be linked to your application. Because *dbExpress* is designed to be cross-platform, it will also be available on Windows as a set of dynamic-link libraries (.dlls).

As with any data-access layer, *dbExpress* requires the client-side software provided by the database vendor. In addition, it uses a database-specific driver, plus two configuration files, dbxconnections and dbxdrivers. This is markedly less than you need for, say, the BDE, which requires the main Borland Database Engine library (Idapi32.dll) plus a database-specific driver and a number of other supporting libraries.

Here are some other differences between *dbExpress* and the other data-access layers from which you need to port your application:

- *dbExpress* allows for a simpler and faster path to remote databases. As a result, you can expect a noticeable performance increase for simple, straight-through data access.
- *dbExpress* can process queries and stored procedures, but does not support the concept of opening tables.
- *dbExpress* returns only unidirectional cursors.
- *dbExpress* has no built-in update support other than the ability to execute an INSERT, DELETE, or UPDATE query.
- *dbExpress* does no metadata caching, and the design time metadata access interface is implemented using the core data-access interface.
- *dbExpress* executes only queries requested by the user, thereby optimizing database access by not introducing any extra queries.
- *dbExpress* manages a record buffer or a block of record buffers internally. This differs from the BDE, where clients are required to allocate the memory used to buffer records.
- *dbExpress* does not support local tables that are not SQL-based (such as Paradox, dBase, or FoxPro).
- *dbExpress* drivers exist for InterBase, Oracle, DB2, and MySQL. If you are using a different database server, you must either port your data to one of these databases, write a *dbExpress* driver for the database server you are using, or obtain a third-party *dbExpress* driver for your database server.

Component-level differences

When you write a *dbExpress* application, it requires a different set of data-access components than those used in your existing database applications. The *dbExpress* components share the same base classes as other data-access components (*TDataSet* and *TCustomConnection*), which means that many of the properties, methods, and events are the same as the components used in your existing applications.

Table 10.10 lists some of the important database components used in InterBase Express, BDE, and ADO in the Windows environment and shows the comparable *dbExpress* components for use on Linux and in cross-platform applications.

Table 10.10 Comparable data-access components

InterBase Express components	BDE components	ADO components	dbExpress components
TIBDatabase	*TDatabase*	*TADOConnection*	***TSQLConnection***
TIBTable	*TTable*	*TADOTable*	***TSQLTable***
TIBQuery	*TQuery*	*TADOQuery*	***TSQLQuery***
TIBStoredProc	*TStoredProc*	*TADOStoredProc*	***TSQLStoredProc***
TIBDataSet		*TADODataSet*	***TSQLDataSet***

The *dbExpress* datasets (*TSQLTable*, *TSQLQuery*, *TSQLStoredProc*, and *TSQLDataSet*) are more limited than their counterparts, however, because they do not support editing and only allow forward navigation. For details on the differences between the *dbExpress* datasets and the other datasets that are available on Windows, see Chapter 18, "Using unidirectional datasets.".

Because of the lack of support for editing and navigation, most *dbExpress* applications do not work directly with the *dbExpress* datasets. Rather, they connect the *dbExpress* dataset to a client dataset, which buffers records in memory and provides support for editing and navigation. For more information about this architecture, see "Database architecture" on page 14-4.

Note For very simple applications, you can use *TSQLClientDataSet* instead of a *dbExpress* dataset connected to a client dataset. This has the benefit of simplicity, because there is a 1:1 correspondence between the dataset in the application you are porting and the dataset in the ported application, but is less flexible that explicitly connecting a *dbExpress* dataset to a client dataset. For most applications, it is recommended that you use a *dbExpress* dataset connected to a *TClientDataSet* component.

User interface-level differences

CLX data-aware controls are designed to be as similar as possible to the corresponding Windows controls. As a result, porting the user-interface portion of your database applications introduces few additional considerations beyond those involved in porting any Windows application to CLX.

The major differences at the user interface level arise from differences in the way *dbExpress* datasets or client datasets supply data.

If you are using only *dbExpress* datasets, then you must adjust your user interface to accommodate the fact that the datasets do not support editing and only support forward navigation. Thus, for example, you may need to remove controls that allow users to move to a previous record. Because *dbExpress* datasets do not buffer data, you can't display data in a data-aware grid: only one record can be displayed at a time.

If you have connected the *dbExpress* dataset to a client dataset, then the user interface elements associated with editing and navigation should still work. You need only reconnect them to the client dataset. The main consideration in this case is handling how updates are written to the database. By default, most datasets on Windows write updates to the database server automatically when they are posted (for example, when the user moves to a new record). Client datasets, on the other hand, always cache updates in memory. For information on how to accommodate this difference, see "Updating data in dbExpress applications" on page 10-25.

Porting database applications to Linux

Porting your database application to *dbExpress* allows you to create a cross-platform application that runs both on Windows and Linux. The porting process involves making changes to your application because the technology is different. How difficult it is to port depends on the type of application it is, how complex it is, and what it needs to accomplish. An application that heavily uses Windows-specific technologies such as ADO will be more difficult to port than one that uses Delphi database technology.

Follow these general steps to port your Windows database application to Linux:

1. Consider where database data is stored. *dbExpress* provides drivers for Oracle, Interbase, DB2, and MySQL. The data needs to reside on one of these SQL servers.

   If you have the Delphi 5 Enterprise version, you can use the Data Pump utility to move local database data from platforms such as Paradox, dBase, and FoxPro onto one of the supported platforms. (See the datapump.hlp file in Program Files\Common Files\Borland\Shared\BDE for information on using the utility.)

2. If you have not isolated your user interface forms from data modules containing the datasets and connection components, you may want to consider doing so before you start the port. That way, you isolate the portions of your application that require a completely new set of components into data modules. Forms that represent the user interface can then be ported like any other application. For details, see “Porting your application” on page 10-2.

   The remaining steps assume that your datasets and connection components are isolated in their own data modules.

3. Create a new data module to hold the CLX versions of your datasets and connection components.

4. For each dataset in the original application, add a *dbExpress* dataset, *TDataSetProvider* component, and *TClientDataSet* component. Use the correspondences in Table 10.10 to decide which *dbExpress* dataset to use. Give these components meaningful names.

   - Set the *ProviderName* property of the *TClientDataSet* component to the name of the *TDataSetProvider* component.
   - Set the *DataSet* property of the *TDataSetProvider* component to the *dbExpress* dataset.
   - Change the *DataSet* property of any data source components that referred to the original dataset so that it now refers to the client dataset.

5. Set properties on the new dataset to match the original dataset:

   - If the original dataset was a *TTable*, *TADOTable*, or *TIBTable* component, set the new *TSQLTable*’s *TableName* property to the original dataset’s *TableName*. Also copy any properties used to set up master/detail relationships or specify indexes. Properties specifying ranges and filters should be set on the client dataset rather than the new *TSQLTable* component.
   - If the original dataset was a *TQuery*, *TADOQuery*, or *TIBQuery* component, set the new *TSQLQuery* component’s *SQL* property to the original dataset’s *SQL* property. Set the *Params* property of the new *TSQLQuery* to match the value of the original dataset’s *Params* or *Parameters* property. If you have set the *DataSource* property to establish a master/detail relationship, copy this as well.

- If the original dataset was a *TStoredProc*, *TADOStoredProc*, or *TIBStoredProc* component, set the new *TSQLStoredProc* component's *StoredProcName* to the *StoredProcName* or *ProcedureName* property of the original dataset. Set the *Params* property of the new *TSQLStoredProc* to match the value of the original dataset's *Params* or *Parameters* property.

6. For any database connection components in the original application (*TDatabase*, *TIBDatabase*, or *TADOConnection*), add a *TSQLConnection* component to the new data module. You must also add a *TSQLConnection* component for every database server to which you connected without a connection component (for example, by using the *ConnectionString* property on an ADO dataset or by setting the *DatabaseName* property of a BDE dataset to a BDE alias).
7. For each *dbExpress* dataset placed in step 4, set its *SQLConnection* property to the *TSQLConnection* component that corresponds to the appropriate database connection.
8. On each *TSQLConnection* component, specify the information needed to establish a database connection. To do so, double-click the *TSQLConnection* component to display the Connection Editor and set parameter values to indicate the appropriate settings. If you had to transfer data to a new database server in step 1, then specify settings appropriate to the new server. If you are using the same server as before, you can look up some of this information on the original connection component:

   - If the original application used *TDatabase*, you must transfer the information that appears in the *Params* and *TransIsolation* properties.
   - If the original application used *TADOConnection*, you must transfer the information that appears in the *ConnectionString* and *IsolationLevel* properties.
   - If the original application used *TIBDatabase*, you must transfer the information that appears in the *DatabaseName* and *Params* properties.
   - If there was no original connection component, you must transfer the information associated with the BDE alias or that appeared in the dataset's *ConnectionString* property.

   You may want to save this set of parameters under a new connection name. For more details on this process, see "Describing the server connection" on page 19-2.

Updating data in dbExpress applications

dbExpress applications use client datasets to support editing. When you post edits to a client dataset, the changes are written to the client dataset's in-memory snapshot of the data, but are not automatically written to the database server. If your original application used a client dataset for caching updates, then you do not need to change anything to support editing on Linux. However, if you relied on the default behavior of most datasets on Windows, which is to write edits to the database server when you post records, you must make changes to accommodate the use of a client dataset.

There are two ways to convert an application that did not previously cache updates:

- You can mimic the behavior of the dataset on Windows by writing code to apply each updated record to the database server as soon as it is posted. To do this, supply the client dataset with an *AfterPost* event handler that applies update to the database server:

```
procedure TForm1.ClientDataSet1AfterPost(DataSet: TDataSet);
begin
  with DataSet as TClientDataSet do
    ApplyUpdates(1);
end;
```

- You can adjust your user interface to deal with cached updates. This approach has certain advantages, such as reducing the amount of network traffic and minimizing transaction times. However, if you switch to using cached updates, you must decide when to apply those updates back to the database server, and probably make user interface changes to let users initiate the application of updates or inform provide them with feedback about whether their edits have been written to the database. Further, because update errors are not detected when the user posts a record, you will need to change the way you report such errors to the user, so that they can see which update caused a problem as well as what type of problem occurred.

If your original application used the support provided by the BDE or ADO for caching updates, you will need to make some adjustments in your code to switch to using a client dataset. The following table lists the properties, events, and methods that support cached updates on BDE and ADO datasets, and the corresponding properties, methods and events on *TClientDataSet*:

Table 10.11 Properties, methods, and events for cached updates

On BDE datasets (or TDatabase)	On ADO datasets	On TClientDataSet	Purpose
CachedUpdates	*LockType*	Not needed, client datasets always cache updates.	Determines whether cached updates are in effect.
Not supported.	*CursorType*	Not supported.	Specifies how isolated the dataset is from changes on the server.
UpdatesPending	Not supported.	*ChangeCount*	Indicates whether the local cache contains updated records that need to be applied to the database.
UpdateRecordTypes	*FilterGroup*	*StatusFilter*	Indicates the kind of updated records to make visible when applying cached updates.
UpdateStatus	*RecordStatus*	*UpdateStatus*	Indicates if a record is unchanged, modified, inserted, or deleted.
OnUpdateError	Not supported.	*OnReconcileError*	An event for handling update errors on a record-by-record basis.
ApplyUpdates (on dataset or database)	UpdateBatch	*ApplyUpdates*	Applies records in the local cache to the database.

Table 10.11 Properties, methods, and events for cached updates (continued)

On BDE datasets (or TDatabase)	On ADO datasets	On TClientDataSet	Purpose
CancelUpdates	CancelUpdates or CancelBatch	*CancelUpdates*	Removes pending updates from the local cache without applying them.
CommitUpdates	Handled automatically	*Reconcile*	Clears the update cache following successful application of updates.
FetchAll	Not supported	*GetNextPacket* (and *PacketRecords*)	Copies database records to the local cache for editing and updating.
RevertRecord	CancelBatch	*RevertRecord*	Undoes updates to the current record if updates are not yet applied.

Cross-platform Internet applications

An Internet application is a client/server application that uses standard Internet protocols for connecting the client to the server. Because your applications use standard Internet protocols for client/server communications, you can make your applications cross-platform. For example, a server-side program for an Internet application communicates with the client through the Web server software for the machine. The server application is typically written for Linux or Windows, but can also be cross-platform. The clients can be on either platform.

Kylix allows you to create Web server applications as CGI or Apache applications for deployment on Linux. On Windows, you can create other types of Web servers such as Microsoft Server DLLs (ISAPI), Netscape Server DLLs (NSAPI), and Windows CGI applications. Only straight CGI applications and some applications that use WebBroker will run on both Windows and Linux.

Porting Internet applications to Linux

If you have existing Internet applications that you want to move to Linux, you should consider whether you want to port your Web server application or if you want to create a new application on Linux. See Chapter 22, "Creating Internet server applications" for information on writing Web servers. If your application uses WebBroker and writes to the WebBroker interface and does not use native API calls, it will not be as difficult to port it to Linux.

If your application writes to ISAPI, NSAPI, Windows CGI, or other Web APIs, it will be more difficult to port. You will need to search through your source files and translate these API calls into Apache (see httpd.pas in the Internet directory for function prototypes for Apache APIs) or CGI calls. You also need to make all other suggested changes described in "Porting Windows applications to Linux" on page 10-1.

Chapter

11

Working with packages and components

A *package* is a special shared object file used by Kylix applications, the IDE, or both. *Runtime packages* provide functionality when a user runs an application. *Design-time packages* are used to install components in the IDE and to create special property editors for custom components. A single package can function at both design time and runtime, and design-time packages frequently work by calling runtime packages. Packages are stored in shared object files (typically prefixed with bpl) such as bplpackage.so.

Like other runtime libraries, packages contain code that can be shared among applications. For example, the most frequently used Kylix components reside in a package called bplclx. Each time you create an application, you can specify that it uses bplclx. When you compile an application created this way, the application's executable image contains only the code and data unique to it; the common code is in bplclx.so.6. A computer with several package-enabled applications installed on it needs only a single copy of bplclx.so.6, which is shared by all the applications and the IDE itself.

Kylix ships with several precompiled runtime packages that encapsulate CLX components. Kylix also uses design-time packages to manipulate components in the IDE.

You can build applications with or without packages. However, if you want to add custom components to the IDE, you must install them as design-time packages.

You can create your own runtime packages to share among applications. If you write Kylix components, you can compile them into design-time packages before installing them.

Why use packages?

Design-time packages simplify the tasks of distributing and installing custom components. Runtime packages, which are optional, offer several advantages over conventional programming. By compiling reused code into a runtime library, you can share it among applications. For example, all of your applications—including Kylix itself—can access standard components through packages. Since the applications don't have separate copies of the component library bound into their executables, the executables are much smaller—saving both system resources and hard disk storage. Moreover, packages allow faster compilation because only code unique to the application is compiled with each build.

Packages and standard shared object files

Create a package when you want to make a custom component that's available through the IDE. Create a standard shared object file when you want to build a library that can be called from any application, regardless of the development tool used to build the application.

The following table lists the file types associated with packages

Table 11.1 Compiled package files

File extension	Contents
dpk	The source file listing the units contained in the package.
dcp	A binary image containing a package header and the concatenation of all dpu files in the package, including all symbol information required by the compiler. A single dcp file is created for each package. The base name for the dcp is the base name of the dpk source file. You must have a .dcp file to build an application with packages.
dpu	A binary image for a unit file contained in a package. One dpu is created, when necessary, for each unit file.
so	The runtime package. This file is a shared object file with special Kylix-specific features. The name for the package is bpl*package*.so where *package* is the base name of the dpk file.

Note Packages share their global data with other modules in an application.

For more information about shared object files and packages, see the *Object Pascal Language Guide*.

Runtime packages

Runtime packages are deployed with Kylix applications. They provide functionality when a user runs the application.

To run an application that uses packages, a computer must have both the application's executable file and all the packages that the application uses. The package files must be on the system path for an application to use them. When you deploy an application, you must make sure that users have correct versions of any required packages.

Using packages in an application

To use packages in an application,

1 Load or create a project in the IDE.

2 Choose Project | Options.

3 Choose the Packages tab.

4 Select the “Build with Runtime Packages” check box, and enter one or more package names in the edit box underneath. (Runtime packages associated with installed design-time packages are already listed in the edit box.)

5 To add a package to an existing list, click the Add button and enter the name of the new package in the Add Runtime Package dialog.

6 To browse from a list of available packages, click the Add button, then click the Browse button next to the Package Name edit box in the Add Runtime Package dialog.

   If you edit the Search Path edit box in the Add Runtime Package dialog, you will be changing Kylix’s global Library Path.

You do not need to include file extensions with package names. If you type directly into the Runtime Packages edit box, be sure to separate multiple names with semicolons.

Packages listed in the Runtime Packages edit box are automatically linked to your application when you compile. Duplicate package names are ignored, and if the edit box is empty the application is compiled without packages.

Runtime packages are selected for the current project only. To make the current choices into automatic defaults for new projects, select the “Defaults” check box at the bottom of the dialog.

Note When you create an application with packages, you still need to include the names of the original Kylix units in the **uses** clause of your source files. For example, the source file for your main form might begin like this:

```
unit MainForm;

interface

uses
SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
```

Each of the units referenced in this example is contained in the bplclx package. Nonetheless, you must keep these references in the **uses** clause, even if you use bplclx in your application, or you will get compiler errors. In generated source files, Kylix adds these units to the **uses** clause automatically.

Dynamically loading packages

To load a package at runtime, call the *LoadPackage* function. For example, the following code could be executed when a file is chosen in a file-selection dialog.

```
with OpenDialog1 do
  if Execute then
    with PackageList.Items do
      AddObject(FileName, Pointer(LoadPackage(FileName)));
```

To unload a package dynamically, call *UnloadPackage*. Be careful to destroy any instances of classes defined in the package and to unregister classes that were registered by it.

Deciding which runtime packages to use

Kylix ships with several precompiled runtime packages, including bplclx, which supply basic language and component support.

The bplclx package contains the most commonly used components, system functions, and user interface elements. It does not include database components, which are available in separate packages.

For a list of the other runtime packages shipped with Kylix, see "runtime packages, precompiled" in your online Help index.

To create a client/server database application that uses packages, you need at least two runtime packages: bplclx and bpldataclx. To use these packages, choose Project | Options, select the Packages tab, and list the packages you want to use in the Runtime Packages edit box.

Custom packages

A custom package is either a package you code and compile yourself, or a precompiled package from a third-party vendor. To use a custom runtime package with an application, choose Project | Options and add the name of the package to the Runtime Packages edit box on the Packages page. For example, suppose you have a statistical package called bplstats.so. To use it in an application, include it in the Runtime Packages edit box.

If you create your own packages, you can add them to the list as needed.

Design-time packages

Design-time packages are used to install components on the IDE's Component palette and to create special property editors for custom components.

Kylixships with the following design-time component packages preinstalled in the IDE.

Table 11.2 Design-time packages

Package	Component palette pages
dclstd	Standard, Additional, Common Controls, Dialogs
dcldbdesign	Data Controls, dbExpress, Data Access
dclnet	Internet
dclindy	Indy Clients, Indy Servers, Indy Misc

The design-time packages work by calling runtime packages, which they reference in their Requires clauses. (See "The Requires clause" on page 11-8.) For example, dclstd references bplvcl. The dclstd package contains additional functionality that makes many of the standard components available on the Component palette.

In addition to preinstalled packages, you can install your own component packages, or component packages from third-party developers, in the IDE. The dclusr design-time package is provided as a default container for new components.

Installing component packages

All components are installed in the IDE as packages. If you've written your own components, create and compile a package that contains them. (See "Creating and editing packages" on page 11-6.) Your component source code must follow the model described in Part IV, "Creating custom components".

To install or uninstall your own components, or components from a third-party vendor, follow these steps:

1. If you are installing a new package, copy or move the package files to a local directory. If the package is shipped with additional files, be sure to copy all of them. (For information about these files, see "Package files created by a successful compilation" on page 11-11.)

   The directory where you store the .dcp file—and the .dpu files, if they are included with the distribution—must be in the Kylix Library Path.
2. Choose Component | Install Packages from the IDE menu, or choose Project | Options and click the Packages tab.
3. A list of available packages appears under "Design packages".
   - To install a package in the IDE, select the check box next to it.
   - To uninstall a package, deselect its check box.
   - To see a list of components included in an installed package, select the package and click Components.
   - To add a package to the list, click Add and browse in the Open Package dialog box for the directory where the package file resides (see step 1). Select the package file and click Open.
   - To remove a package from the list, select the package and click Remove.

4 Click OK.

The components in the package are installed on the Component palette pages specified in the components' *RegisterComponents* procedure, with the names they were assigned in the same procedure.

New projects are created with all available packages installed, unless you change the default settings. To make the current installation choices into the automatic default for new projects, check the Default check box at the bottom of the dialog box.

To remove components from the Component palette without uninstalling a package, select Component | Configure Palette, or select Tools | Environment Options and click the Palette tab. The Palette options tab lists each installed component along with the name of the Component palette page where it appears. Selecting any component and clicking Hide removes the component from the palette.

Creating and editing packages

Creating a package involves specifying

- A *name* for the package.
- A list of other packages to be *required* by, or linked to, the new package.
- A list of unit files to be *contained* by, or bound into, the package when it is compiled. The package is essentially a wrapper for these source-code units, which contain the functionality of the compiled package. The Contains clause is where you put the source-code units for custom components that you want to compile into a package.

Package source files, which end with the .dpk extension, are generated by the Package editor.

Creating a package

To create a package, follow the procedure below. Refer to "Understanding the structure of a package" on page 11-8 for more information about the steps outlined here.

Note Do not use IFDEFs in a package file (.dpk) such as when doing cross platform development. You can use them in the source code, however.

1 Choose File | New, select the Package icon, and click OK.

2 The generated package is displayed in the Package editor.

3 The Package editor shows a *Requires* node and a *Contains* node for the new package.

4 To add a unit to the **contains** clause, click the Add to package speed button. In the Add unit page, type a .pas file name in the Unit file name edit box, or click Browse to browse for the file, and then click OK. The unit you've selected appears under the Contains node in the Package editor. You can add additional units by repeating this step.

5 To add a package to the **requires** clause, click the Add to package speed button. In the Requires page, type a .dcp file name in the Package name edit box, or click Browse to browse for the file, and then click OK. The package you've selected appears under the Requires node in the Package editor. You can add additional packages by repeating this step.

6 Click the Options speed button, and decide what kind of package you want to build.

   - To create a design-time only package (a package that cannot be used at runtime), select the Designtime only radio button. (Or add the {$DESIGNONLY} compiler directive to the dpk file.)
   - To create a runtime-only package (a package that cannot be installed), select the Runtime only radio button. (Or add the {$RUNONLY} compiler directive to the dpk file.)
   - To create a package that is available at both design time and runtime, select the Designtime and runtime radio button.

7 In the Package editor, click the Compile package speed button to compile your package.

Editing an existing package

You can open an existing package for editing in several ways:

- Choose File | Open (or File | Reopen) and select a dpk file.
- Choose Component | Install Packages, select a package from the Design Packages list, and click the Edit button.
- When the Package editor is open, select one of the packages in the Requires node, right-click, and choose Open.

To edit a package's description or set usage options, click the Options speed button in the Package editor and select the Description tab.

The Project Options dialog has a Default check box in the lower left corner. If you click OK when this box is checked, the options you've chosen are saved as default settings for new projects. To restore the original defaults, delete or rename the defjproj.dof file.

Editing package source files manually

Package source files, like project files, are generated by Kylix from information you supply. Like project files, they can also be edited manually. A package source file should be saved with the .dpk (Kylix package) extension to avoid confusion with other files containing Object Pascal source code.

To open a package source file in the Code editor,

1 Open the package in the Package editor.

2 Right-click in the Package editor and select View Source.

- The **package** heading specifies the name for the package.
- The **requires** clause lists other, external packages used by the current package. If a package does not contain any units that use units in another package, then it doesn't need a **requires** clause.
- The **contains** clause identifies the unit files to be compiled and bound into the package. All units used by contained units which do not exist in required packages will also be bound into the package, although they won't be listed in the contains clause (the compiler will give a warning).

For example, the following code declares the bplclxdb package.

```
package bplclxdb;
   requires bplclx;
   contains Db, Dbcgrids, Dbctrls, Dbgrids, Dbinpreq, Dblogdlg, Dbpwdlg, Dbtables,
mycomponent in 'usr/components/mycomponent.pas';
end.
```

Understanding the structure of a package

Packages include the following parts:

- Package name
- Requires clause
- Contains clause

Naming packages

Package names must be unique within a project. If you name a package **stats**, the Package editor generates a source file for it called stats.dpk; the compiler generates an executable and a binary image called bplstats.so and stats.dcp, respectively. Use stats to refer to the package in the **requires** clause of another package, or when using the package in an application.

The Requires clause

The **requires** clause specifies other, external packages that are used by the current package. An external package included in the **requires** clause is automatically linked at compile time into any application that uses both the current package and one of the units contained in the external package.

If the unit files contained in your package make references to other packaged units, the other packages should appear in your package's **requires** clause or you should add them. If the other packages are omitted from the **requires** clause, the compiler will import them into your package 'implicitly contained units.'

Note Most packages that you create will require bplclx. Any package that depends on CLX units (including SysUtils) must list bplclx, or another package that requires bplclx, in its **requires** clause.

Avoiding circular package references

Packages cannot contain circular references in their **requires** clause. This means that

- A package cannot reference itself in its own **requires** clause.
- A chain of references must terminate without rereferencing any package in the chain. If package A requires package B, then package B cannot require package A; if package A requires package B and package B requires package C, then package C cannot require package A.

Handling duplicate package references

Duplicate references in a package's **requires** clause—or in the Runtime Packages edit box—are ignored by the compiler. For programming clarity and readability, however, you should catch and remove duplicate package references.

The Contains clause

The **contains** clause identifies the unit files to be bound into the package. If you are writing your own package, put your source code in pas files and include them in the **contains** clause.

Avoiding redundant source code uses

A package cannot appear in the **contains** clause of another package.

All units included directly in a package's **contains** clause, or included indirectly in any of those units, are bound into the package at compile time.

A unit cannot be contained (directly or indirectly) in more than one package used by the same application, *including the Kylix IDE*. This means that if you create a package that contains one of the units in bplclx, you won't be able to install your package in the IDE. To use an already-packaged unit file in another package, put the first package in the second package's **requires** clause.

Compiling packages

You can compile a package from the IDE or from the command line. To recompile a package by itself from the IDE,

1. Choose File | Open.
2. Select Kylix Package (*.dpk) from the Files of Type drop-down list.
3. Select a .dpk file in the dialog.
4. When the Package editor opens, click the Compile speed button.

You can insert compiler directives into your package source code. For more information, see "Package-specific compiler directives" below.

If you compile from the command line, several package-specific switches are available. For more information, see "Using the command-line compiler and linker" on page 11-11.

Package-specific compiler directives

The following table lists package-specific compiler directives that you can insert into your source code.

Table 11.3 Package-specific compiler directives

Directive	Purpose
{$IMPLICITBUILD OFF}	Prevents a package from being implicitly recompiled later. Use in .dpk files when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.
{$G-} or {IMPORTEDDATA OFF}	Disables creation of imported data references. This directive increases memory-access efficiency, but prevents the unit where it occurs from referencing variables in other packages.
{$WEAKPACKAGEUNIT ON}	Packages unit "weakly." See "Weak packaging" on page 11-10 below.
{$DENYPACKAGEUNIT ON}	Prevents unit from being placed in a package.
{$DESIGNONLY ON}	Compiles the package for installation in the IDE. (Put in .dpk file.)
{$RUNONLY ON}	Compiles the package as runtime only. (Put in .dpk file.)

Note Including **{$DENYPACKAGEUNIT ON}** in your source code prevents the unit file from being packaged. Including **{$G-}** or {**IMPORTEDDATA OFF}** may prevent a package from being used in the same application with other packages. Packages compiled with the **{$DESIGNONLY ON}** directive should not ordinarily be used in applications, since they contain extra code required by the IDE. Other compiler directives may be included, if appropriate, in package source code. See Compiler directives in the online help for information on compiler directives not discussed here.

Weak packaging

The **$WEAKPACKAGEUNIT** directive affects the way a .dpu file is stored in a package's files. (For information about files generated by the compiler, see "Package files created by a successful compilation" on page 11-11.) If **{$WEAKPACKAGEUNIT ON}** appears in a unit file, the compiler omits the unit from packages when possible, and creates a non-packaged local copy of the unit when it is required by another application or package. A unit compiled with this directive is said to be "weakly packaged."

For example, suppose you've created a package called **pack** that contains only one unit, unit1. Suppose unit1 does not use any further units, but it makes calls to rare.so. If you put **{$WEAKPACKAGEUNIT ON}** in unit1.pas when you compile your package, unit1 will not be included in bplpack.so; you will not have to distribute copies of rare.so with pack. However, unit1 will still be included in pack.dcp. If unit1 is referenced by another package or application that uses pack, it will be copied from pack.dcp and compiled directly into the project.

Now suppose you add a second unit, unit2, to pack. Suppose that unit2 uses unit1. This time, even if you compile pack with **{$WEAKPACKAGEUNIT ON}** in unit1.pas, the compiler will include unit1 in bplpack.so. But other packages or applications that reference unit1 will use the (non-packaged) copy taken from pack.dcp.

Note Unit files containing the **{$WEAKPACKAGEUNIT ON}** directive must not have global variables, initialization sections, or finalization sections.

The **$WEAKPACKAGEUNIT** directive is an advanced feature intended for developers who distribute their packages to other Kylix programmers. It can help you to avoid distribution of infrequently used shared object files, and to eliminate conflicts among packages that may depend on the same external library.

Using the command-line compiler and linker

When you compile from the command line, you can use the package-specific switches listed in the following table.

Table 11.4 Package-specific command-line compiler switches

Switch	Purpose
-$G-	Disables creation of imported data references. Using this switch increases memory-access efficiency, but prevents packages compiled with it from referencing variables in other packages.
-LE*path*	Specifies the directory where the package (bcp*package*.so) file will be placed.
-LN*path*	Specifies the directory where the package (*package*.dcp) file will be placed.
-LU*package*	Use packages.
-Z	Prevents a package from being implicitly recompiled later. Use when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.

Note Using the **-$G-** switch may prevent a package from being used in the same application with other packages. Other command-line options may be used, if appropriate, when compiling packages. See “The Command-line compiler” in the online help for information on command-line options not discussed here.

Package files created by a successful compilation

To create a package, you compile a source file that has a .dpk extension. The base name of the .dpk file becomes the base name of the files generated by the compiler. For example, if you compile a package source file called traypak.dpk, the compiler creates a package called bpltraypak.so.

The following table lists the files produced by the successful compilation of a package.

Table 11.5 Compiled package files

Filename or extension	Contents
dcp	A binary image containing a package header and the concatenation of all dpu files in the package. A single dcp file is created for each package. The base name for the dcp is the base name of the dpk source file.
dpu	A binary image for a unit file contained in a package. One dpu is created, when necessary, for each unit file.
bpl*package*.so	The runtime package. This file is a shared object file with special Kylix-specific features. In the name, *package* is the base name of the dpk source file.

When compiled, the package and library files are generated by default in the directories specified on the Library page of the Tools | Environment Options dialog. You can override the default settings by clicking the Options speed button in the Package editor to display the Project Options dialog; make any changes on the Directories/Conditionals page.

Deploying packages

The following sections provide recommendations about package deployment.

Deploying applications that use packages

When distributing an application that uses runtime packages, make sure that your users have the application's executable file as well as all the library files that the application calls. If the library files are in a different directory from the executable file, they must be accessible through the user's path.

Distributing packages to other developers

If you distribute runtime or design-time packages to other Kylix developers, be sure to supply both the .dcp and .so files. You will probably want to include .dpu files as well.

Chapter

12

Creating international applications

This chapter discusses guidelines for writing applications that you plan to distribute to an international market. By planning ahead, you can reduce the amount of time and code necessary to make your application function in its foreign market as well as in its domestic market.

Internationalization and localization

To create an application that you can distribute to foreign markets, there are two major steps that need to be performed:

- Internationalization
- Localization

Internationalization

Internationalization is the process of enabling your program to work in multiple locales. A *locale* is the user's environment, which includes the cultural conventions of the target country as well as the language. Linux supports many locales, each of which is described by a language and country pair (for example, en_US for English/United States). The user locale identifier can be obtained from the operating system by looking for the LANG environment variable.

Most Kylix applications do not need to determine the exact locale in which the application is running but do need to handle strings in a locale-independent manner.

Localization

Localization is the process of translating an application so that it functions in a specific locale. In addition to translating the user interface, localization may include functionality customization. For example, a financial application may be modified to be aware of the different tax laws in different countries.

Internationalizing applications

It is not difficult to create internationalized applications. You need to complete the following steps:

1 You must enable your code to handle strings from international character sets.

2 You need to design your user interface so that it can accommodate the changes that result from localization.

3 You need to isolate all resources that need to be localized.

Enabling application code

You must make sure that the code in your application can handle the strings it will encounter in the various target locales.

Character sets

The Linux operating system uses UTF-8 to encode file names and paths, and other strings passed to the operating system kernel. The Linux operating system generally has no knowledge of locale specifics.

End user applications generally require more specific locale support than simply which character set to display on the screen. Locales and languages have different rules on how string data should be sorted, what characters are considered equivalent in comparisons, and how numbers should be formatted for display. UTF-8 provides no such support; you have to use locales and local character sets.

Multiple byte character sets

The ideographic character sets used in Asia cannot use the simple 1:1 mapping between characters in the language and the one byte (8-bit) *char* type. These languages have too many characters to be represented using the 1-byte *char*. Instead, characters in a multibyte string can contain one or more bytes per character. AnsiStrings can contain a mix of single-byte and multibyte characters.

The first byte of every multibyte character code is taken from a reserved range that depends on the specific character set. The second and subsequent bytes can sometimes be the same as the character code for a separate 1-byte character, or it can fall in the range reserved for the first byte of multibyte characters. Thus, the only way to tell whether a particular byte in a string represents a single character or is part of a multibyte character is to read the string, starting at the beginning, parsing it into 2- or more byte characters when a lead byte from the reserved range is encountered.

When writing code for Asian locales, you must be sure to handle all string manipulation using functions that are enabled to parse strings into multibyte characters. Kylix also provides a number of runtime library functions that allow you to do this many of which are listed here:

AdjustLineBreaks	AnsiStrLIComp	CharToByteLen
AnsiCompareFileName	AnsiStrLower	ExtractFileDir
AnsiCompareText	AnsiStrPos	ExtractFileExt
AnsiCompareStr	AnsiStrRScan	ExtractFileName
AnsiDequotedStr	AnsiStrScan	ExtractFilePath
AnsiExtractQuotedStr	AnsiStrUpper	ExtractRelativePath
AnsiLastChar	AnsiToUtf8	FileSearch
AnsiLowerCase	AnsiUpperCase	IsDelimiter
AnsiLowerCaseFileName	AnsiUpperCaseFileName	IsPathDelimiter
AnsiPos	ByteToCharIndex	LastDelimiter
AnsiQuotedStr	ByteToCharLen	StrByteType
AnsiStrComp	ByteType	StringReplace
AnsiStrIComp	ChangeFileExt	Utf8ToAnsi
AnsiStrLastChar	AnsiStrLIComp	WrapText
AnsiStrLComp	CharToByteIndex	

Note If your application will be run on Linux systems using pre-2.2 versions of glibc, avoid passing large strings (greater than 50K) to multibyte-enabled routines. (There are no string limitations if using glibc version 2.2 or greater.)

Remember that the length of the strings in bytes does not necessarily correspond to the length of the string in characters. Be careful not to truncate strings by cutting a multibyte character in half. Do not pass characters as a parameter to a function or procedure, since the size of a character can't be known up front. Instead, always pass a pointer to a character or a string.

Wide characters

Display string properties such as captions, descriptions, text properties, combo boxes, and so on are all wide strings.

Note The Linux wchar_t WideChar is 32 bits per character. The 16-bit Unicode standard that Object Pascal WideChars support is a subset of the 32-bit UCS standard supported by Linux and the GNU libraries. Pascal WideChar data must be widened to 32 bits per character before it can be passed to an OS function as wchar_t.

The Linux kernel uses 4-byte widechars. The kernel expects strings (file names and so forth) to be encoded in UTF-8. Kylix WideChar and WideString are 2 bytes per character Unicode, which is a subset of the UCS-4 specification. To translate Unicode 2-byte characters to UCS 4-byte characters, you must add two bytes of zeros in front.

Designing the user interface

When creating an application for several foreign markets, it is important to design your user interface so that it can accommodate the changes that occur during translation.

Text

All text that appears in the user interface must be translated. English text is almost always shorter than its translations. Design the elements of your user interface that display text so that there is room for the text strings to grow. Create dialogs, menus, status bars, and other user interface elements that display text so that they can easily display longer strings. Avoid abbreviations—they do not exist in languages that use ideographic characters.

Short strings tend to grow in translation more than long phrases. Table 12.1 provides a rough estimate of how much expansion you should plan for given the length of your English strings:

Table 12.1 Estimating string lengths

Length of English string (in characters)	Expected increase
1-5	100%
6-12	80%
13-20	60%
21-30	40%
31-50	20%
over 50	10%

Graphic images

Ideally, you will want to use images that do not require translation. Most obviously, this means that graphic images should not include text, which will always require translation. If you must include text in your images, it is a good idea to use a label object with a transparent background over an image rather than including the text as part of the image.

There are other considerations when creating graphic images. Try to avoid images that are specific to a particular culture. For example, mailboxes in different countries look very different from each other. Religious symbols are not appropriate if your application is intended for countries that have different dominant religions. Even color can have different symbolic connotations in different cultures.

Formats and sort order

The date, time, number, and currency formats used in your application should be localized for the target locale. If you specify any of your own format strings, be sure to declare them as resourced constants so that they can be localized.

The order in which strings are sorted also varies from country to country. Many European languages include diacritical marks that are sorted differently, depending on the locale. In addition, in some countries, 2-character combinations are treated as a single character in the sort order. For example, in Spanish, the combination *ch* is sorted like a single unique letter between *c* and *d*. Sometimes a single character is sorted as if it were two separate characters, such as the German *eszett*.

Keyboard mappings

Be careful with key-combinations shortcut assignments. Not all the characters available on the US keyboard are easily reproduced on all international keyboards. Where possible, use number keys and function keys for shortcuts, as these are available on virtually all keyboards.

Isolating resources

The most obvious task in localizing an application is translating the strings that appear in the user interface. To create an application that can be translated without altering code everywhere, the strings in the user interface should be isolated into a single module. Kylix automatically creates a form file that contains the resources for your menus, dialogs, and bitmaps.

In addition to these obvious user interface elements, you will need to isolate any strings, such as error messages, that you present to the user. String resources are not included in the form file. You can isolate them by declaring constants for them using the **resourcestring** keyword. For more information about resource string constants, see the *Object Pascal Language Guide*. It is best to include all resource strings in a single, separate unit.

Kylix resource strings are encoded in UTF-8 (1 byte per character, usually) in the executable file.

Creating resource modules

Isolating resources simplifies the translation process. The next level of resource separation is the creation of a resource module. A resource module is a library that contains all the resources and only the resource strings for a program (no code). Resource modules allow you to create a program that supports many translations simply by swapping the resource module.

To create a resource module for your program, create a file that contains the **resourcestring** strings for the project and generate a project for a resource only shared object file that contains the relevant forms. The resources are compiled into a separate section of the executable file.

You should create a resource module for each translation you want to support. Each resource module should have a file name specific to the target locale, for example en_US for US English.

Using resource modules

The executable, shared object files, and packages that make up your application contain all the necessary resources. However, to replace those resources with localized versions, you need only ship your application with localized resource modules that have the same name as your executable, shared object file, or package files.

When your application starts up, it checks the locale of the local system by looking at the LC_ALL environment variable. If it finds any resource modules with the same name as the executable file, shared object file, or package files it is using, it checks the extension of those shared object files. If the extension of the resource module matches the language and country of the system locale, your application will use the resources in that resource module instead of the resources in the executable, shared object file, or package. If no resource module matches both the language and the country, your application will try to locate a resource module that matches the language only. If no resource module file name extension matches the language, your application uses the resources compiled into the executable, shared object file, or package.

You can ship a single application that adapts itself automatically to the locale of the system it is running on, simply by providing the appropriate resource modules.

Localizing applications

Once your application is internationalized, you can create localized versions for the different foreign markets in which you want to distribute it.

Ideally, your resources have been isolated into a resource module that contains form files. You can open your forms in the IDE, translate the relevant properties, then compile the application. You can extract any resources needed using the *resbind* command line utility located in ~/kylix/bin.

Resbind extracts the Borland resources from your application and creates a shared object file that contains the resources. You can then dynamically link the resources at runtime or let the application check the environment variable on the local system on which it is running.

Resbind reads resources from an ELF file and optionally one or more resource files. The ELF file may be a Linux i386 executable or shared object file. *Resbind* accepts resource files in Windows 32-bit and 16-bit formats, and in Borland's resource file format for Linux.

Resbind can print the resources, copy them to a new resource file, or copy them to a copy of the ELF file with the resources replaced by resources read from the resource files using the following syntax:

```
resbind [option]... ELF-FILE RESOURCE-FILE...
```

The following table lists the *resbind* options.

Table 12.2 Resbind options

Option	Description
-h	Display help and exit
-V	Output version information and exit
-f FMT	Control the resource file output format. FMT may be 'w32' (default) or 'borland'
-l FILE	Dynamic linker to use for -s, default /lib/ld-linux.so.2
-o FILE	Write the updated ELF-FILE to FILE
-p	Print combined resources
-r FILE	Write combined resources to resource file FILE
-s FILE	Write combined resources to resource shared object FILE
-S SONAME	Define soname for -s

Resbind can be used to copy resources from a application into a resource file for editing. Later, you can copy the edited resources back into the application.

For example, the following command copies resources from an ELF file called "program" to a Windows 32-bit resource file called "program.res:"

```
$ resbind -r program.res program
```

You can then combine the edited resources located in "program.res" with the ELF file called "program" and write the result to "program.new."

```
$ resbind -o program.new program program.res
```

Resources not included in "program.res" remain unchanged.

You can also write combined resources to a resource shared object:

```
$ resbind -s program-res.so program program.res
```

Chapter

13

Deploying applications

Once your Kylix application is up and running, you can deploy it. That is, you can make it available for others to run. A number of steps must be taken to deploy an application to another computer so that the application is completely functional. The steps required by a given application vary, depending on the type of application. The following sections describe those steps for deploying applications:

- Deploying general applications
- Deploying database applications
- Deploying Web applications
- Programming for varying host environments
- Software license requirements

Deploying general applications

In Linux, applications are commonly deployed using one of the following methods:

- Using a tool such as Red Hat Package Manager (RPM) from Red Hat, Inc.
- By creating tar files (tar.gz) or gnu zip files (gzip)
- Using a setup program developed for Linux such as the Setup Graphic Installer open sourced by Loki Entertainment Software

Each of these methods has its advantages and disadvantages and which one to use depends on the type of application and files required to run the application. Simple programs can be deployed using tar files. More complex programs should be deployed using either RPM or a setup program.

RPM is used to install, uninstall, upgrade, verify, and build software packages. RPM creates an archive of files and application information (including a name, version, and brief description). It is available on many versions of Linux and UNIX and is widely used for software distribution. When using RPM, the term *package* is used to mean a .rpm file or a preassembled unit containing software that is meant to be installed using RPM. RPM is a command executed from the shell:

```
rpm -i [options] [packages]
```

Other setup programs are available for deploying applications on Linux. For example, the Setup Graphic Installer is a graphic installation utility that developers can use to create an easy-to-use installation for all types of applications.

Deployment issues

Beyond the executable file, an application may require a number of supporting files, such as shared object files, initialization files, package files, and helper applications. The process of copying an application's files to a computer and making any needed settings can be automated by an installation program. Following are the main deployment concerns common to nearly all types of applications:

- Using installation programs
- Identifying application files

Kylix applications that access databases and those that run across the Web require additional installation steps beyond those that apply to general applications. For additional information on installing database applications, see "Deploying database applications" on page 13-3. For more information on installing Web applications, see "Deploying Web applications" on page 13-4.

Using installation programs

Simple Kylix applications that consist of only an executable file are easy to install on a target computer. Just copy the executable file onto the computer. However, more complex applications that comprise multiple files require more extensive installation procedures. These applications require dedicated installation programs.

Identifying application files

Besides the executable file, a number of other files may need to be distributed with an application.

The following types of files may need to be distributed with an application.

- Application files
- Shared object files
- Initialization files
- Package files
- Helper applications
- Help files
- Database files

Package files

If the application uses runtime packages, those package files need to be distributed with the application. By convention, package files are typically placed in a lib directory along with other shared objects. This serves as a common location so that multiple applications could access a single instance of the files. For packages you create, it is recommended that you install them in the same directory as the application. Only the .so files need to be distributed.

If you are distributing packages to other developers, you need to supply both the .so and the .dcp files.

Helper applications

Helper applications are separate programs without which your Kylix application would be partially or completely unable to function. Helper applications may be those supplied by Borland, or they might be third-party products.

If an application depends on a helper program, be sure to deploy it with your application, where possible. Distribution of helper programs may be governed by redistribution license agreements. Consult the documentation provided with the helper program for specific information.

Shared object file locations

You can install shared object files used only by a single application in the same directory as the application. Shared object files that will be used by a number of applications should be installed in a location accessible to all of those applications. A common convention for locating such community shared object files on Linux systems is to place them in the /bin directory.

Deploying database applications

Applications that access databases involve special installation considerations beyond copying the application's executable file onto the host computer. Database access is most often handled by a separate database engine, the files of which cannot be linked into the application's executable file. The data files, when not created beforehand, must be made available to the application. Multi-tier database applications require additional handling on installation, because the files that make up the application are typically located on multiple computers.

Kylix applications use *dbExpress* to connect to a database. *dbExpress* is a set of database drivers that provide quick access to SQL database servers. For each supported database, *dbExpress* provides a driver that adapts the server-specific software to a set of uniform *dbExpress* interfaces. When you deploy your application, you need only include a single shared object (the server-specific driver) with the application files you build. If you are supporting multiple databases, you'll need a shared object file for each database.

If you are using a client dataset (*TClientDataSet* or *TSQLClientDataSet*) or if you are using *TDataSetProvider*, you also need to distribute midas.so.

Connecting to a database

To open a database connection, you must identify both the driver to use and a set of connection parameters to be passed to that driver. The driver is identified by its *DriverName* property, which is the name of an installed *dbExpress* driver. Supported drivers include databases such as InterBase, MYSQL, Oracle, or DB2. Two files are connected to the driver name:

- A *dbExpress* driver, which is a shared object file with names such as libsqlib.so, libsqlmys.so, libsqlora.so, or libsqldb2.so
- On the client side, the shared object file provided by the database vendor.

The relationship between these two shared object files and the database name is stored in a file called dbxdrivers. *dbExpress* also lets you define database connection names and save them in a file called dbxconnections. This allows you to deploy your database applications onto systems that access different databases. You only need to deploy the dbxdrivers and the dbxconnections files if you are loading database information at runtime. Details about database connections are covered in “Controlling connections” on page 19-2.

Install the drivers and connections files either in the same directory as your executable or in a .borland subdirectory under the home directory. If you want to share the *dbExpress* drivers and configuration files among several applications that you are developing, put them in the .borland directory. (This is the same place that Kylix places the dbxdrivers and dbxconnections files.)

Updating configuration files

If using shared configuration files, your installation program can use the MergeIniFile utility provided with the product to merge the contents of an existing drivers or connections file with a new or updated one. The syntax is:

```
MergeIniFile( SourceIniFile, TargetIniFile, FOverwrite )
```

The utility returns a count of the number of sections added. FOverwrite is *False* by default. If set to *True*, it adds the section even if it exists already and overwrites any settings for the section found in the SourceIniFile. Sections not in the SourceIniFile remain unchanged, and settings not in the configuration file are not deleted.

Deploying Web applications

Some Kylix applications are designed to be run over the World Wide Web, such as those in the form of Apache shared object files and CGI applications. CGI applications can be deployed on any Web server that supports CGI interfaces. For deployment on Apache Web servers, see “Deployment on Apache” on page 13-5.

The steps for installing Web applications are the same as those for general applications, except the application’s files are deployed on the Web server. For information on installing general applications, see “Deploying general applications” on page 13-1.

Here are some special considerations for deploying Web applications:

- For database applications, the *dbExpress* driver is installed along with the application files on the Web server.
- Security for the directories must not be so high that access to application files or database files is not possible.
- The directory containing an application must have read and execute attributes.
- Apache configuration files must specify the location of CGI applications. That directory must have read and execute attributes.
- The application should not use hard-coded paths for accessing database or other files.

Deployment on Apache

WebBroker supports Apache version 1.3.9 and later. If using Apache, you need to be sure to set appropriate directives in the Apache configuration file, called httpd.conf. The SetEnv directive should be set to export the LD_LIBRARY_PATH variable and set it to point to the location of all shared objects required by the application at runtime. Therefore, httpd.conf must contain:

```
SetEnv LD_LIBRARY_PATH <directory>
```

where <directory> is the full path to the Kylix installation directory.

The physical directory must have the ExecCGI option set to allow execution of programs; httpd.conf should contain lines similar to the following:

```
<Directory "/<directory>/httpd/cgi-bin">
    AllowOverride None
    Options ExecCGI
    Order allow,deny
    Allow from all
</Directory>
```

You also need to set the HomeEnv variable in the httpd.conf file. For database applications, *dbExpress* requires a .borland directory to be located in the home directory.

Apache executes locally on the server within the account specified in the User directive in the httpd.conf file.

Programming for varying host environments

Due to the nature of the Linux environment, there are a number of factors that vary with user preference or configuration. The following factors can affect an application deployed to another computer:

- Screen resolutions and color depths
- Fonts
- Helper applications
- Shared object file locations

Screen resolutions and color depths

The size of the desktop and number of available colors on a computer is configurable and dependent on the hardware and Xserver that is installed. These attributes are also likely to differ on the deployment computer compared to those on the development computer.

An application's appearance (window, object, and font sizes) on computers configured for different screen resolutions can be handled in various ways:

- Design the application for the lowest resolution users will have (typically, 640x480). Take no special actions to dynamically resize objects to make them proportional to the host computer's screen display. Visually, objects will appear smaller the higher the resolution is set.
- Design using any screen resolution on the development computer and, at runtime, dynamically resize all forms and objects proportional to the difference between the screen resolutions for the development and deployment computers (a screen resolution difference ratio).
- Design using any screen resolution on the development computer and, at runtime, dynamically resize only the application's forms. Depending on the location of visual controls on the forms, this may require the forms be scrollable for the user to be able to access all controls on the forms.

Considerations when not dynamically resizing

If the forms and visual controls that make up an application are not dynamically resized at runtime, design the application's elements for the lowest resolution. Otherwise, the forms of an application run on a computer configured for a lower screen resolution than the development computer may overlap the boundaries of the screen.

For example, if the development computer is set up for a screen resolution of 1024x768 and a form is designed with a width of 700 pixels, not all of that form will be visible within the desktop on a computer configured for a 640x480 screen resolution.

Considerations when dynamically resizing forms and controls

If the forms and visual controls for an application are dynamically resized, accommodate all aspects of the resizing process to ensure optimal appearance of the application under all possible screen resolutions. Here are some factors to consider when dynamically resizing the visual elements of an application:

- The resizing of forms and visual controls is done at a ratio calculated by comparing the screen resolution of the development computer to that of the computer onto which the application installed. Use a constant to represent one dimension of the screen resolution on the development computer: either the height or the width, in pixels. Retrieve the same dimension for the user's computer at runtime using the *TScreen.Height* or the *TScreen.Width* property. Divide the value for the development computer by the value for the user's computer to derive the difference ratio between the two computers' screen resolutions.

- Resize the visual elements of the application (forms and controls) by reducing or increasing the size of the elements and their positions on forms. This resizing is proportional to the difference between the screen resolutions on the development and user computers. Resize and reposition visual controls on forms automatically by setting the *CustomForm.Scaled* property to *True* and calling the *TWidgetControl.ScaleBy* method. The *ScaleBy* method does not change the form's height and width, though. Do this manually by multiplying the current values for the *Height* and *Width* properties by the screen resolution difference ratio.
- The controls on a form can be resized manually, instead of automatically with the *TWidgetControl.ScaleBy* method, by referencing each visual control in a loop and setting its dimensions and position. The *Height* and *Width* property values for visual controls are multiplied by the screen resolution difference ratio. Reposition visual controls proportional to screen resolution differences by multiplying the *Top* and *Left* property values by the same ratio.
- If an application is designed on a computer configured for a higher screen resolution than that on the user's computer, font sizes will be reduced in the process of proportionally resizing visual controls. If the size of the font at design time is too small, the font as resized at runtime may be unreadable. For example, the default font size for a form is 8. If the development computer has a screen resolution of 1024x768 and the user's computer 640x480, visual control dimensions will be reduced by a factor of 0.625 (640 / 1024 = 0.625). The original font size of 8 is reduced to 5 (8 * 0.625 = 5). Text in the application appears jagged and unreadable as it is displayed in the reduced font size.
- Some visual controls, such as *TLabel* and *TEdit*, dynamically resize when the size of the font for the control changes. This can affect deployed applications when forms and controls are dynamically resized. The resizing of the control due to font size changes are in addition to size changes due to proportional resizing for screen resolutions. This effect is offset by setting the *AutoSize* property of these controls to *False*.
- Avoid making use of explicit pixel coordinates, such as when drawing directly to a canvas. Instead, modify the coordinates by a ratio proportionate to the screen resolution difference ratio between the development and user computers. For example, if the application draws a rectangle to a canvas ten pixels high by twenty wide, instead multiply the ten and twenty by the screen resolution difference ratio. This ensures that the rectangle visually appears the same size under different screen resolutions.

Accommodating varying color depths

To account for all deployment computers not being configured with the same color availability, the safest way is to use graphics with the least possible number of colors. This is especially true for control glyphs, which should typically use 16-color graphics. For displaying pictures, either provide multiple copies of the images in different resolutions and color depths or explain in the application the minimum resolution and color requirements for the application.

Fonts

Linux comes with a standard set of fonts. When designing an application to be deployed on other computers, realize that not all computers will have fonts outside the standard set.

Text components used in the application should all use fonts that are likely to be available on all deployment computers.

When use of a nonstandard font is absolutely necessary in an application, you need to distribute that font with the application. Either the installation program or the application itself must install the font on the deployment computer. Distribution of third-party fonts may be subject to limitations imposed by the font creator.

Software license requirements

The distribution of some files associated with Kylix applications is subject to limitations or cannot be redistributed at all. The following documents describe the legal stipulations regarding the distribution of these files:

- deploy.txt
- README
- No-nonsense license agreement
- GPL license agreement
- Third-party product documentation

Deploy.txt

Deploy.txt covers some of the legal aspects of distributing of various components and utilities, and other product areas that can be part of or associated with a Kylix application. Deploy.txt is a text file installed in the main Kylix directory. The topics covered include

- Executable files, shared object files, and package files
- Components and design-time packages
- Sample Images
- Client datasets (*TDataSetProvider*, *TClientDataSet*, or *TSQLDataSet*)

README

The README file contains last minute information about Kylix, possibly including information that could affect the redistribution rights for components, or utilities, or other product areas. README is a text file installed into the main Kylix directory.

No-nonsense license agreement

The Kylix no-nonsense license agreement, a printed document, covers other legal rights and obligations concerning Kylix.

GPL license agreement

The General Public License (GPL) agreement specifies information concerning open source licensing terms for using CLX to develop open sourced applications.

Third-party product documentation

Redistribution rights for third-party components, utilities, helper applications, database engines, and other products are governed by the vendor supplying the product. Consult the documentation for the product or the vendor for information regarding the redistribution of the product with Kylix applications prior to distribution.

Part

II

Developing database applications

The chapters in “Developing Database Applications” present concepts and skills necessary for creating Kylix database applications.

Note Database components are not available in all versions of Kylix.

Chapter

14

Designing database applications

Database applications let users interact with information that is stored in databases. Databases provide structure for the information, and allow it to be shared among different applications.

Kylix provides support for relational database applications. Relational databases organize information into tables, which contain rows (records) and columns (fields). These tables can be manipulated by simple operations known as the relational calculus.

You have several choices available when designing a database application. This chapter describes the various architectures available. Consider each type of architecture's advantages and disadvantages to choose the approach that best suits your needs. You may want to start with a simple approach, and then scale it up later to another, more powerful architecture. Kylix's database support makes this easy because the same components are used in most of the architectures.

Using databases

The components on the dbExpress page of the Component palette let your application connect to and read from databases. These components use *dbExpress* to access database information, which they make available to other components that repackage and buffer the information, or display it to end-users.

dbExpress is a set of drivers for different types of databases. While all types of databases contain tables which store information, different types support additional features such as

- Database security
- Transactions
- Referential integrity, stored procedures, and triggers

Types of databases

Relational database servers vary in the way they store information and in the way they allow multiple users to access that information simultaneously. Kylix provides support for two types of relational database server:

- **Remote database servers** typically reside on a separate machine. Sometimes, the data from a remote database server does not even reside on a single machine, but is distributed over several servers. Although remote database servers vary in the way they store information, they provide a common logical interface to clients. This common interface is Structured Query Language (SQL). Because you access them using SQL, they are sometimes called SQL servers. (Another name is Remote Database Management system, or RDBMS.) In addition to the common commands that make up SQL, most remote database servers support a unique "dialect" of SQL. Examples of SQL servers include InterBase, Oracle, DB2, and MySQL.
- **Local databases** reside on your local drive or on a local area network. They often have proprietary APIs for accessing the data. When they are shared by several users, they use file-based locking mechanisms. Because of this, they are sometimes called file-based databases. Kylix provides support for two types of local databases: Local InterBase, which is a local version of the InterBase server, and a proprietary file format for the data stored in a client dataset.

Applications that use local databases are called **single-tiered applications** because the application and the database share a single file system. Applications that use remote database servers are called **two-tiered applications** or **multi-tiered applications** because the application and the database operate on independent systems (or tiers).

Choosing the type of database to use depends on several factors. For example, your data may already be stored in an existing database. If you are creating the database tables your application uses, you may want to consider the following questions:

- How many users will be sharing these tables? Remote database servers are designed for access by several users at the same time. They provide support for multiple users through a mechanism called transactions. Some local databases (such as Local InterBase) also provide transaction support, but many only provide file-based locking mechanisms, and some (such as client dataset files) provide no multi-user support at all.
- How much data will the tables hold? Remote database servers can hold more data than local databases. Some remote database servers are designed for warehousing large quantities of data while others are optimized for other criteria (such as fast updates).
- What type of performance (speed) do you require from the database? Local databases are usually faster than remote database servers because they reside on the same system as the database application. Different remote database servers are optimized to support different types of operations, so you may want to consider performance when choosing a remote database server.
- What type of support will be available for database administration? Local databases require less support than remote database servers. Typically, they are

less expensive to operate because they do not require separately installed servers or expensive site licenses.

Database security

Databases often contain sensitive information. Most SQL servers provide security at multiple levels. Typically, they require a password and user name to use the database server at all. Once the user has logged in to the database, that username and password determine which tables can be used. For information on providing passwords to SQL servers, see “Controlling server login” on page 19-5.

When designing database applications, you must consider what type of authentication is required by your database server. Often, applications are designed to hide the explicit database login from users, who need only log in to the application itself. If you do not want to require your users to provide a database password, you must either use a database that does not require one or you must provide the password and username to the server programmatically. When providing the password programmatically, care must be taken that security can’t be breached by reading the password from the application.

If your application requires multiple passwords because you must log in to several protected systems or databases, you can have your users provide a single master password which is used to access a table of passwords required by the protected systems. The application then supplies passwords programmatically, without requiring the user to provide multiple passwords.

In multi-tiered applications, you may want to use a different security model altogether. You can use a protocol such as HTTPs to control access to middle tiers, and let the middle tiers handle all details of logging into database servers.

Transactions

A transaction is a group of actions that must all be carried out successfully on one or more tables in a database before they are committed (made permanent). If any of the actions in the group fails, then all actions are rolled back (undone).

Transactions ensure that

- All updates in a single transaction are either committed or aborted and rolled back to their previous state. This is referred to as **atomicity**.
- A transaction is a correct transformation of the system state, preserving the state invariants. This is referred to as **consistency**.
- Concurrent transactions do not see each other's partial or uncommitted results, which might create inconsistencies in the application state. This is referred to as **isolation**.
- Committed updates to records survive failures, including communication failures, process failures, and server system failures. This is referred to as **durability**.

Thus, transactions protect against hardware failures that occur in the middle of a database command or set of commands. Transactional logging allows you to recover the durable state after disk media failures. Transactions also form the basis of multi-user concurrency control on SQL servers. When each user interacts with the database only through transactions, one user's commands can't disrupt the unity of another user's transaction. Instead, the SQL server schedules incoming transactions, which either succeed as a whole or fail as a whole.

Transaction support is provided by most SQL servers. However, some servers, such as MySQL, provide no support for transactions. Similarly, if you are only using client datasets and storing the data in files, there is no transaction support.

For details on using transactions in your database applications, see "Managing transactions" on page 19-7.

Referential integrity, stored procedures, and triggers

All relational databases have certain features in common that allow applications to store and manipulate data. In addition, databases often provide other, database-specific, features that can prove useful for ensuring consistent relationships between the tables in a database. These include

- **Referential integrity.** Referential integrity provides a mechanism to prevent master/detail relationships between tables from being broken. When the user attempts to delete a field in a master table which would result in orphaned detail records, referential integrity rules prevent the deletion or automatically delete the orphaned detail records.
- **Stored procedures.** Stored procedures are sets of SQL statements that are named and stored on an SQL server. Stored procedures usually perform common database-related tasks on the server, and sometimes return sets of records (datasets).
- **Triggers.** Triggers are sets of SQL statements that are automatically executed in response to a particular command.

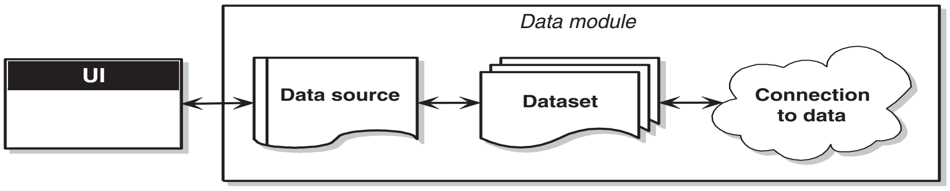
Database architecture

Database applications are built from user interface elements, components that represent database information (datasets), and components that connect these to each other and to the source of the database information. How you organize these pieces is the architecture of your database application.

General structure

While there are many distinct ways to organize the components in a database application, most follow the general scheme illustrated in Figure 14.1:

Figure 14.1 Generic Database Architecture



The user interface form

It is a good idea to isolate the user interface on a form that is completely separate from the rest of the application. This has several advantages. By isolating the user interface from the components that represent the database information itself, you introduce a greater flexibility into your design: Changes to the way you manage the database information do not require you to rewrite your user interface, and changes to the user interface do not require you to change the portion of your application that works with the database. In addition, this type of isolation lets you develop common forms that you can share between multiple applications, thereby providing a consistent user interface. By storing links to well-designed forms in the Object Repository, you and other developers can build on existing foundations rather than starting over from scratch for each new project. Sharing forms also makes it possible for you to develop corporate standards for application interfaces. For more information about creating the user interface for a database application, see Chapter 15, "Using data controls".

The data module

If you have isolated your user interface into its own form, you can use a data module to house the components that represent database information (datasets), and the components that connect these datasets to the other parts of your application. Like the user interface forms, you can share data modules in the Object Repository so that they can be reused or shared between applications.

The data source

The first item in the data module is a data source. The data source acts as a conduit between the user interface and a dataset that represents information from a database. Several data-aware controls on a form can share a single data source, in which case the display in each control is synchronized so that as the user scrolls through records, the corresponding value in the fields for the current record is displayed in each control.

The dataset

The heart of your database application is the dataset. This component represents a set of records from the underlying database. These records can be the data from a single database table, a subset of the fields or records in a table, or information from more than one table joined into a single view. By using datasets, your application logic is buffered from restructuring of the physical tables in your databases. When the underlying database changes, you might need to alter the way the dataset component specifies the data it contains, but the rest of your application can continue to work without alteration. For more information on the common properties and methods of datasets, see Chapter 16, “Understanding datasets”.

The data connection

Different types of datasets use different mechanisms for connecting to the underlying database information. These different mechanisms, in turn, make up the major differences in the architecture of the database applications you can build. Kylix provides support for two types of datasets:

- **Client datasets**, which buffer data in memory so that you can navigate through records more easily and perform operations on the data such as filtering records or maintaining aggregate values that summarize the data. Client datasets provide support for applying the updates in the in-memory cache back to the underlying database. Because client datasets cache the records in memory, they can only hold a limited number of records. There are two types of client datasets: generic client datasets and SQL client datasets. Generic client datasets access their data by working directly with a file stored locally on disk, connecting to another dataset in the same data module, or connecting to an application server on another (server) machine. SQL client datasets can use a file stored locally on disk or connect to a database server. For more information about client datasets, see Chapter 20, “Using client datasets”.
- **Unidirectional datasets**, which can read data that is described by an SQL query or that is returned by a stored procedure. Unidirectional datasets do not buffer data, so they are less flexible than client datasets. The only way to navigate through the records of a unidirectional dataset is to iterate through them in the order specified by the ORDER BY clause of the SQL query. Further, you can’t use a unidirectional dataset to update data. However, unidirectional datasets provide fast access to information from a database server, and can represent far larger sets of data than client datasets. Unidirectional datasets always fetch their data using an SQL connection component For more information about unidirectional data sets, see Chapter 18, “Using unidirectional datasets.”

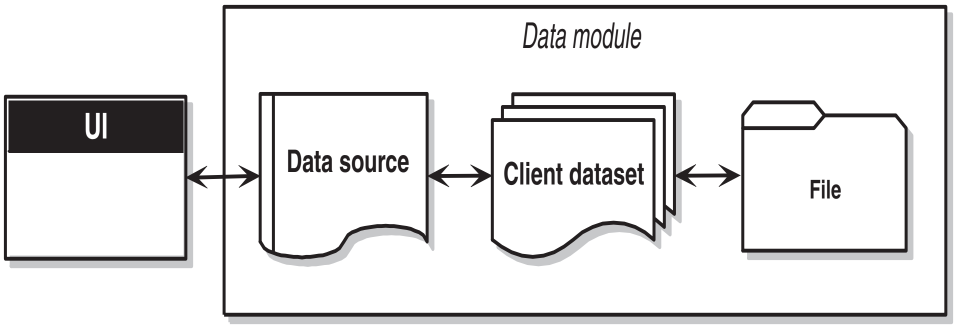
In addition, you can create your own custom datasets**.** These are descendants of *TDataSet* that represent a body of data that you create or access in code you write. Writing custom datasets allows you the flexibility of managing the data using any method you choose, while still letting you use the CLX data controls to build your user interface.

The following topics describe the most common ways to use client datasets, local datsets, and unidirectional datasets in database applications.

Using a client dataset with data stored on disk

The simplest form of database application you can write does not use a database server at all. Instead, it uses the ability of client datasets to save themselves to a file and to load the data from a file. The architecture for this type of application is illustrated in Figure 14.2:

Figure 14.2 Architecture of a file-based database application



This simple file-based architecture is a single-tiered application. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

The file-based approach has the benefit of simplicity. There is no database server to install, configure, or deploy (although the client dataset does require midas.so). There is no need for site licenses or database administration.

However, there is no support for multiple users. The dataset must be dedicated entirely to the application. Data is saved to files on disk, and loaded at a later time, but there is no built-in protection to prevent multiple users from overwriting each other's data files.

Because there is no separate database server, you are responsible for creating the underlying table yourself. Once this table is created, you can save it to a file. From then on, you do not need to recreate the table, only load it from the file you saved. When beginning a file-based database application, you may want to first create and save empty files for your datasets before writing the application itself. This way, you do not need to define the metadata for your client datasets in the final application. For more information about creating the tables for file-based applications, see "Creating a new dataset" on page 20-37.

In this file-based model, all edits to the data exist only in an in-memory change log. This log can be maintained separately from the data itself, although it is completely transparent to objects that use the client dataset. That is, controls that navigate the client dataset or display its data see a view of the data that includes the changes. If you do not want to back out of changes, however, you should merge the change log into the data of the client dataset by calling the *MergeChangeLog* method. For more information about the change log, see "Editing data" on page 20-14.

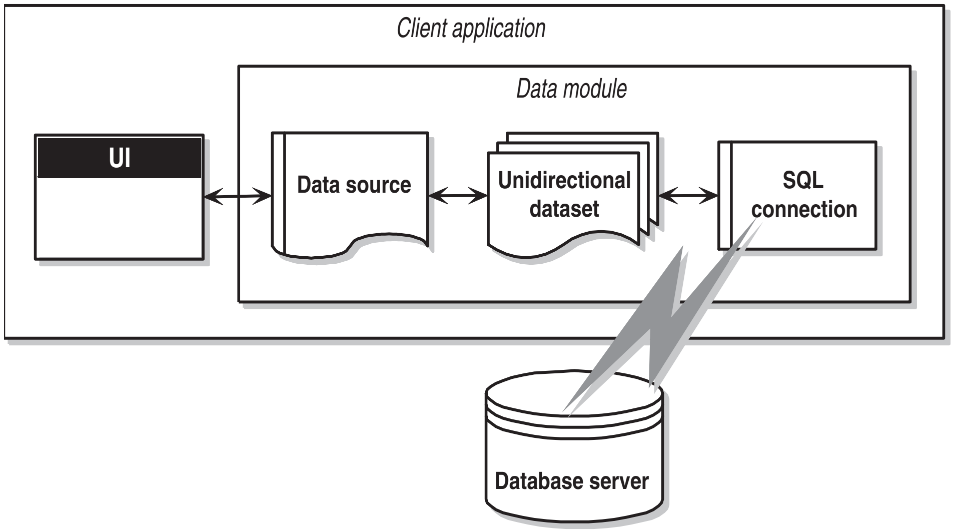
Even when you have merged changes into the data of the client dataset, this data still exists only in memory. While it persists if you close the client dataset and reopen it in your application, it will disappear when your application shuts down. To make the data permanent, it must be written to disk. Write changes to disk using the *SaveToFile* method. *SaveToFile* takes one parameter, the name of the file which is created (or overwritten) containing the table. When you want to read a table previously written using the *SaveToFile* method, use the *LoadFromFile* method. *LoadFromFile* also takes one parameter, the name of the file containing the table.

If you always load to and save from the same file, you can use the *FileName* property instead of the *SaveToFile* and *LoadFromFile* methods. When *FileName* is set to a valid file name, the data is automatically loaded from the file when the client dataset is opened and saved to the file when the client dataset is closed.

Using a unidirectional dataset directly

Kylix requires a unidirectional dataset to connect to a database server. Thus, the simplest way to use data from an SQL server is to connect this unidirectional dataset directly to the user interface. The architecture for this type of application is illustrated in Figure 14.3.

Figure 14.3 Architecture of a unidirectional database application



This architecture represents either a single-tiered or two-tiered application, depending on whether the database server is a local database such as Local InterBase or a remote database server. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

This model provides very fast access when working with a database server. There is no overhead for buffering data or managing metadata. Because no data is buffered in memory, you can use this model with arbitrarily large datasets.

However, when using this model, your application can only view one record at a time, and you can only progress through records in order. In addition, the data is read-only: there is no built-in way to post changes back to the database.

This model is best suited to applications that read the data and analyze it to present a set of summary statistics or print a report. It can, however, be used to present users with values through which to browse.

The dataset in this model is a unidirectional dataset (*TSQLDataSet*, *TSQLQuery*, *TSQLTable*, or *TSQLStoredProcedure*). Unidirectional datasets execute an SQL statement on the database server and, if the SQL statement returns data (that is, if it is a SELECT statement), obtains a unidirectional cursor for accessing the resulting data.

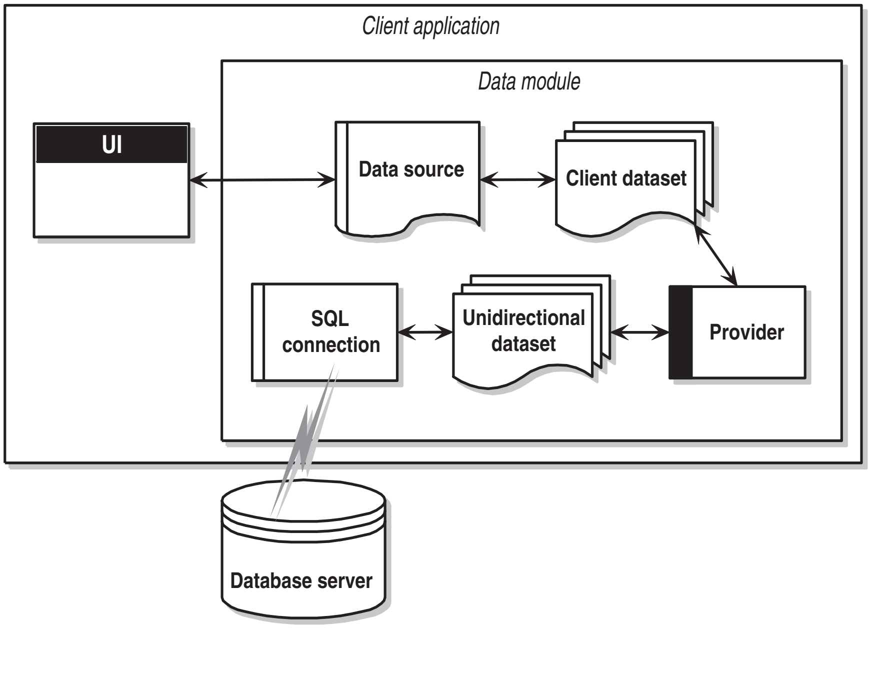
In order to connect a unidirectional dataset to a server, the data module must contain a SQL connection component (*TSQLConnection*). The unidirectional dataset is linked to this SQL connection component via its *SQLConnection* property. The SQL connection component represents the connection to the database server, and requires dbExpress. Double-click on the SQL connection component to identify the server in the Connection Editor. This editor lets you specify a connection name, which represents a named set of configuration parameters for a specific driver. Each named connection configuration is defined in a special file called connections.ini. You can use the Connection Editor to select a configuration, modify a named configuration, or define a new one.

Note You may need to purchase some of the drivers separately.

Using a client dataset to buffer records

To display multiple records, use data-aware controls to update data, move backwards through records, or move to records that meet a specific criterion, you must use a client dataset. On the other hand, you must use a unidirectional dataset to connect to a database server. You can combine these to create an application that has the flexibility of a client dataset but works with data from a database server. The architecture for this type of application is illustrated in Figure 14.4:

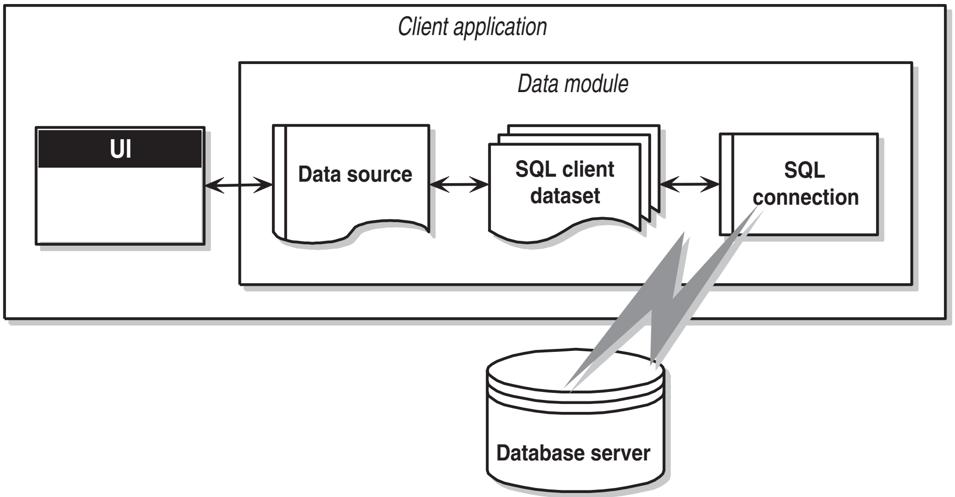
Figure 14.4 Architecture combining a client dataset and a unidirectional dataset



The connection between the client dataset and the unidirectional dataset is provided by a dataset provider. The provider packages database information into transportable data packets (which can be used by client datasets) and applies updates received in delta packets (which client datasets create) back to a database server. To link the client dataset to the provider, set its *ProviderName* property to the name of the provider component. The provider and the client dataset must be in the same data module. To link the provider to the unidirectional dataset, set its *DataSet* property to the unidirectional dataset.

To simplify this architecture, Kylix includes a special type of client dataset, called an SQL client dataset, that contains its own, internal provider and unidirectional dataset. By using an SQL client dataset, the preceding arrangement is simplified to look like Figure 14.5.

Figure 14.5 Architecture using a client dataset with an internal unidirectional dataset



When using an SQL client dataset, you need not explicitly add a dataset provider and unidirectional dataset to the data module: these components are internal to the SQL client dataset. This simplifies your application, but at the cost of some control:

- Because the unidirectional dataset is internal to the SQL client dataset component, you can't link two unidirectional datasets in a master/detail relationship to obtain nested detail sets. (You can, however, link two SQL client datasets into a master/detail relationship.)
- The SQL client dataset component does not surface most of the events that occur on its internal dataset provider. However, in most cases, these events are used in multi-tiered applications, and are not needed for two-tiered applications.

Both of these models represent either a single-tiered or two-tiered application, depending on whether the database server is a local database such as Local Interbase or a remote database server. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

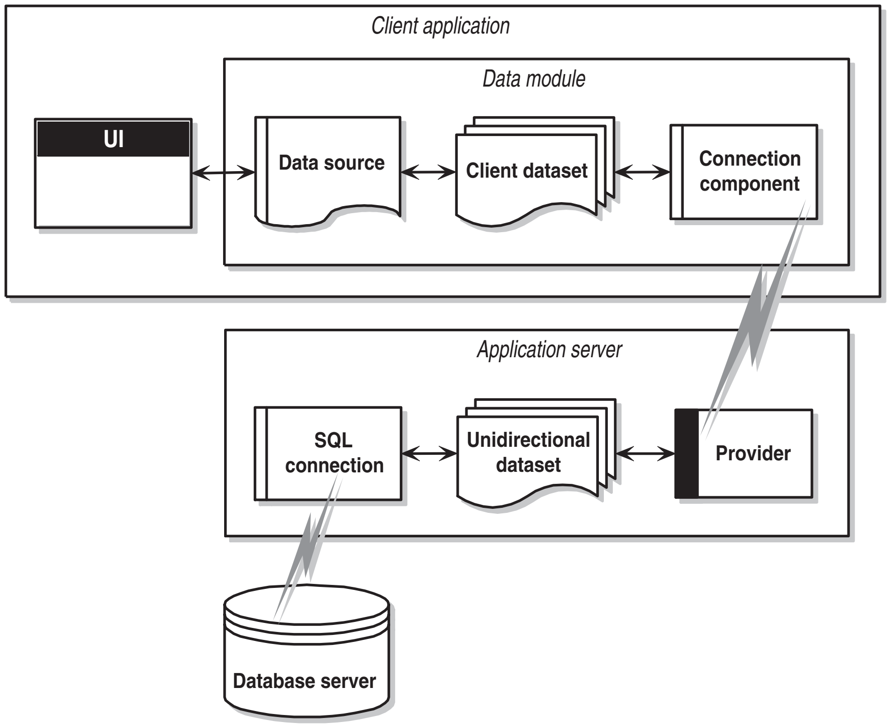
These models represent a hybrid of the two previous models. They use a (possibly internal) unidirectional dataset to access the database server and fetch records, while the client dataset buffers the records to provide more flexibility and applies updates back to the database server when the user edits the data.

Client datasets automatically handle all the details necessary for fetching, displaying, and navigating through database records. To apply user edits back to the database, you need only call the client dataset's *ApplyUpdates* method. (Note that, unlike when using a client dataset with data stored on disk, you must not call the *MergeChangeLog* method, or the client can't produce accurate delta packets for the provider.)

Using a multi-tiered architecture

When the database information includes complicated relationships between several tables, or the number of clients grows, you may want to use a multi-tiered model. Multi-tiered applications have middle tiers between the client application and the database server. This architecture is illustrated in Figure 14.6:

Figure 14.6 Multi-tiered database architecture



The preceding figure represents three-tiered application. The logic that manipulates database information is on a separate system, or tier. This middle tier centralizes the logic that governs your database interactions so there is centralized control over data relationships. This allows different client applications to use the same data while ensuring that the data logic is consistent. It also allows for smaller client applications because much of the processing is off-loaded onto the middle tier. These smaller client applications are easier to install, configure, and maintain. Multi-tiered applications can also improve performance by spreading the data-processing tasks over several systems.

The multi-tiered architecture is very similar to using a client dataset to buffer records from an external provider and unidirectional dataset. It differs mainly in that the unidirectional dataset that connects to the database server and the provider that acts as an intermediary between that unidirectional dataset and the client dataset have both moved to a separate application. That separate application is called the application server (or sometimes the "remote data broker").

Because the provider has moved to a separate application, the client dataset can no longer connect to the unidirectional dataset by simply setting its *ProviderName* property. In addition, it must use some type of connection component to locate and connect to the application server. This component is a descendant of *TCustomRemoteServer* such as *TSoapConnection*. Link the client dataset to its connection component by setting the *RemoteServer* property.

Note You may need to purchase the components to connect a client dataset with a remote application server separately.

The connection component establishes a connection to the application server and returns an interface that the client dataset uses to call the provider specified by its *ProviderName* property. Each time the client dataset calls the application server, it passes the value of *ProviderName*, and the application server forwards the call to the provider.

Combining approaches

The previous sections describe several architectures you can use when writing database applications. There is no reason, however, why you can't combine two or more of the available architectures in a single application. In fact, some combinations can be extremely powerful.

For example, you can combine the disk-based architecture described in "Using a client dataset with data stored on disk" on page 14-6 with another approach such as "Using a client dataset to buffer records" on page 14-9 or "Using a multi-tiered architecture" on page 14-11. The result is called the briefcase model (or sometimes the disconnected model, or mobile computing).

The briefcase model is useful in a situation such as the following: An onsite company database contains customer contact data that sales representatives can use and update in the field. While onsite, sales representatives download information from the database. Later, they work with it on their laptops as they fly across the country, and even update records at existing or new customer sites. When the sales reps return onsite, they upload their data changes to the company database for everyone to use.

When operating on site, the client dataset in a briefcase model application fetches its data from a provider. The dataset is therefore connected to the database server and can, through the provider, fetch server data and send updates back to the server. Before disconnecting from the provider, the client dataset saves its snapshot of the information to a file on disk. While offsite, the client dataset loads its data from the file, and saves any changes back to that file. Finally, back onsite, the client dataset reconnects to the provider so that it can apply its updates to the database server or refresh its snapshot of the data.

Chapter

15

Using data controls

The Data Controls page of the Component palette provides a set of data-aware controls that represent data from fields in a database record, and, if the dataset allows it, enable users to edit that data and post changes back to the database. By placing data controls onto the forms in your database application, you can build your database application's user interface (UI) so that information is visible and accessible to users.

The data-aware controls you add to your user interface depend on several factors, including the following:

- The type of data you are displaying. You can choose between controls that are designed to display and edit plain text, controls that work with formatted text, controls for graphics, multimedia elements, and so on. Controls that display different types of information are described in "Displaying a single record" on page 15-7.
- How you want to organize the information. You may choose to display information from a single record on the screen, or list the information from multiple records using a grid. "Choosing how to organize the data" on page 15-7 describes some of the possibilities.
- The type of dataset that supplies data to the controls. You want to use controls that reflect the limitations of the underlying dataset. For example, you would not use a grid with a unidirectional dataset because unidirectional datasets can only supply a single record at a time.
- How (or if) you want to let users navigate through the records of datasets and add or edit data. You may want to add your own controls or mechanisms to navigate and edit, or you may want to use a built-in control such as a data navigator. For more information about using a data navigator, see "Navigating and manipulating records" on page 15-25.

Regardless of the data-aware controls you choose to add to your interface, certain common features apply. These are described below.

Using common data control features

The following tasks are common to most data controls:

- Associating a data control with a dataset
- Editing and updating data
- Disabling and enabling data display
- Refreshing data display
- Enabling mouse, keyboard, and timer events

Data controls generally let you display and edit fields of data associated with the current record in a dataset. Table 15.1 summarizes the data controls that appear on the Data Controls page of the Component palette.

Table 15.1 Data controls

Data control	Description
TDBGrid	Displays information from a data source in a tabular format. Columns in the grid correspond to columns in the underlying table or query's dataset. Rows in the grid correspond to records.
TDBNavigator	Navigates through data records in a dataset. updating records, posting records, deleting records, canceling edits to records, and refreshing data display.
TDBText	Displays data from a field as a label.
TDBEdit	Displays data from a field in an edit box.
TDBMemo	Displays data from a memo or BLOB field in a scrollable, multi-line edit box.
TDBImage	Displays graphics from a data field in a graphics box.
TDBListBox	Displays a list of items from which to update a field in the current data record.
TDBComboBox	Displays a list of items from which to update a field, and also permits direct text entry like a standard data-aware edit box.
TDBCheckBox	Displays a check box that indicates the value of a Boolean field.
TDBRadioGroup	Displays a set of mutually exclusive options for a field.
TDBLookupListBox	Displays a list of items looked up from another dataset based on the value of a field.
TDBLookupComboBox	Displays a list of items looked up from another dataset based on the value of a field, and also permits direct text entry like a standard data-aware edit box.

Data controls are data-aware at design time. When you associate the data control with an active dataset while building an application, you can immediately see live data in the control. You can use the Fields editor to scroll through a dataset at design time to verify that your application displays data correctly without having to compile and run the application. For more information about the Fields editor, see "Creating persistent fields" on page 17-4.

At runtime, data controls display data and, if your application, the control, and the dataset all permit it, a user can edit data through the control.

Associating a data control with a dataset

Data controls connect to datasets by using a data source. A data source component (*TDataSource*) acts as a conduit between the control and a dataset containing data. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component in order for their data to be displayed and manipulated in data-aware controls on a form.

Note Data source components are also required for linking unnested datasets in master-detail relationships.

To associate a data control with a dataset,

1. Place a dataset in a data module (or on a form), and set its properties as appropriate.
2. Place a data source in the same data module (or form). Using the Object Inspector, set its *DataSet* property to the dataset you placed in step 1.
3. Place a data control from the Data Access page of the Component palette onto a form.
4. Using the Object Inspector, set the *DataSource* property of the control to the data source component you placed in step 2.
5. Set the *DataField* property of the control to the name of a field to display, or select a field name from the drop-down list for the property. This step does not apply to *TDBGrid* and *TDBNavigator* because they access all available fields in the dataset.
6. Set the *Active* property of the dataset to *True* to display data in the control.

Changing the associated dataset at runtime

In the preceding example, the datasource was associated with its dataset by setting the *DataSet* property at design time. At runtime, you can switch the dataset for a data source component as needed. For example, the following code swaps the dataset for the *CustSource* data source component between the dataset components named *Customers* and *Orders*:

```
with CustSource do begin
  if (DataSet = Customers) then
    DataSet := Orders
  else
    DataSet := Customers;
end;
```

You can also set the *DataSet* property to a dataset on another form to synchronize the data controls on two forms. For example:

```
procedure TForm2.FormCreate (Sender : TObject);
begin
  DataSource1.DataSet := Form1.ClientDataSet1;
end;
```

Enabling and disabling the data source

The data source has an *Enabled* property that determines if it is connected to its dataset. When *Enabled* is *True*, the data source is connected to a dataset.

You can temporarily disconnect a single data source from its dataset by setting *Enabled* to *False*. When *Enabled* is *False*, all data controls attached to the data source component go blank and become inactive until *Enabled* is set to *True*. It is recommended, however, to control access to a dataset through a dataset component's *DisableControls* and *EnableControls* methods because they affect all attached data sources.

Responding to changes mediated by the data source

Because the data source provides the link between the data control and its dataset, it mediates all of the communication that occurs between the two. Typically, the data-aware control automatically responds to changes in the dataset. However, if your user interface is using controls that are not data-aware, you can use the events of a data source component to manually provide the same sort of response.

The *OnDataChange* event occurs whenever the data in a record may change, including field edits or when the cursor moves to a new record. This event is useful for making sure the control reflects the current field values in the dataset, because it is triggered by all changes. Typically, an *OnDataChange* event handler refreshes the value of a non-data-aware control that displays field data.

The *OnUpdateData* event occurs when the data in the current record is about to be posted. For instance, an *OnUpdateData* event occurs after *Post* is called, but before the data is actually posted to the change log.

The *OnStateChange* event occurs when the state of the dataset changes. When this event occurs, you can examine the dataset's *State* property to determine its current state.

For example, the following *OnStateChange* event handler enables or disables buttons or menu items based on the current state:

```
procedure Form1.DataSource1.StateChange(Sender: TObject);
begin
  CustTableEditBtn.Enabled := (CustTable.State = dsBrowse);
  CustTableCancelBtn.Enabled := CustTable.State in [dsInsert, dsEdit, dsSetKey];
  CustTableActivateBtn.Enabled := CustTable.State in [dsInactive];
  ⋮
end;
```

Note For more information about dataset states, see "Determining and setting dataset states" on page 16-3.

Editing and updating data

All data controls except the navigator display data from a database field. In addition, you can use them to edit and update data as long as the underlying dataset allows it.

Note Unidirectional datasets never permit users to edit and update data.

Enabling editing in controls on user entry

A dataset must be in *dsEdit* state to permit editing to its data. If the data source's *AutoEdit* property is *True* (the default), the data control handles the task of putting the dataset into *dsEdit* mode as soon as the user tries to edit its data.

If *AutoEdit* is *False*, you must provide an alternate mechanism for putting the dataset into edit mode. One such mechanism is to use a *TDBNavigator* control with an *Edit* button, which lets users explicitly put the dataset into edit mode. For more information about *TDBNavigator*, see "Navigating and manipulating records" on page 15-25. Alternately, you can write code that calls the dataset's *Edit* method when you want to put the dataset into edit mode.

Editing data in a control

A data control can only post edits to its associated dataset if the dataset's *CanModify* property is *True*. *CanModify* is always *False* for unidirectional datasets. Client datasets have a *ReadOnly* property that lets you specify whether *CanModify* is *True*.

Note Whether a client dataset can update data does not depend on whether its source dataset permits updates, but rather, it depends on whether the underlying database table permits updates. Thus, a client dataset can set *ReadOnly* to *False*, even if it fetches its data from a unidirectional dataset, which is always read-only. This is because client datasets can apply updates directly to the underlying database server.

Even if the dataset's *CanModify* property is *True*, the *Enabled* property of the data source that connects the dataset to the control must be *True* as well before the control can post updates back to the database table. The *Enabled* property of the data source determines whether the control can display field values from the dataset, and therefore also whether a user can edit and post values. If *Enabled* is *True* (the default), controls can display field values.

Finally, you can control whether the user can even enter edits to the data that is displayed in the control. The *ReadOnly* property of the data control determines if a user can edit the data displayed by the control. If *False* (the default), users can edit data. Clearly, you will want to ensure that the control's *ReadOnly* property is *True* when the dataset's *CanModify* property is *False*. Otherwise, you give users the false impression that they can affect the data in the underlying database table.

In all data controls except *TDBGrid*, when you modify a field, the modification is copied to the underlying dataset when you *Tab* from the control. If you press *Esc* before you *Tab* from a field, the data control abandons the modifications, and the value of the field reverts to the value it held before any modifications were made.

In *TDBGrid*, modifications are posted when you move to a different record; you can press *Esc* in any record of a field before moving to another record to cancel all changes to the record.

Client datasets cache all modifications to the data in a change log. These modifications are not applied to the underlying database table until you call the client dataset's *ApplyUpdates* method or, if you are storing data in a file on disk, until you call the client dataset's *MergeChangeLog* method and then save the client dataset.

Disabling and enabling data display

When your application iterates through a dataset or performs a search, you should temporarily prevent refreshing of the values displayed in data-aware controls each time the current record changes. Preventing refreshing of values speeds the iteration or search and prevents annoying screen-flicker.

DisableControls is a dataset method that disables display for all data-aware controls linked to a dataset. As soon as the iteration or search is over, your application should immediately call the dataset's *EnableControls* method to re-enable display for the controls.

Usually you disable controls before entering an iterative process. The iterative process itself should take place inside a **try...finally** statement so that you can re-enable controls even if an exception occurs during processing. The **finally** clause should call *EnableControls*. The following code illustrates how you might use *DisableControls* and *EnableControls* in this manner:

```
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets EOF False }
  while not CustTable.EOF do { Cycle until EOF is True }
  begin
    { Process each record here }
    ⋮
    CustTable.Next; { EOF False on success; EOF True when Next fails on last record }
  end;
finally
  CustTable.EnableControls;
end;
```

Refreshing data display

The *Refresh* method for a dataset flushes local buffers and refetches data for an open dataset. You can use this method to update the display in data-aware controls if you think that the underlying data has changed because other applications have simultaneous access to the data used in your application. However, before you refresh the dataset, be sure to apply any updates the dataset has currently cached. Client datasets raise an exception if you attempt to refresh the data before applying pending updates.

Refreshing can sometimes lead to unexpected results. For example, if a user is viewing a record deleted by another application, then the record disappears the moment your application calls *Refresh*. Data can also appear to change if another user changes a record after you originally fetched the data and before you call *Refresh*.

Enabling mouse, keyboard, and timer events

The *Enabled* property of a data control determines whether it responds to mouse, keyboard, or timer events, and passes information to its data source. The default setting for this property is *True*.

To prevent mouse, keyboard, or timer events from reaching a data control, set its *Enabled* property to *False*. When *Enabled* is *False*, the data source that connects the control to its dataset does not receive information from the data control. The data control continues to display data, but the text displayed in the control is dimmed.

Choosing how to organize the data

When you build the user interface for your database application, you have choices to make about how you want to organize the display of information and the controls that manipulate that information.

One of the first decisions to make is whether you want to display a single record at a time, or multiple records.

In addition, you will want to add controls to navigate and manipulate records. The *TDBNavigator* control provides built-in support for many of the functions you may want to perform.

Displaying a single record

In many applications, you may only want to provide information about a single record of data at a time. For example, an order-entry application may display the information about a single order without indicating what other orders are currently logged. This information probably comes from a single record in an orders dataset. If you are connecting your user interface directly to a unidirectional dataset, a single-record user interface is the only available possibility.

Applications that display a single record are usually easy to read and understand, because all database information is about the same thing (in the previous case, the same order). The data-aware controls in these user interfaces represent a single field from a database record. The Data Controls page of the Component palette provides a wide selection of controls to represent different kinds of fields. These controls are typically data-aware versions of other controls that are available on the component palette. For example, the *TDBEdit* control is a data-aware version of the standard *TEdit* control which enables users to see and edit a text string.

Which control you use depends on the type of data (text, formatted text, graphics, boolean information, and so on) contained in the field.

Displaying data as labels

TDBText is a read-only control similar to the *TLabel* component on the Standard page of the Component palette. A *TDBText* control is useful when you want to provide display-only data on a form that allows user input in other controls. For example, suppose a form is created around the fields in a customer list table, and that once the user enters a street address, city, and state or province information in the form, you use a dynamic lookup to automatically determine the zip code field from a separate table. A *TDBText* component tied to the zip code table could be used to display the zip code field that matches the address entered by the user.

TDBText gets the text it displays from a specified field in the current record of a dataset. Because *TDBText* gets its text from a dataset, the text it displays is dynamic—the text changes as the user navigates the database table. Therefore you cannot specify the display text of *TDBText* at design time as you can with *TLabel*.

Note When you place a *TDBText* component on a form, make sure its *AutoSize* property is *True* (the default) to ensure that the control resizes itself as necessary to display data of varying widths. If *AutoSize* is *False*, and the control is too small, data display is clipped.

Displaying and editing fields in an edit box

TDBEdit is a data-aware version of an edit box component. *TDBEdit* displays the current value of a data field to which it is linked and permits it to be edited using standard edit box techniques.

For example, suppose *CustomersSource* is a *TDataSource* component that is active and linked to an open *TSQLClientDataSet* called *CustomersTable*. You can then place a *TDBEdit* component on a form and set its properties as follows:

- *DataSource*: CustomersSource
- *DataField*: CustNo

The data-aware edit box component immediately displays the value of the current row of the *CustNo* column of the *CustomersTable* dataset, both at design time and at runtime.

Displaying and editing text in a memo control

TDBMemo is a data-aware component—similar to the standard *TMemo* component—that can display lengthy text data. *TDBMemo* displays multi-line text, and permits a user to enter multi-line text as well. You can use *TDBMemo* controls to display large text fields or text data contained in binary large object (BLOB) fields.

By default, *TDBMemo* permits a user to edit memo text. To prevent editing, set the *ReadOnly* property of the memo control to *True*. To display tabs and permit users to enter them in a memo, set the *WantTabs* property to *True*. To limit the number of characters users can enter into the database memo, use the *MaxLength* property. The default value for *MaxLength* is 0, meaning that there is no character limit other than that imposed by the operating system.

Several properties affect how the database memo appears and how text is entered. To prevent word wrap, set the *WordWrap* property to *False*. The *Alignment* property determines how the text is aligned within the control. Possible choices are *taLeftJustify* (the default), *taCenter*, and *taRightJustify*. To change the font of the text, use the *Font* property.

At runtime, users can cut, copy, and paste text to and from a database memo control. You can accomplish the same task programmatically by using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods.

Because the *TDBMemo* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes to scroll through data records, *TDBMemo* has an *AutoDisplay* property that controls whether the accessed data should be displayed automatically. If you set *AutoDisplay* to *False*, *TDBMemo* displays the field name rather than actual data. Double-click inside the control to view the actual data.

Displaying and editing graphics fields in an image control

TDBImage is a data-aware control that displays graphics contained in BLOB fields.

By default, *TDBImage* permits a user to edit a graphics image by cutting and pasting to and from the clipboard using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods. You can, instead, supply your own editing methods attached to the event handlers for the control.

By default, an image control displays as much of a graphic as fits in the control, cropping the image if it is too big. You can set the *Stretch* property to *True* to resize the graphic to fit within an image control as it is resized.

Because the *TDBImage* can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes scroll through data records, *TDBImage* has an *AutoDisplay* property that controls whether the accessed data should automatically displayed. If you set *AutoDisplay* to *False*, *TDBImage* displays the field name rather than actual data. Double-click inside the control to view the actual data.

Displaying and editing data in list and combo boxes

There are four data controls that provide the user with a set of default data values to choose from at runtime. These are data-aware versions of standard list box and combo box controls:

- *TDBListBox*, which displays a scrollable list of items from which a user can choose to enter in a data field. A data-aware list box displays a default value for a field in the current record and highlights its corresponding entry in the list. If the current row's field value is not in the list, no value is highlighted in the list box. When a user selects a list item, the corresponding field value is changed in the underlying dataset.
- *TDBComboBox*, which combines the functionality of a data-aware edit control and a drop-down list. At runtime it can display a drop-down list from which a user can pick from a predefined set of values, and it can permit a user to enter an entirely different value.
- *TDBLookupListBox*, which behaves like *TDBListBox* except the list of display items is looked up in another dataset.
- *TDBLookupComboBox*, which behaves like *TDBComboBox* except the list of display items is looked up in another dataset.

Note At runtime, users can use an incremental search to find list box items. When the control has focus, for example, typing 'ROB' selects the first item in the list box beginning with the letters 'ROB'. Typing an additional 'E' selects the first item starting with 'ROBE', such as 'Robert Johnson'. The search is case-insensitive. *Backspace* and *Esc* cancel the current search string (but leave the selection intact), as does a two second pause between keystrokes.

Using TDBListBox and TDBComboBox

When using *TDBListBox* or *TDBComboBox*, you must use the String List editor at design time to create the list of items to display. To bring up the String List editor, click the ellipsis button for the *Items* property in the Object Inspector. Then type in the items that you want to have appear in the list. At runtime, use the methods of the *Items* property to manipulate its string list. You can specify that the items in the list should be displayed in alphabetical order by setting the *Sorted* property to *True*.

When a *TDBListBox* or *TDBComboBox* control is linked to a field through its *DataField* property, the field value appears selected in the list. If the current value is not in the list, no item appears selected. However, *TDBComboBox* displays the current value for the field in its edit box, regardless of whether it appears in the *Items* list.

The items in the list of *TDBListBox* or the drop-down list of *TDBComboBox* each have the height specified by *ItemHeight*. To ensure that the bottom on the list is fully visible in *TDBListBox*, you may therefore want to set the *Height* property to a multiple of *ItemHeight*. On *TDBComboBox*, this is not necessary, because the size of the drop-down list is not controlled by the *Height* property. Instead, you can set the *DropDownCount*: the maximum number of items displayed in the list. If the number of items in the list exceeds the size of the list (as determined by *Height* on *TDBListBox* or *DropDownCount* on *TDBComboBox*), the user can scroll the list.

For *TDBComboBox*, the *Style* property determines user interaction with the control. By default, *Style* is *csDropDown*, meaning a user can enter values from the keyboard, or choose an item from the drop-down list. The possible values of *Style* are as follows:

- *csDropDown* (default): Displays a drop-down list with an edit box in which the user can enter text. All items are strings and have the same height.
- *csDropDownList*: Displays a drop-down list and edit box, but the user cannot enter or change values that are not in the drop-down list at runtime.
- *csOwnerDrawFixed* and *csOwnerDrawVariable*: Allows the items list to display values other than strings (for example, bitmaps) or to use different fonts for individual items in the list.

Displaying and editing data in lookup list and combo boxes

Lookup list boxes and lookup combo boxes (*TDBLookupListBox* and *TDBLookupComboBox*) present the user with a restricted list of choices from which to set a valid field value. When a user selects a list item, the corresponding field value is changed in the underlying dataset.

For example, consider an order form whose fields are tied to the *OrdersTable*. *OrdersTable* contains a *CustNo* field corresponding to a customer ID, but *OrdersTable* does not have any other customer information. The *CustomersTable*, on the other hand, contains a *CustNo* field corresponding to a customer ID, and also contains additional information, such as the customer's company and mailing address. It would be convenient if the order form enabled a clerk to select a customer by company name instead of customer ID when creating an invoice. A *TDBLookupListBox* that displays all company names in *CustomersTable* enables a user to select the company name from the list, and set the *CustNo* on the order form appropriately.

These lookup controls derive the list of display items from one of two sources:

- **A lookup field defined for a dataset.** To specify list box items using a lookup field, the dataset to which you link the control must already define a lookup field. (This process is described in "Defining a lookup field" on page 17-8). To specify the lookup field for the list box items,

  1. Set the *DataSource* property of the list box to the data source for the dataset containing the lookup field to use.
  2. Choose the lookup field to use from the drop-down list for the *DataField* property.

  When you activate a table associated with a lookup control, the control recognizes that its data field is a lookup field, and displays the appropriate values from the lookup.

- **A secondary data source, data field, and key**. If you have not defined a lookup field for a dataset, you can establish a similar relationship using a secondary data source, a field value to search on in the secondary data source, and a field value to return as a list item. To specify a secondary data source for list box items,

  1. Set the *DataSource* property of the list box to the data source for the control.
  2. Choose a field into which to insert looked-up values from the drop-down list for the *DataField* property. The field you choose cannot be a lookup field.
  3. Set the *ListSource* property of the list box to the data source for the dataset that contain the field whose values you want to look up.
  4. Choose a field to use as a lookup key from the drop-down list for the *KeyField* property. The drop-down list displays fields for the dataset associated with data source you specified in Step 3. The field you choose need not be part of an index, but if it is, lookup performance is even faster.
  5. Choose a field whose values to return from the drop-down list for the *ListField* property. The drop-down list displays fields for the dataset associated with the data source you specified in Step 3.

  When you activate a table associated with a lookup control, the control recognizes that its list items are derived from a secondary source, and displays the appropriate values from that source.

To specify the number of items that appear at one time in a *TDBLookupListBox* control, use the *RowCount* property. The height of the list box is adjusted to fit this row count exactly.

To specify the number of items that appear in the drop-down list of *TDBLookupComboBox*, use the *DropDownRows* property instead.

Note You can also set up a column in a data grid to act as a lookup combo box. For information on how to do this, see "Defining a lookup list column" on page 15-19.

Handling Boolean field values with check boxes

TDBCheckBox is a data-aware check box control. It can be used to set the values of Boolean fields in a dataset. For example, a customer invoice form might have a check box control that when checked indicates the customer is tax-exempt, and when unchecked indicates that the customer is not tax-exempt.

The data-aware check box control manages its checked or unchecked state by comparing the value of the current field to the contents of *ValueChecked* and *ValueUnchecked* properties. If the field value matches the *ValueChecked* property, the control is checked. Otherwise, if the field matches the *ValueUnchecked* property, the control is unchecked.

Note The values in *ValueChecked* and *ValueUnchecked* cannot be identical.

Set the *ValueChecked* property to a value the control should post to the database if the control is checked when the user moves to another record. By default, this value is set to "true," but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of *ValueChecked*. If any of the items matches the contents of that field in the current record, the check box is checked. For example, you can specify a *ValueChecked* string like:

```
DBCheckBox1.ValueChecked := 'True;Yes;On';
```

If the field for the current record contains values of "true," "Yes," or "On," then the check box is checked. Comparison of the field to *ValueChecked* strings is case-insensitive. If a user checks a box for which there are multiple *ValueChecked* strings, the first string is the value that is posted to the database.

Set the *ValueUnchecked* property to a value the control should post to the database if the control is not checked when the user moves to another record. By default, this value is set to "false," but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of *ValueUnchecked*. If any of the items matches the contents of that field in the current record, the check box is unchecked.

A data-aware check box is disabled whenever the field for the current record does not contain one of the values listed in the *ValueChecked* or *ValueUnchecked* properties.

If the field with which a check box is associated is a logical field, the check box is always checked if the contents of the field is *True*, and it is unchecked if the contents of the field is *False*. In this case, strings entered in the *ValueChecked* and *ValueUnchecked* properties have no effect on logical fields.

Restricting field values with radio controls

TDBRadioGroup is a data-aware version of a radio group control. It enables you to set the value of a data field with a radio button control where there is a limited number of possible values for the field. The radio group includes one button for each value a field can accept. Users can set the value for a data field by selecting the desired radio button.

The *Items* property determines the radio buttons that appear in the group. *Items* is a string list. One radio button is displayed for each string in *Items*, and each string appears to the right of a radio button as the button's label.

If the current value of a field associated with a radio group matches one of the strings in the *Items* property, that radio button is selected. For example, if three strings, "Red," "Yellow," and "Blue," are listed for *Items*, and the field for the current record contains the value "Blue," then the third button in the group appears selected.

Note If the field does not match any strings in *Items*, a radio button may still be selected if the field matches a string in the *Values* property. If the field for the current record does not match any strings in *Items* or *Values*, no radio button is selected.

The *Values* property can contain an optional list of strings that can be returned to the dataset when a user selects a radio button and posts a record. Strings are associated with buttons in numeric sequence. The first string is associated with the first button, the second string with the second button, and so on. For example, suppose *Items* contains "Red," "Yellow," and "Blue," and *Values* contains "Magenta," "Yellow," and "Cyan." If a user selects the button labeled "Red," "Magenta" is posted to the database.

If strings for *Values* are not provided, the *Item* string for a selected radio button is returned to the database when a record is posted.

Displaying multiple records

Sometimes you want to display many records in the same form. For example, an invoicing application might show all the orders made by a single customer on the same form.

To display multiple records, use a grid control. Grid controls provide a multi-field, multi-record view of data that can make your application's user interface more compelling and effective. They are discussed in "Viewing and editing data with TDBGrid" on page 15-14.

Note You can't display multiple records when using a unidirectional dataset.

You may want to design a user interface that displays both fields from a single record and grids that represent multiple records. There are two models that combine these two approaches:

- **Master-detail forms:** You can represent information from both a master table and a detail table by including both controls that display a single field and grid controls. For example, you could display information about a single customer with a detail grid that displays the orders for that customer. For information about linking the underlying tables in a master-detail form, see "Setting up master/detail relationships" on page 18-13 and "Making the client dataset a detail of another dataset" on page 20-10.

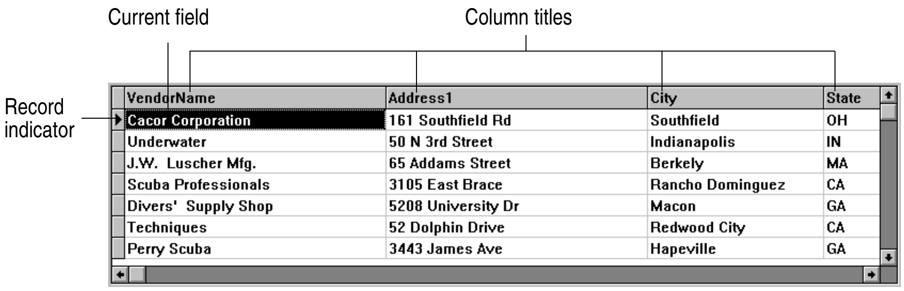
- **Drill-down forms**: In a form that displays multiple records, you can include single field controls that display detailed information from the current record only. This approach is particularly useful when the records include long memos or graphic information. As the user scrolls through the records of the grid, the memo or graphic updates to represent the value of the current record. Setting this up is very easy. The synchronization between the two displays is automatic if the grid and the memo or image control share a common data source.

Tip It is generally not a good idea to combine these two approaches on a single form. While the result can sometimes be effective, it can be confusing for users to understand the data relationships.

Viewing and editing data with TDBGrid

A *TDBGrid* control lets you view and edit records from a dataset in a tabular grid format.

Figure 15.1 TDBGrid control



VendorName	Address1	City	State
Cacor Corporation	161 Southfield Rd	Southfield	OH
Underwater	50 N 3rd Street	Indianapolis	IN
J.W. Luscher Mfg.	65 Addams Street	Berkely	MA
Scuba Professionals	3105 East Brace	Rancho Dominguez	CA
Divers' Supply Shop	5208 University Dr	Macon	GA
Techniques	52 Dolphin Drive	Redwood City	CA
Perry Scuba	3443 James Ave	Hapeville	GA

Three factors affect the appearance of records displayed in a grid control:

- Existence of persistent column objects defined for the grid using the Columns editor. Persistent column objects provide great flexibility setting grid and data appearance. For information on using persistent columns, see “Creating a customized grid” on page 15-15.
- Creation of persistent field components for the dataset displayed in the grid. For information about creating persistent field components using the Fields editor, see Chapter 17, “Working with field components.”
- The dataset’s *ObjectView* property setting for grids displaying nested dataset or array fields. For information on displaying such composite fields in a grid, see “Displaying composite fields” on page 15-21.

A grid control has a *Columns* property that is itself a wrapper on a *TDBGridColumns* object. *TDBGridColumns* is a collection of *TColumn* objects representing all of the columns in a grid control. You can use the Columns editor to set up column attributes at design time, or use the *Columns* property of the grid to access the properties, events, and methods of *TDBGridColumns* at runtime.

Using a grid control in its default state

The *State* property of the grid's *Columns* property indicates whether persistent column objects exist for the grid. *Columns.State* is a runtime-only property that is automatically set for a grid. The default state is *csDefault*, meaning that persistent column objects do not exist for the grid. In that case, the display of data in the grid is determined primarily by the properties of the fields in the grid's dataset, or, if there are no persistent field components, by a default set of display characteristics.

When the grid's *Columns.State* property is *csDefault*, grid columns are dynamically generated from the visible fields of the dataset and the order of columns in the grid matches the order of fields in the dataset. Every column in the grid is associated with a field component. Property changes to field components immediately show up in the grid.

Using a grid control with dynamically-generated columns is useful for viewing and editing the contents of arbitrary tables selected at runtime. Because the grid's structure is not set, it can change dynamically to accommodate different datasets. A single grid with dynamically-generated columns can display a Local InterBase table at one moment, then switch to display the results of a MySQL query when the grid's *DataSource* property changes or the *DataSet* property of the data source changes.

You can change the appearance of a dynamic column at design time or runtime, but what you are actually modifying are the corresponding properties of the field component displayed in the column. Properties of dynamic columns exist only so long as a column is associated with a particular field in a single dataset. For example, changing the *Width* property of a column changes the *DisplayWidth* property of the field associated with that column. Changes made to column properties that are not based on field properties, such as *Font*, exist only for the lifetime of the column.

If a grid's dataset consists of dynamic field components, the fields are destroyed each time the dataset is closed. When the field components are destroyed, all dynamic columns associated with them are destroyed as well. If a grid's dataset consists of persistent field components, the field components exist even when the dataset is closed, so the columns associated with those fields also retain their properties when the dataset is closed.

Note Changing a grid's *Columns.State* property to *csDefault* at runtime deletes all column objects in the grid (even persistent columns), and rebuilds dynamic columns based on the visible fields of the grid's dataset.

Creating a customized grid

A customized grid is one for which you define persistent column objects that describe how a column appears and how the data in the column is displayed. A customized grid lets you configure multiple grids to present different views of the same dataset (different column orders, different field choices, and different column colors and fonts, for example). A customized grid also enables you to let users modify the appearance of the grid at runtime without affecting the fields used by the grid or the field order of the dataset.

Customized grids are best used with datasets whose structure is known at design time. Because they expect field names established at design time to exist in the dataset, customized grids are not well suited to browsing arbitrary tables selected at runtime.

Understanding persistent columns

When you create persistent column objects for a grid, they are only loosely associated with underlying fields in a grid's dataset. Default property values for persistent columns are dynamically fetched from a default source (the associated field or the grid itself) until a value is assigned to the column property. Until you assign a column property a value, its value changes as its default source changes. Once you assign a value to a column property, it no longer changes when its default source changes.

For example, the default source for a column title caption is an associated field's *DisplayLabel* property. If you modify the *DisplayLabel property*, the column title reflects that change immediately. If you then assign a string to the column title's caption, the title caption is independent of the associated field's *DisplayLabel* property. Subsequent changes to the field's *DisplayLabel* property no longer affect the column's title.

Persistent columns exist independently from field components with which they are associated. In fact, persistent columns need not be associated with field objects at all. If a persistent column's *FieldName* property is blank, or if the field name does not match the name of any field in the grid's current dataset, the column's *Field* property is NULL and the column is drawn with blank cells. If you override the cell's default drawing method, you can display your own custom information in the blank cells. For example, you can use a blank column to display aggregated values on the last record of a group of records that the aggregate summarizes. Another possibility is to display a bitmap or bar chart that graphically depicts some aspect of a record's data.

Two or more persistent columns can be associated with the same field in a dataset. For example, you might display a part number field at the left and right extremes of a wide grid to make it easier to find the part number without having to scroll the grid.

Note Because persistent columns do not have to be associated with a field in a dataset, and because multiple columns can reference the same field, a customized grid's *FieldCount* property can be less than or equal to the grid's column count. Also note that if the currently selected column in a customized grid is not associated with a field, the grid's *SelectedField* property is NULL and the *SelectedIndex* property is –1.

Persistent columns can be configured to display grid cells as a combo box drop-down list of lookup values from another dataset or from a static pick list, or as an ellipsis button (…) in a cell that users can click to launch special data viewers or dialogs related to the current cell.

Creating persistent columns

To customize the appearance of grid at design time, you invoke the Columns editor to create a set of persistent column objects for the grid. At runtime, the *State* property for a grid with persistent column objects is automatically set to *csCustomized*.

To create persistent columns for a grid control,

1 Select the grid component in the form.

2 Invoke the Columns editor by double clicking on the grid's *Columns* property in the Object Inspector.

The Columns list box displays the persistent columns that have been defined for the selected grid. When you first bring up the Columns editor, this list is empty because the grid is in its default state, containing only dynamic columns.

You can create persistent columns for all fields in a dataset at once, or you can create persistent columns on an individual basis. To create persistent columns for all fields:

1 Right-click the grid to invoke the context menu and choose Add All Fields. Note that if the grid is not already associated with a data source, Add All Fields is disabled. Associate the grid with a data source that has an active dataset before choosing Add All Fields.

2 If the grid already contains persistent columns, a dialog box asks if you want to delete the existing columns, or append to the column set. If you choose Yes, any existing persistent column information is removed, and all fields in the current dataset are inserted by field name according to their order in the dataset. If you choose No, any existing persistent column information is retained, and new column information, based on any additional fields in the dataset, are appended to the dataset.

3 Click Close to apply the persistent columns to the grid and close the dialog box.

To create persistent columns individually:

1 Choose the Add button in the Columns editor. The new column will be selected in the list box. The new column is given a sequential number and default name (for example, 0 - TColumn).

2 To associate a field with this new column, set the *FieldName* property in the Object Inspector.

3 To set the title for the new column, expand the *Title* property in the Object Inspector and set its *Caption* property.

4 Close the Columns editor to apply the persistent columns to the grid and close the dialog box.

At runtime, you can switch to persistent columns by assigning *csCustomized* to the *Columns.State* property. Any existing columns in the grid are destroyed and new persistent columns are built for each field in the grid's dataset. You can then add a persistent column at runtime by calling the *Add* method for the column list:

```
DBGrid1.Columns.Add;
```

Deleting persistent columns

Deleting a persistent column from a grid is useful for eliminating fields that you do not want to display. To remove a persistent column from a grid,

1 Double-click the grid to display the Columns editor.

2 Select the field to remove in the Columns list box.

3 Click Delete (you can also use the context menu or *Del* key, to remove a column).

Note If you delete all the columns from a grid, the *Columns.State* property reverts to its *csDefault* state and automatically build dynamic columns for each field in the dataset.

You can delete a persistent column at runtime by simply freeing the column object:

```
DBGrid1.Columns[5].Free;
```

Arranging the order of persistent columns

The order in which columns appear in the Columns editor is the same as the order the columns appear in the grid. You can change the column order by dragging and dropping columns within the Columns list box.

To change the order of a column,

1 Select the column in the Columns list box.

2 Drag it to a new location in the list box.

You can also change the column order by clicking on the column title of the actual grid and dragging the column to a new position, just as you can at runtime.

Note Reordering persistent fields in the Fields editor also reorders columns in a default grid, but not a custom grid.

Important You cannot reorder columns in grids containing both dynamic columns and dynamic fields at design time, since there is nothing persistent to record the altered field or column order.

At runtime, a user can use the mouse to drag a column to a new location in the grid if its *DragMode* property is set to *dmManual*. Reordering the columns of a grid with a *State* property of *csDefault* state also reorders field components in the dataset underlying the grid. The order of fields in the physical table is not affected. To prevent a user from rearranging columns at runtime, set the grid's *DragMode* property to *dmAutomatic*.

At runtime, the grid's *OnColumnMoved* event is fired after a column has been moved.

Setting column properties at design time

Column properties determine how data is displayed in the cells of that column. Most column properties obtain their default values from properties associated with another component, called the *default source*, such as a grid or an associated field component.

To set a column's properties, select the column in the Columns editor and set its properties in the Object Inspector. The following table summarizes key column properties you can set.

Table 15.2 Column properties

Property	Purpose
Alignment	Left justifies, right justifies, or centers the field data in the column. Default source: *TField.Alignment*.
ButtonStyle	*cbsAuto*: (default) Displays a drop-down list if the associated field is a lookup field, or if the column's *PickList* property contains data. *cbsEllipsis*: Displays an ellipsis (...) button to the right of the cell. Clicking on the button fires the grid's *OnEditButtonClick* event. *cbsNone*: The column uses only the normal edit control to edit data in the column.
Color	Specifies the background color of the cells of the column. Default Source: *TDBGrid.Color. (*For text foreground color, see the Font property.)
DropDownRows	The number of lines of text displayed by the drop-down list. Default: 7.
Expanded	Specifies whether the column is expanded. Only applies to columns representing composite fields.
FieldName	Specifies the field name associated with this column. This can be blank.
ReadOnly	*True:* The data in the column cannot be edited by the user. *False:* (default) The data in the column can be edited.
Width	Specifies the width of the column in screen pixels. Default Source: derived from *TField.DisplayWidth*.
Font	Specifies the font type, size, and color used to draw text in the column. Default Source: *TDBGrid.Font*.
PickList	Contains a list of values to display in a drop-down list in the column.
Title	Sets properties for the title of the selected column.

The following table summarizes the options you can specify for the *Title* property.

Table 15.3 Expanded TColumn Title properties

Property	Purpose
Alignment	Left justifies (default), right justifies, or centers the caption text in the column title.
Caption	Specifies the text to display in the column title. Default Source: *TField.DisplayLabel*.
Color	Specifies the background color used to draw the column title cell. Default Source: *TDBGrid.FixedColor*.
Font	Specifies the font type, size, and color used to draw text in the column title. Default Source: *TDBGrid.TitleFont*.

Defining a lookup list column

You can create a column that displays a drop-down list of values, similar to a lookup combo box control. To specify that the column acts like a combo box, set the column's *ButtonStyle* property to *cbsAuto*. Once you populate the list with values, the grid automatically displays a combo box-like drop-down button when a cell of that column is in edit mode.

There are two ways to populate that list with the values for users to select:

- You can fetch the values from a lookup table. To make a column display a drop-down list of values drawn from a separate lookup table, you must define a lookup field in the dataset. For information about creating lookup fields, see "Defining a lookup field" on page 17-8. Once the lookup field is defined, set the column's *FieldName* to the lookup field name. The drop-down list is automatically populated with lookup values defined by the lookup field.
- You can specify a list of values explicitly at design time. To enter the list values at design time, double-click the *PickList* property for the column in the Object Inspector. This brings up the String List editor, where you can enter the values that populate the pick list for the column.

By default, the drop-down list displays 7 values. You can change the length of this list by setting the *DropDownRows* property.

Note To restore a column with an explicit pick list to its normal behavior, delete all the text from the pick list using the String List editor.

Putting a button in a column

A column can display an ellipsis button (…) to the right of the normal cell editor. *Ctrl+Enter* or a mouse click fires the grid's *OnEditButtonClick* event. You can use the ellipsis button to bring up forms containing more detailed views of the data in the column. For example, in a table that displays summaries of invoices, you could set up an ellipsis button in the invoice total column to bring up a form that displays the items in that invoice, or the tax calculation method, and so on. For graphic fields, you could use the ellipsis button to bring up a form that displays an image.

To create an ellipsis button in a column:

1. Select the column in the *Columns* list box.
2. Set *ButtonStyle* to *cbsEllipsis*.
3. Write an *OnEditButtonClick* event handler.

Restoring default values to a column

At runtime you can test a column's *AssignedValues* property to determine whether a column property has been explicitly assigned. Values that are not explicitly defined are dynamically based on the associated field or the grid's defaults.

You can undo property changes made to one or more columns. In the Columns editor, select the column or columns to restore, and then select Restore Defaults from the context menu. Restore defaults discards assigned property settings and restores a column's properties to those derived from its underlying field component

At runtime, you can reset all default properties for a single column by calling the column's *RestoreDefaults* method. You can also reset default properties for all columns in a grid by calling the column list's *RestoreDefaults* method:

```
DBGrid1.Columns.RestoreDefaults;
```

Displaying composite fields

Sometimes the fields of the grid's dataset do not represent simple values such as text, graphics, numerical values, and so on. Some database servers allow fields that are a composite of simpler data types, such as ADT fields or array fields.

There are two ways a grid can display composite fields:

- It can "flatten out" a composite field type so that each of the simpler types that make up the field appears as a separate field in the dataset. When a composite field is flattened out, its constituents appear as separate fields that reflect their common source only in that each field name is preceded by the name of the common parent field in the underlying database table.

  To display composite fields as if they were flattened out, set the dataset's *ObjectView* property to *False*. The dataset stores composite fields as a set of separate fields, and the grid reflects this by assigning each constituent part a separate column.

- It can display composite fields in a single column, reflecting the fact that they are a single field. When displaying composite fields in a single column, the column can be expanded and collapsed by clicking on the arrow in the title bar of the field, or by setting the *Expanded* property of the column:

  - When a column is expanded, each child field appears in its own sub-column with a title bar that appears below the title bar of the parent field. That is, the title bar for the grid increases in height, with the first row giving the name of the composite field, and the second row subdividing that for the individual parts. Fields that are not composites appear with title bars that are extra high. This expansion continues for constituents that are in turn composite fields (for example, a detail table nested in a detail table), with the title bar growing in height accordingly.
  - When the field is collapsed, only one column appears with an uneditable comma delimited string containing the child fields.

  To display a composite field in an expanding and collapsing column, set the dataset's *ObjectView* property to *True*. The dataset stores the composite field as a single field component that contains a set of nested sub-fields. The grid reflects this in a column that can expand or collapse.

Table 15.4 Properties that affect the way composite fields appear

Property	Object	Purpose
Expandable	TColumn	Indicates whether the column can be expanded to show child fields in separate, editable columns. (read-only)
Expanded	TColumn	Specifies whether the column is expanded.
MaxTitleRows	TDBGrid	Specifies the maximum number of title rows that can appear in the grid
ObjectView	TDataSet	Specifies whether fields are displayed flattened out, or in object mode, where each object field can be expanded and collapsed.
ParentColumn	TColumn	Refers to the TColumn object that owns the child field's column.

Note In addition to Composite fields, some datasets include fields that refer to another dataset (dataset fields) or a record in another dataset (reference) fields. Data-aware grids display such fields as "(DataSet)" or "(Reference)", respectively. At runtime an ellipsis button appears to the right. Clicking on the ellipsis brings up a new form with a grid displaying the contents of the field. For dataset fields, this grid displays the dataset that is the field's value. For reference fields, this grid contains a single row that displays the record from another dataset.

Setting grid options

You can use the grid *Options* property at design time to control basic grid behavior and appearance at runtime. When a grid component is first placed on a form at design time, the *Options* property in the Object Inspector is displayed with a + (plus) sign to indicate that the *Options* property can be expanded to display a series of Boolean properties that you can set individually. To view and set these properties, click on the + sign. The list of options in the Object Inspector below the *Options* property. The + sign changes to a – (minus) sign, that collapses the list back when you click it.

The following table lists the *Options* properties that can be set, and describes how they affect the grid at runtime.

Table 15.5 Expanded TDBGrid Options properties

Option	Purpose
dgEditing	*True*: (Default). Enables editing, inserting, and deleting records in the grid.
	False: Disables editing, inserting, and deleting records in the grid.
dgAlwaysShowEditor	*True:* When a field is selected, it is in Edit state.
	False: (Default). A field isn't automatically in Edit state when selected.
dgTitles	*True:* (Default). Displays field names across the top of the grid.
	False: Field name display is turned off.
dgIndicator	*True:* (Default). The indicator column is displayed at the left of the grid, and the current record indicator (an arrow at the left of the grid) is activated to show the current record. On insert, the arrow becomes an asterisk. On edit, the arrow becomes an I-beam.
	False: The indicator column is turned off.
dgColumnResize	*True:* (Default). Columns can be resized by dragging the column rulers in the title area. Resizing can change the corresponding width of the underlying *TField* component.
	False: Columns cannot be resized in the grid.
dgColLines	*True:* (Default). Displays vertical dividing lines between columns.
	False: Does not display dividing lines between columns.
dgRowLines	*True:* (Default). Displays horizontal dividing lines between records.
	False: Does not display dividing lines between records.
dgTabs	*True:* (Default). Enables tabbing between fields in records.
	False: Tabbing exits the grid control.

Table 15.5 Expanded TDBGrid Options properties (continued)

Option	Purpose
dgRowSelect	*True:* The selection bar spans the entire width of the grid.
	False: (Default). Selecting a field in a record selects only that field.
dgAlwaysShowSelection	*True:* (Default). The selection bar in the grid is always visible, even if another control has focus.
	False: The selection bar in the grid is only visible when the grid has focus.
dgConfirmDelete	*True:* (Default). Prompt for confirmation to delete records (*Ctrl+Del*).
	False: Delete records without confirmation.
dgCancelOnExit	*True:* (Default). Cancels a pending insert when focus leaves the grid. This option prevents inadvertent posting of partial or blank records.
	False: Permits pending inserts.
dgMultiSelect	*True:* Allows user to select noncontiguous rows in the grid using *Ctrl+Shift* or *Shift+ arrow* keys.
	False: (Default). Does not allow user to multi-select rows.

Editing in the grid

At runtime, you can use a grid to modify existing data and enter new records, if the following default conditions are met:

- The *CanModify property* of the *Dataset* is *True*.
- The *ReadOnly* property of grid is *False*.

When a user edits a record in the grid, changes to each field are posted to an internal record buffer, but are not posted until the user moves to a different record in the grid. Even if focus changes to another control on a form, the grid does not post changes until the cursor for the dataset moves to another record. When a record is posted, the dataset checks all associated data-aware components for a change in status. If there is a problem updating any fields that contain modified data, it raises an exception, and does not modify the record.

Note Posting record changes to a client dataset only adds them to the client dataset's internal change log. They are not posted back to the underlying database table until the client dataset applies its updates.

You can cancel all edits for a record by pressing *Esc* in any field before moving to another record.

Controlling grid drawing

Your first level of control over how a grid control draws itself is setting column properties. The grid automatically uses the font, color, and alignment properties of a column to draw the cells of that column. The text of data fields is drawn using the *DisplayFormat* or *EditFormat* properties of the field component associated with the column.

You can augment the default grid display logic with code in a grid's *OnDrawColumnCell* event. If the grid's *DefaultDrawing* property is *True*, all the normal drawing is performed before your *OnDrawColumnCell* event handler is called. Your code can then draw on top of the default display. This is primarily useful when you have defined a blank persistent column and want to draw special values in that column's cells.

If you want to replace the drawing logic of the grid entirely, set *DefaultDrawing* to *False* and place your drawing code in the grid's *OnDrawColumnCell* event. If you want to replace the drawing logic only in certain columns or for certain field data types, you can call the *DefaultDrawColumnCell* method inside your *OnDrawColumnCell* event handler to have the grid use its normal drawing code for selected columns. This reduces the amount of work you have to do if you only want to change the way Boolean field types are drawn, for example.

Responding to user actions at runtime

You can modify grid behavior by writing event handlers to respond to specific actions within the grid at runtime. Because a grid typically displays many fields and records at once, you may have very specific needs to respond to changes to individual columns. For example, you might want to activate and deactivate a button elsewhere on the form every time a user enters and exits a specific column.

The following table lists the grid events available in the Object Inspector.

Table 15.6 Grid control events

Event	Purpose
OnCellClick	Occurs when a user clicks on a cell in the grid.
OnColEnter	Occurs when a user moves into a column on the grid.
OnColExit	Occurs when a user leaves a column on the grid.
OnColumnMoved	Occurs when the user moves a column to a new location.
OnDblClick	Occurs when a user double clicks in the grid.
OnDragDrop	Occurs when a user drags and drops in the grid.
OnDragOver	Occurs when a user drags over the grid.
OnDrawColumnCell	Occurs when application needs to draw individual cells.
OnDrawDataCell	(obsolete) Occurs when application needs to draw individual cells if *State* is *csDefault*.
OnEditButtonClick	Occurs when the user clicks on an ellipsis button in a column.
OnEndDrag	Occurs when a user stops dragging on the grid.
OnEnter	Occurs when the grid gets focus.
OnExit	Occurs when the grid loses focus.
OnKeyDown	Occurs when a user presses any key or key combination on the keyboard when the grid has focus.
OnKeyPress	Occurs when a user presses a single alphanumeric key on the keyboard when the grid has focus.
OnKeyUp	Occurs when a user releases a key when in the grid.

Table 15.6 Grid control events (continued)

Event	Purpose
OnMouseDown	Occurs when the user clicks the mouse in the grid.
OnMouseMove	Occurs when the user moves the mouse over the grid.
OnMouseUp	Occurs when the user releases the mouse button over the grid.
OnStartDrag	Occurs when a user starts dragging on the grid.
OnTitleClick	Occurs when a user clicks the title for a column.

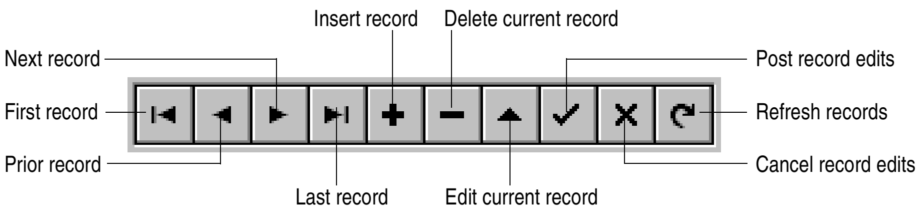
There are many uses for these events. For example, you might write a handler for the *OnDblClick* event that pops up a list from which a user can choose a value to enter in a column. Such a handler would use the *SelectedField* property to determine to current row and column.

Navigating and manipulating records

TDBNavigator provides users a simple control for navigating through records in a dataset, and for manipulating records. The navigator consists of a series of buttons that enable a user to scroll forward or backward through records one at a time, go to the first record, go to the last record, insert a new record, update an existing record, post data changes, cancel data changes, delete a record, and refresh record display.

Figure 15.2 shows the navigator that appears by default when you place it on a form at design time. The navigator consists of a series of buttons that let a user navigate from one record to another in a dataset, and edit, delete, insert, and post records. The *VisibleButtons* property of the navigator lets you hide or show a subset of these buttons dynamically.

Figure 15.2 Buttons on the TDBNavigator control



The following table describes the buttons on the navigator.

Table 15.7 TDBNavigator buttons

Button	Purpose
First	Calls the dataset's *First* method to set the current record to the first record.
Prior	Calls the dataset's *Prior* method to set the current record to the previous record.
Next	Calls the dataset's *Next* method to set the current record to the next record.
Last	Calls the dataset's *Last* method to set the current record to the last record.
Insert	Calls the dataset's *Insert* method to insert a new record before the current record, and set the dataset in Insert state.

Table 15.7 TDBNavigator buttons (continued)

Button	Purpose
Delete	Deletes the current record. If the *ConfirmDelete* property is *True* it prompts for confirmation before deleting.
Edit	Puts the dataset in Edit state so that the current record can be modified.
Post	Writes changes in the current record to the database.
Cancel	Cancels edits to the current record, and returns the dataset to Browse state.
Refresh	Clears data control display buffers, then refreshes its buffers from the physical table or query. Useful if the underlying data may have been changed by another application.

Choosing navigator buttons to display

When you first place a *TDBNavigator* on a form at design time, all its buttons are visible. You can use the *VisibleButtons* property to turn off buttons you do not want to use on a form. For example, when working with a unidirectional dataset, only the First, Next, and Refresh buttons are meaningful, and you probably want to hide the others. On a form that is intended for browsing rather than editing, you might want to disable the *Edit*, *Insert*, *Delete*, *Post*, and *Cancel* buttons.

Hiding and showing navigator buttons at design time

The *VisibleButtons* property in the Object Inspector is displayed with a + sign to indicate that it can be expanded to display a Boolean value for each button on the navigator. To view and set these values, click on the + sign. The list of buttons that can be turned on or off appears in the Object Inspector below the *VisibleButtons* property. The + sign changes to a – (minus) sign, which you can click to collapse the list of properties.

Button visibility is indicated by the *Boolean* state of the button value. If a value is set to *True*, the button appears in the *TDBNavigator*. If *False*, the button is removed from the navigator at design time and runtime.

Note As button values are set to *False*, they are removed from the *TDBNavigator* on the form, and the remaining buttons are expanded in width to fill the control. You can drag the control's handles to resize the buttons.

Hiding and showing navigator buttons at runtime

At runtime you can hide or show navigator buttons in response to user actions or application states. For example, suppose you provide a single navigator for navigating through two different datasets, one of which permits users to edit records, and the other of which is read-only. When you switch between datasets, you want to hide the navigator's *Insert, Delete, Edit, Post, Cancel*, and *Refresh* buttons for the read-only dataset, and show them for the other dataset.

For example, suppose you want to prevent edits to the *OrdersTable* by hiding the *Insert, Delete, Edit, Post, Cancel,* and *Refresh* buttons on the navigator, but that you also want to allow editing for the *CustomersTable*. The *VisibleButtons* property controls which buttons are displayed in the navigator. Here's one way you might code the *OnEnter* event handler:

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
  begin
    DBNavigatorAll.DataSource := CustomerCompany.DataSource;
    DBNavigatorAll.VisibleButtons := [nbFirst,nbPrior,nbNext,nbLast];
  end
  else
  begin
    DBNavigatorAll.DataSource := OrderNum.DataSource;
    DBNavigatorAll.VisibleButtons := DBNavigatorAll.VisibleButtons + [nbInsert,
      nbDelete,nbEdit,nbPost,nbCancel,nbRefresh];
  end;
end;
```

Displaying fly-over help

To display fly-over help for each navigator button at runtime, set the navigator *ShowHint* property to *True*. When *ShowHint* is *True*, the navigator displays fly-by Help Hints whenever you pass the mouse cursor over the navigator buttons. *ShowHint* is *False* by default.

The *Hints* property controls the fly-over help text for each button. By default *Hints* is an empty string list. When *Hints* is empty, each navigator button displays default help text. To provide customized fly-over help for the navigator buttons, use the String list editor to enter a separate line of hint text for each button in the *Hints* property. When present, the strings you provide override the default hints provided by the navigator control.

Using a single navigator for multiple datasets

As with other data-aware controls, a navigator's *DataSource* property specifies the data source that links the control to a dataset. By changing a navigator's *DataSource* property at runtime, a single navigator can provide record navigation and manipulation for multiple datasets.

Suppose a form contains two edit controls linked to the *CustomersTable* and *OrdersTable* datasets through the *CustomersSource* and *OrdersSource* data sources respectively. When a user enters the edit control connected to *CustomersSource*, the navigator should also use *CustomersSource*, and when the user enters the edit control connected to *OrdersSource*, the navigator should switch to *OrdersSource* as well. You can code an *OnEnter* event handler for one of the edit controls, and then share that event with the other edit control. For example:

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
    DBNavigatorAll.DataSource := CustomerCompany.DataSource
  else
    DBNavigatorAll.DataSource := OrderNum.DataSource;
end;
```

Chapter

16

Understanding datasets

The fundamental unit for accessing data is the dataset family of objects. Your application uses datasets for all database access. A dataset object represents a set of records from a database organized into a logical table. These records may come from a query or stored procedure that accesses a database, or from another dataset.

All dataset objects that you use in your database applications descend from the virtualized dataset object, *TDataSet*, and they inherit data fields, properties, events, and methods from *TDataSet*. This chapter describes the functionality of *TDataSet* that is inherited by the dataset objects you will use in your database applications. You need to understand this shared functionality to use any dataset object.

TDataSet is a virtualized dataset, meaning that many of its properties and methods are **virtual** or **abstract**. A *virtual method* is a function or procedure declaration where the implementation of that method can be (and usually is) overridden in descendant objects. An *abstract method* is a function or procedure declaration without an actual implementation. The declaration is a prototype that describes the method (and its parameters and return type, if any) that must be implemented in all descendant dataset objects, but that might be implemented differently by each of them.

Because *TDataSet* contains **abstract** methods, you cannot use it directly in an application without generating a runtime error. Instead, you either create instances of *TDataSet*'s descendants, such as *TClientDataSet*, *TSQLClientDataSet*, *TSQLDataSet*, *TSQLQuery*, *TSQLTable*, and *TSQLStoredProc*, or you derive your own dataset object from *TDataSet* or its descendants and write implementations for all its **abstract** methods.

TDataSet defines much that is common to all dataset objects. For example, *TDataSet* defines the basic structure of all datasets: an array of *TField* components that correspond to actual columns in one or more database tables, lookup fields provided by your application, or calculated fields provided by your application. For more information about *TField* components, see Chapter 17, “Working with field components.”

This chapter describes how to use the common database functionality introduced by *TDataSet*. Bear in mind, however, that although *TDataSet* introduces the methods for this functionality, not all *TDataSet* dependants implement them in a meaningful way.

Types of datasets

TDataSet has two immediate descendants, *TCustomClientDataset* and *TCustomSQLDataSet*. In addition you can create your own custom *TDataSet* descendants — for example to supply data from a process other than a database server.

TCustomClientDataSet is the base class for all client datasets. It is not used directly in an application because many crucial properties are protected. Client datasets buffer data and updates in memory, and can be used when you need to navigate through a dataset or edit values and apply them back to the database server. Client datasets implement most of the features introduced by *TDataSet*, as well as introducing additional features such as maintained aggregates. For information about the features introduced by client datasets, see Chapter 20, “Using client datasets.”

TCustomSQLDataSet is the base class for all unidirectional datasets. This class can’t be used directly in an application because the properties that specify what data to access are all protected. Unidirectional datasets raise exceptions when you attempt any navigation other than moving to the next record. They do not provide support for any data buffering, or the functions that require data buffering (such as updating data, filters, and lookup fields). For information about the *TCustomSQLDataSet* descendants that you can use in your applications, see Chapter 18, “Using unidirectional datasets.”

Each of these *TDataset* descendants has advantages and disadvantages. For an overview of how these types of datasets can be built into your database application, see “Database architecture” on page 14-4.

Opening and closing datasets

To read or edit the data in a dataset, an application must first open it. You can open a dataset in two ways,

- Set the *Active* property of the dataset to *True*, either at design time in the Object Inspector, or in code at runtime:

  ```
  CustTable.Active := True;
  ```

- Call the *Open* method for the dataset at runtime,

  ```
  CustQuery.Open;
  ```

You can close a dataset in two ways,

- Set the *Active* property of the dataset to *False*, either at design time in the Object Inspector, or in code at runtime,

  ```
  CustQuery.Active := False;
  ```

- Call the *Close* method for the dataset at runtime,

```
CustTable.Close;
```

You may need to close a dataset when you change certain of its properties, such as *CommandText* on a *TSQLDataSet* component. When you reopen the dataset, the new property value takes effect.

Determining and setting dataset states

The *state*—or *mode*—of a dataset determines what can be done to its data. For example, when a dataset is closed, its state is *dsInactive*, meaning that nothing can be done to its data. At runtime, you can examine a dataset's read-only *State* property to determine its current state. The following table summarizes possible values for the *State* property and what they mean:

Table 16.1 Values for the dataset State property

Value	State	Meaning
dsInactive	Inactive	DataSet closed. Its data is unavailable.
dsBrowse	Browse	DataSet open. Its data can be viewed, but not changed. This is the default state of an open dataset.
dsEdit	Edit	DataSet open. The current row can be modified. (not supported on unidirectional datasets)
dsInsert	Insert	DataSet open. A new row is inserted or appended. (not supported on unidirectional datasets)
dsSetKey	SetKey	*TClientDataSet* only. DataSet open. Enables setting of ranges and key values used for ranges and *GotoKey* operations.
dsCalcFields	CalcFields	DataSet open. Indicates that an *OnCalcFields* event is under way. Prevents changes to fields that are not calculated.
dsCurValue	CurValue	Internal use only.
dsNewValue	NewValue	Internal use only.
dsOldValue	OldValue	Internal use only.
dsFilter	Filter	DataSet open. Indicates that a filter operation is under way. A restricted set of data can be viewed, and no data can be changed. (not supported on unidirectional datasets)
dsBlockRead	Block Read	DataSet open. Data-aware controls are not updated and events are not triggered when the current record changes.
dsInternalCalc	Internal Calc	DataSet open. An *OnCalcFields* event is underway for calculated values that are stored with the record. (client datasets only)

When an application opens a dataset, it appears by default in *dsBrowse* mode. The state of a dataset changes as an application processes data. An open dataset changes from one state to another based on either the

- code in your application, or
- built-in behavior of data-related components.

To put a dataset into *dsBrowse*, *dsEdit*, *dsInsert*, *dsSetKey*, or *dsBlockRead* states, call the method or set the property that corresponds to the name of the state. For example, the following code puts *CustTable* into *dsInsert* state, accepts user input for a new record, and writes the new record to the change log:

```
CustTable.Insert; { Your application explicitly sets dataset state to Insert }
AddressPromptDialog.ShowModal;
if (AddressPromptDialog.ModalResult = mrOK) then
  CustTable.Post { Kylix sets dataset state to Browse on successful completion }
else
  CustTable.Cancel; {Kylix sets dataset state to Browse on cancel }
```

This example also illustrates that the state of a dataset automatically changes to *dsBrowse* when

- The *Post* method successfully writes a record to the change log. (If *Post* fails, the dataset state remains unchanged.)
- The *Cancel* method is called.

Some states cannot be set directly. For example, to put a dataset into *dsInactive* state, set its *Active* property to *False*, or call the *Close* method for the dataset. The following statements are equivalent:

```
CustTable.Active := False;

CustTable.Close;
```

The remaining states (*dsCalcFields*, *dsCurValue*, *dsNewValue*, *dsOldValue*, *dsFilter*, and *dsInternalCalc*) cannot be set by your application. Instead, the state of the dataset changes automatically to these values as necessary. For example, *dsCalcFields* is set when a dataset's *OnCalcFields* event occurs. When the *OnCalcFields* event finishes, the dataset is restored to its previous state.

Note Whenever a dataset's state changes, the *OnStateChange* event is called for any data source components associated with the dataset. For more information about data source components and *OnStateChange*, see "Responding to changes mediated by the data source" on page 15-4.

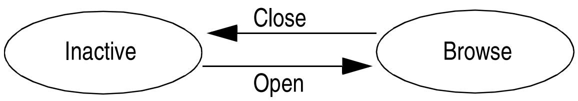
The following sections provide overviews of the most common states, how and when they are set, how states relate to one another, and where to go for related information, if applicable.

Inactivating a dataset

A dataset is inactive when it is closed. You cannot access records in a closed dataset. At design time, a dataset is closed until you set its *Active* property to *True*. At runtime, a dataset is initially closed until an application opens it by calling the *Open* method, or by setting the *Active* property to *True*.

When you open an inactive dataset, its state automatically changes to the *dsBrowse* state. The following diagram illustrates the relationship between these states and the methods that set them.

Figure 16.1 Relationship of Inactive and Browse states



To make a dataset inactive, call its *Close* method. You can write *BeforeClose* and *AfterClose* event handlers that respond to the *Close* method for a dataset. For example, if a dataset is in *dsEdit* or *dsInsert* modes when an application calls *Close*, you can prompt the user to post pending changes or cancel them before closing the dataset. The following code illustrates such a handler:

```
procedure TForm1.CustTableVerifyBeforeClose(DataSet: TDataSet);
begin
  if (CustTable.State in [dsEdit, dsInsert]) then begin
    case MessageDlg('Post changes before closing?', mtConfirmation,
      mbYesNoCancel, 0) of
      mrYes:    CustTable.Post;   { save the changes }
      mrNo:     CustTable.Cancel; { abandon the changes}
      mrCancel: Abort;            { abort closing the dataset }
    end;
  end;
end;
```

To associate a procedure with the *BeforeClose* event for a dataset at design time:

1 Select the table in the data module (or form).
2 Click the Events page in the Object Inspector.
3 Enter the name of the procedure for the *BeforeClose* event (or choose it from the drop-down list).

Browsing a dataset

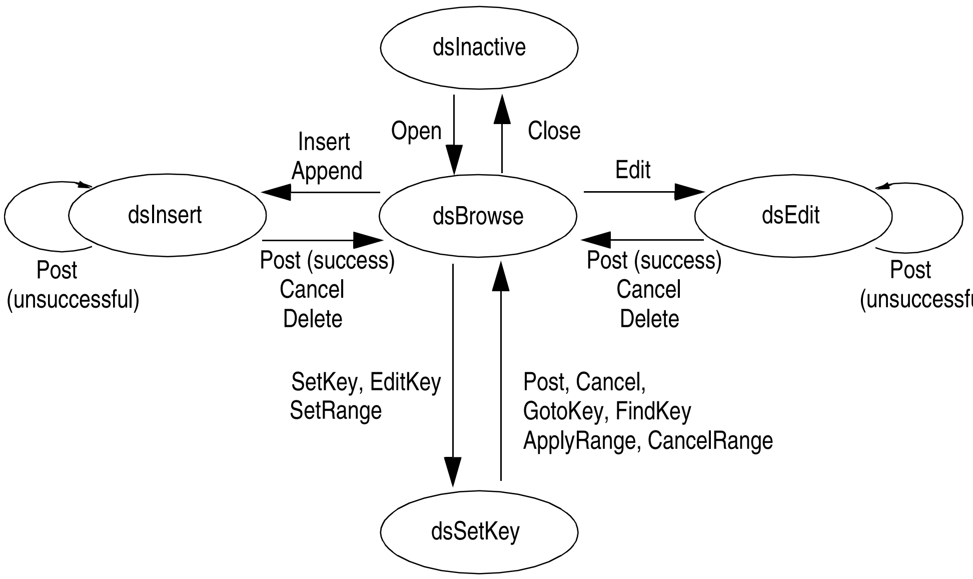
When an application opens a dataset, the dataset automatically enters *dsBrowse* state. Browsing enables you to view records in a dataset, but you cannot edit records or insert new records. You mainly use *dsBrowse* to scroll from record to record in a dataset. For more information about scrolling from record to record, see “Navigating datasets” on page 16-8.

From *dsBrowse* all other dataset states can be set, as long as the dataset supports them. For example, calling the *Insert* or *Append* methods for a dataset changes its state from *dsBrowse* to *dsInsert* (unless other factors and dataset properties, such as *CanModify* prevent this change). Calling *SetKey* to search for records puts a client dataset in *dsSetKey* mode. For more information about inserting and appending records in a dataset, see “Modifying data” on page 16-20.

Two methods associated with all datasets can return a dataset to *dsBrowse* state. *Cancel* ends the current edit, insert, or search task, and always returns a dataset to *dsBrowse* state. *Post* attempts to write changes to the database, and if successful, also returns a dataset to *dsBrowse* state. If *Post* fails, the current state remains unchanged.

The following diagram illustrates the relationship of *dsBrowse* both to the other dataset modes you can set in your applications, and the methods that set those modes.

Figure 16.2 Relationship of Browse to other dataset states



Enabling dataset editing

A dataset must be in *dsEdit* mode before an application can modify records. In your code you can use the *Edit* method to put a dataset into *dsEdit* mode if the read-only *CanModify* property for the dataset is *True*. *CanModify* is *True* unless the dataset is unidirectional, the database underlying the dataset does not permit read and write privileges, or some other factor (such as a client dataset's *ReadOnly* property or a dataset provider's *poReadOnly* option) intervenes.

On forms in your application, some data-aware controls can automatically put a dataset into *dsEdit* state if all of the following conditions apply:

- The control's *ReadOnly* property is *False* (the default)
- The *AutoEdit* property of the data source for the control is *True*
- *CanModify* is *True* for the dataset.

Note Even if a dataset is in *dsEdit* state, editing records may not succeed if your application user does not have proper SQL access privileges.

You can return a dataset from *dsEdit* state to *dsBrowse* state in code by calling the *Cancel*, *Post*, or *Delete* methods. *Cancel* discards edits to the current field or record. *Post* attempts to write a modified record to the change log, and if it succeeds, returns the dataset to *dsBrowse*. If *Post* cannot write changes, the dataset remains in *dsEdit* state. *Delete* tries to remove the current record from the dataset, and if it succeeds, returns the dataset to *dsBrowse* state. If *Delete* fails, the dataset remains in *dsEdit* state.

Data-aware controls for which editing is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid) or that causes the control to lose focus (such as moving to a different control on the form).

For a complete discussion of editing fields and records in a dataset, see “Modifying data” on page 16-20.

Enabling insertion of new records

A dataset must be in *dsInsert* mode before an application can add new records. In your code you can use the *Insert* or *Append* methods to put a dataset into *dsInsert* mode if the read-only *CanModify* property for the dataset is *True*. *CanModify* is *True* unless the dataset is unidirectional, the database underlying the dataset does not permit read and write privileges, or some other factor (such as a client dataset’s *ReadOnly* property or a dataset provider’s *poReadOnly* option) intervenes.

On forms in your application, the data-aware grid and navigator controls can put a dataset into *dsInsert* state if all of the following conditions apply:

- The control’s *ReadOnly* property is *False* (the default)
- The *AutoEdit* property of the data source for the control is *True*
- *CanModify* is *True* for the dataset.

Note Even if a dataset is in *dsInsert* state, inserting records may not succeed if your application user does not have proper SQL access privileges.

You can return a dataset from *dsInsert* state to *dsBrowse* state in code by calling the *Cancel*, *Post*, or *Delete* methods. *Delete* and *Cancel* discard the new record. *Post* attempts to write the new record to the change log, and if it succeeds, returns the dataset to *dsBrowse*. If *Post* cannot write the record, the dataset remains in *dsInsert* state.

Data-aware controls for which inserting is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid).

For more discussion of inserting and appending records in a dataset, see “Modifying data” on page 16-20.

Enabling index-based operations

There are no methods defined in *TDataSet* that put the dataset into the *dsSetKey* state. This state exists for client datasets, which uses it when setting up information for index-based operations such as indexed-based searches or limiting records to a specified range. You can put a client dataset into *dsSetKey* mode with the *SetKey* or *EditKey* method at runtime.

While the dataset is in the *dsSetKey* state, assigning values to its fields changes an internal buffer containing the criteria for the indexed-based operation, rather than the values of the fields on the current record.

The dataset remains in the *dsSetKey* state until you call a method to perform the index-based operation, or call the *Post* method.

For more information about indexed-based operations in client datasets, see "Navigating data in client datasets" on page 20-2 and "Limiting what records appear" on page 20-5.

Calculating fields

A dataset enters *dsCalcFields* or *dsInternalCalc* mode whenever its *OnCalcFields* event occurs. These states prevent modifications or additions to the records in a dataset except for the calculated fields the handler is designed to modify. The reason all other modifications are prevented is because *OnCalcFields* uses the values in other fields to derive values for calculated fields. Changes to those other fields might otherwise invalidate the values assigned to calculated fields.

When the *OnCalcFields* handler finishes, the dataset is returned to *dsBrowse* state.

For more information about creating calculated fields and *OnCalcFields* event handlers, see "Using OnCalcFields" on page 16-26.

Filtering records

If a dataset is not unidirectional, it enters *dsFilter* mode whenever an application calls the dataset's *OnFilterRecord* event handler. (Unidirectional datasets do not support filters). This state prevents modifications or additions to the records in a dataset during the filtering process so that the filter request is not invalidated.

When the *OnFilterRecord* handler finishes, the dataset is returned to *dsBrowse* state.

For more information about filtering, see "Displaying and editing a subset of data using filters" on page 16-15.

Applying updates

When applying the updates in its change log back to the database table, client datasets may enter the *dsNewValue*, *dsOldValue*, or *dsCurValue* states temporarily. These states indicate that the corresponding properties of a field component (*NewValue*, *OldValue*, and *CurValue*, respectively) are being accessed, usually in an *OnReconcileError* event handler. Your applications cannot see or set these states.

Navigating datasets

Each active dataset has a *cursor*, or pointer, to the current row in the dataset. The *current row* in a dataset is the one whose field values currently show in single-field, data-aware controls on a form, such as *TDBEdit*, *TDBLabel*, and *TDBMemo*. If the dataset supports editing, the current record contains the values that can be manipulated by edit, insert, and delete methods.

You can change the current row by moving the cursor to point at a different row. The following table lists methods you can use in application code to move to different records:

Table 16.2 Navigational methods of datasets

Method	Moves the cursor to
First	The first row in a dataset. (all datasets)
Last	The last row in a dataset. (not available for unidirectional datasets).
Next	The next row in a dataset. (all datasets)
Prior	The previous row in a dataset. (not available for unidirectional datasets).
MoveBy	A specified number of rows forward or back in a dataset. (unidirectional datasets raise an exception if you try to move backward)

The data-aware, visual component *TDBNavigator* encapsulates these methods as buttons that users can click to move among records at runtime. For more information about the navigator component, see “Navigating and manipulating records” on page 15-25.

In addition to these methods, *TDataSet* defines two Boolean properties that provide useful information when iterating through the records in a dataset:.

Table 16.3 Navigational properties of datasets

Property	Description
Bof (Beginning-of-file)	*True*: the cursor is at the first row in the dataset.
	False: the cursor is not known to be at the first row in the dataset.
Eof (End-of-file)	*True*: the cursor is at the last row in the dataset.
	False: the cursor is not known to be at the last row in the dataset.

Using the First and Last methods

The *First* method moves the cursor to the first row in a dataset and sets the *Bof* property to *True*. If the cursor is already at the first row in the dataset, *First* does nothing.

For example, the following code moves to the first record in *CustTable*:

```
CustTable.First;
```

The *Last* method moves the cursor to the last row in a dataset and sets the *Eof* property to *True*. If the cursor is already at the last row in the dataset, *Last* does nothing.

The following code moves to the last record in *CustTable*:

```
CustTable.Last;
```

Note The *Last* method raises an exception in unidirectional datasets.

Tip While there may be programmatic reasons to move to the first or last rows in a dataset without user intervention, you can also enable your users to navigate from record to record using the *TDBNavigator* component. The navigator component contains buttons that, when active and visible, enable a user to move to the first and last rows of an active dataset. The *OnClick* events for these buttons occur immediately after the navigator calls the *First* and *Last* methods of the dataset. For more information about making effective use of the navigator component, see "Navigating and manipulating records" on page 15-25.

Using the Next and Prior methods

The *Next* method moves the cursor forward one row in the dataset and sets the *Bof* property to *False* if the dataset is not empty. If the cursor is already at the last row in the dataset when you call *Next*, nothing happens.

For example, the following code moves to the next record in *CustTable*:

```
CustTable.Next;
```

The *Prior* method moves the cursor back one row in the dataset, and sets *Eof* to *False* if the dataset is not empty. If the cursor is already at the first row in the dataset when you call *Prior*, *Prior* does nothing.

For example, the following code moves to the previous record in *CustTable*:

```
CustTable.Prior;
```

Note The *Prior* method raises an exception in unidirectional datasets.

Using the MoveBy method

MoveBy lets you specify how many rows forward or back to move the cursor in a dataset. Movement is relative to the current record at the time that *MoveBy* is called. *MoveBy* also sets the *Bof* and *Eof* properties for the dataset as appropriate.

This function takes an integer parameter, the number of records to move. Positive integers indicate a forward move and negative integers indicate a backward move.

Note *MoveBy* raises an exception in unidirectional datasets if you use a negative argument.

MoveBy returns the number of rows it moves. If you attempt to move past the beginning or end of the dataset, the number of rows returned by *MoveBy* differs from the number of rows you requested to move. This is because *MoveBy* stops when it reaches the first or last record in the dataset.

The following code moves two records backward in *CustTable*:

```
CustTable.MoveBy(-2);
```

Using the Eof and Bof properties

Two read-only, runtime properties, *Eof* (End-of-file) and *Bof* (Beginning-of-file), are useful for controlling dataset navigation, particularly when you want to iterate through all records in a dataset.

Eof

When *Eof* is *True*, it indicates that the cursor is unequivocally at the last row in a dataset. *Eof* is set to *True* when an application

- Opens an empty dataset.
- Successfully calls a dataset's *Last* method.
- Calls a dataset's *Next* method, and the method fails (because the cursor is currently at the last row in the dataset.
- Calls *SetRange* on an empty range or dataset. (client datasets only)

Eof is set to *False* in all other cases; you should assume *Eof* is *False* unless one of the conditions above is met *and* you test the property directly.

Eof is commonly tested in a loop condition to control iterative processing of all records in a dataset. If you open a dataset containing records (or you call *First*), *Eof* is *False*. To iterate through the dataset a record at a time, create a loop that terminates when *Eof* is *True*. Inside the loop, call *Next* for each record in the dataset. *Eof* remains *False* until you call *Next* when the cursor is already on the last record.

The following code illustrates one way you might code a record-processing loop for a dataset called *CustTable*:

```
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets EOF False }
  while not CustTable.EOF do { Cycle until EOF is True }
  begin
    { Process each record here }
    ⋮
    CustTable.Next; { EOF False on success; EOF True when Next fails on last record }
  end;
finally
  CustTable.EnableControls;
end;
```

Tip This example also demonstrates how to disable and enable data-aware visual controls tied to a dataset. If you disable visual controls during dataset iteration, it speeds processing because your application does not need to update the contents of the controls as the current record changes. After iteration is complete, controls should be enabled again to update them with values for the new current row. Note that enabling of the visual controls takes place in the **finally** clause of a **try...finally** statement. This guarantees that even if an exception terminates loop processing prematurely, controls are not left disabled.

Bof

When *Bof* is *True*, it indicates that the cursor is unequivocally at the first row in a dataset. *Bof* is set to *True* when an application

- Opens a dataset.
- Calls a dataset's *First* method.

- Calls a dataset's *Prior* method, and the method fails because the cursor is currently at the first row in the dataset. (Note that this condition does not apply to unidirectional datasets.)
- Calls *SetRange* on an empty range or dataset. (client datasets only)

Bof is set to *False* in all other cases; you should assume *Bof* is *False* unless one of the conditions above is met *and* you test the property directly.

Like *Eof*, *Bof* can be in a loop condition to control iterative processing of records in a dataset. The following code illustrates one way you might code a record-processing loop for a dataset called *CustTable*:

```
CustTable.DisableControls; { Speed up processing; prevent screen flicker }
try
  while not CustTable.BOF do { Cycle until BOF is True }
  begin
    { Process each record here }
    ⋮
    CustTable.Prior; { BOF False on success; BOF True when Prior fails on first record }
  end;
finally
  CustTable.EnableControls; { Display new current row in controls }
end;
```

Marking and returning to records

In addition to moving from record to record in a dataset (or moving from one record to another by a specific number of records), it is often also useful to mark a particular location in a dataset so that you can return to it quickly when desired. *TDataSet* introduces a bookmarking feature that lets you tag records and return to them later. The bookmarking feature consists of a *Bookmark* property and five bookmark methods.

The *Bookmark* property indicates which bookmark among any number of bookmarks in your application is current. *Bookmark* is a string that identifies the current bookmark. Each time you add another bookmark, it becomes the current bookmark.

TDataSet implements **virtual** bookmark methods. While these methods ensure that any dataset object derived from *TDataSet* returns a value if a bookmark method is called, the return values are merely defaults that do not keep track of the current location. Unidirectional datasets do not add support for bookmarks. *TCustomClientDataSet*, however, reimplements the bookmark methods to return meaningful values as described in the following list:

- *BookmarkValid*, for determining if a specified bookmark is in use.
- *CompareBookmarks*, to test two bookmarks to see if they are the same.
- *GetBookmark*, to allocate a bookmark for your current position in the dataset.
- *GotoBookmark*, to return to a bookmark previously created by *GetBookmark.*
- *FreeBookmark*, to free a bookmark previously allocated by *GetBookmark*.

To create a bookmark, you must declare a variable of type *TBookmark* in your application. The *TBookmark* type is a pointer. When you call *GetBookmark*, the dataset allocates storage for the bookmark and sets your variable to point to that storage. The bookmark contains information that identifies a particular location in the dataset.

Before calling *GotoBookmark* to move to a specific record, you can call *BookmarkValid* to determine if the bookmark points to a record. *BookmarkValid* returns *True* if a specified bookmark points to a record. In *TDataSet*, *BookmarkValid* is a virtual method that always returns *False*, indicating that the bookmark is not valid. Descendants that support bookmarks reimplement this method to provide a meaningful return value.

You can also call *CompareBookmarks* to see if a bookmark to which you want to move is different from another (or the current) bookmark. *TDataSet.CompareBookmarks* always returns 0, indicating that the bookmarks are identical. Descendants that support bookmarks reimplement this method to provide a meaningful return value.

When passed a bookmark, *GotoBookmark* moves the cursor for the dataset to the location specified in the bookmark. *TDataSet* generates an error when you call *GotoBookmark*. Unidirectional datasets do nothing. Descendants that support bookmarks reimplement this method to provide a meaningful return value.

FreeBookmark frees the memory allocated for a specified bookmark when you no longer need it. You should also call *FreeBookmark* before reusing an existing bookmark.

The following code illustrates one use of bookmarking:

```
procedure DoSomething (const MyDS: TSQLClientDataSet)
var
  Bookmark: TBookmark;
begin
  Bookmark := MyDS.GetBookmark; { Allocate memory and assign a value }
  MyDS.DisableControls; { Turn off display of records in data controls }
  try
    MyDS.First; { Go to first record in table }
    while not MyDS.Eof do {Iterate through each record in table }
    begin
      { Do your processing here }
      ⋮
      MyDs.Next;
    end;
  finally
    MyDs.GotoBookmark(Bookmark);
    MyDs.EnableControls; { Turn on display of records in data controls, if necessary }
    MyDs.FreeBookmark(Bookmark); {Deallocate memory for the bookmark }
  end;
end;
```

Before iterating through records, controls are disabled. Should an error occur during iteration through records, the **finally** clause ensures that controls are always enabled and that the bookmark is always freed even if the loop terminates prematurely.

Searching datasets

If a dataset is not unidirectional, you can search against it using the *Locate* and *Lookup* methods of *TDataSet*. These methods enable you to search on any type of columns in any dataset.

Note Client datasets introduce an additional family of methods that let you search for records based on an index. For information about searching a client dataset based on its index, see "Navigating data in client datasets" on page 20-2.

Using Locate

Locate moves the cursor to the first row matching a specified set of search criteria. In its simplest form, you pass *Locate* the name of a column to search, a field value to match, and an options flag specifying whether the search is case-insensitive or if it can use partial-key matching. (Partial-key matching is when the criterion string need only be a prefix of the field value.) For example, the following code moves the cursor to the first row in the *CustTable* where the value in the *Company* column is "Professional Divers, Ltd.":

```
var
  LocateSuccess: Boolean;
  SearchOptions: TLocateOptions;
begin
  SearchOptions := [loPartialKey];
  LocateSuccess := CustTable.Locate('Company', 'Professional Divers, Ltd.',
    SearchOptions);
end;
```

If *Locate* finds a match, the first record containing the match becomes the current record. *Locate* returns *True* if it finds a matching record, *False* if it does not. If a search fails, the current record does not change.

The real power of *Locate* comes into play when you want to search on multiple columns and specify multiple values to search for. Search values are Variants, which means you can specify different data types in your search criteria. To specify multiple columns in a search string, separate individual items in the string with semicolons.

Because search values are Variants, if you pass multiple values, you must either pass a Variant array type as an argument (for example, the return values from the *Lookup* method), or you must construct the Variant array on the fly using the *VarArrayOf* function. The following code illustrates a search on multiple columns using multiple search values and partial-key matching:

```
with CustTable do
  Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver','P']), loPartialKey);
```

Locate uses the fastest possible method to locate matching records. If the columns to search are indexed and the index is compatible with the search options you specify, *Locate* uses the index.

Using Lookup

Lookup searches for the first row that matches specified search criteria. If it finds a matching row, it forces the recalculation of any calculated fields and lookup fields associated with the dataset, then returns one or more fields from the matching row. *Lookup* does not move the cursor to the matching row; it only returns values from it.

In its simplest form, you pass *Lookup* the name of field to search, the field value to match, and the field or fields to return. For example, the following code looks for the first record in the *CustTable* where the value of the *Company* field is "Professional Divers, Ltd.", and returns the company name, a contact person, and a phone number for the company:

```
var
  LookupResults: Variant;
begin
with CustTable do
  LookupResults := Lookup('Company', 'Professional Divers, Ltd.', 'Company;
    Contact; Phone');
end;
```

Lookup returns values for the specified fields from the first matching record it finds. Values are returned as Variants. If more than one return value is requested, *Lookup* returns a Variant array. If there are no matching records, *Lookup* returns a Null Variant. For more information about Variant arrays, see the online help.

The real power of *Lookup* comes into play when you want to search on multiple columns and specify multiple values to search for. To specify strings containing multiple columns or result fields, separate individual fields in the string items with semicolons.

Because search values are Variants, if you pass multiple values, you must either pass a Variant array type as an argument (for example, the return values from the *Lookup* method), or you must construct the Variant array on the fly using the *VarArrayOf* function. The following code illustrates a lookup search on multiple columns:

```
var
  LookupResults: Variant;
begin
with CustTable do
  LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver', 'Christiansted']),
    'Company; Addr1; Addr2; State; Zip');
end;
```

Lookup uses the fastest possible method to locate matching records. If the columns to search are indexed, *Lookup* uses the index.

Displaying and editing a subset of data using filters

An application is frequently interested in only a subset of records within a dataset. For example, you may be interested in retrieving or viewing only those records for companies based in California in your customer database, or you may want to find a record that contains a particular set of field values.

With unidirectional datasets, you can only limit the records in the dataset when you specify the SQL command that fetches the records. With other *TDataSet* descendants, however, you can define a subset of the data that has already been fetched. To restrict an application's access to a subset of all records in the dataset, you can use filters.

A filter specifies conditions a record must meet to be displayed. Filter conditions can be stipulated in a dataset's *Filter* property or coded into its *OnFilterRecord* event handler. Filter conditions are based on the values in any specified number of fields in a dataset whether or not those fields are indexed. For example, to view only those records for companies based in California, a simple filter might require that records contain a value in the State field of "CA".

Note Filters are applied to every record retrieved in a dataset. When you want to filter large volumes of data, it may be more efficient to restrict record retrieval using the SQL command that fetches the data, or to set a range on an indexed client dataset rather than using filters.

Enabling and disabling filtering

Enabling filters on a dataset is a three-step process:

1. Create a filter.
2. Set filter options for string-based filter tests, if necessary.
3. Set the *Filtered* property to *True*.

When filtering is enabled, only those records that meet the filter criteria are available to an application. Filtering is always a temporary condition. You can turn off filtering by setting the *Filtered* property to *False*.

Creating filters

There are two ways to create a filter for a dataset:

- Specify simple filter conditions in the *Filter* property. *Filter* is especially useful for creating and applying filters at runtime.
- Write an *OnFilterRecord* event handler for simple or complex filter conditions. With *OnFilterRecord*, you specify filter conditions at design time. Unlike the *Filter* property, which is restricted to a single string containing filter logic, an *OnFilterRecord* event can take advantage of branching and looping logic to create complex, multi-level filter conditions.

The main advantage to creating filters using the *Filter* property is that your application can create, change, and apply filters dynamically, (for example, in response to user input). Its main disadvantages are that filter conditions must be expressible in a single text string, cannot make use of branching and looping constructs, and cannot test or compare its values against values not already in the dataset.

The strengths of the *OnFilterRecord* event are that a filter can be complex and variable, can be based on multiple lines of code that use branching and looping constructs, and can test dataset values against values outside the dataset, such as the text in an edit box. The main weakness of using *OnFilterRecord* is that you set the filter at design time and it cannot be modified in response to user input. (You can, however, create several filter handlers and switch among them in response to general application conditions.)

The following sections describe how to create filters using the *Filter* property and the *OnFilterRecord* event handler.

Setting the Filter property

To create a filter using the *Filter* property, set the value of the property to a string that contains the filter conditions. The string contains the filter's test condition. For example, the following statement creates a filter that tests a dataset's *State* field to see if it contains a value for the state of California:

```
Dataset1.Filter := 'State = ' + QuotedStr('CA');
```

You can also supply a value for *Filter* based on the text entered in a control. For example, the following statement assigns the text in an edit box to *Filter*:

```
Dataset1.Filter := Edit1.Text;
```

You can, of course, create a string based on both hard-coded text and data entered by a user in a control:

```
Dataset1.Filter := 'State = ' + QuotedStr(Edit1.Text);
```

Blank records do not appear unless they are explicitly included in the filter:

```
Dataset1.Filter := 'State <> ''CA'' or State = BLANK';
```

After you specify a value for *Filter*, to apply the filter to the dataset, set the *Filtered* property to *True*.

You can also compare field values to literals and to constants or calculate values using the following operators:

Table 16.4 Operators that can appear in a filter

Operator	Meaning
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal to
AND	Tests two statements are both *True*
NOT	Tests that the following statement is not *True*
OR	Tests that at least one of two statements is *True*
IS NULL	Tests that a field value is null.
IS NOT NULL	Tests that a field value is not null.

Table 16.4 Operators that can appear in a filter (continued)

Operator	Meaning
+	Adds numbers, concatenates strings, adds numbers to date/time values
-	Subtracts numbers, subtracts dates, or subtracts a number from a date
*	Multiplies two numbers
/	Divides two numbers
Upper	Upper-cases a string
Lower	Lower-cases a string
Substring	Returns the substring starting at a specified position.
Trim	Trims spaces or a specified character from front and back of a string.
TrimLeft	Trims spaces or a specified character from front of a string.
TrimRight	Trims spaces or a specified character from back of a string.
Year	Returns the year from a date/time value
Month	Returns the month from a date/time value
Day	Returns the day from a date/time value
Hour	Returns the hour from a time value
Minute	Returns the minute from a time value
Second	Returns the seconds from a time value
GetDate	Returns the current date
Date	Returns the date part of a date/time value
Time	Returns the time part of a date/time value
Like	Provides pattern matching in string comparisons.
In	Tests for set inclusion.
*	Wildcard for partial comparisons.

By using combinations of these operators, you can create fairly sophisticated filters. For example, the following statement checks to make sure that two test conditions are met before accepting a record for display:

```
(Custno > 1400) AND (Custno < 1500);
```

Note When filtering is on, user edits to a record may mean that the record no longer meets a filter's test conditions. The next time the record is retrieved from the dataset, it may therefore "disappear." If that happens, the next record that passes the filter condition becomes the current record.

Writing an OnFilterRecord event handler

A filter for a dataset is an event handler that responds to *OnFilterRecord* events generated by the dataset for each record it retrieves. At the heart of every filter handler is a test that determines if a record should be included in those that are visible to the application.

To indicate whether a record passes the filter condition, your filter handler must set an *Accept* parameter to *True* to include a record, or *False* to exclude it. For example, the following filter displays only those records with the State field set to "CA":

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
  Accept := DataSet['State'].AsString = 'CA';
end;
```

When filtering is enabled, an *OnFilterRecord* event is generated for each record retrieved. The event handler tests each record, and only those that meet the filter's conditions are displayed. Because the *OnFilterRecord* event is generated for every record in a dataset, you should keep the event handler as tightly-coded as possible to avoid adversely affecting the performance of your application.

Switching filter event handlers at runtime

You can code any number of filter event handlers and switch among them at runtime. To switch to a different filter event handler at runtime, assign the new event handler to the dataset's *OnFilterRecord* property.

For example, the following statements switch to an *OnFilterRecord* event handler called *NewYorkFilter*:

```
DataSet1.OnFilterRecord := NewYorkFilter;
Refresh;
```

Setting filter options

The *FilterOptions* property lets you specify whether a filter that compares string-based fields accepts records based on partial comparisons and whether string comparisons are case-sensitive. *FilterOptions* is a set property that can be an empty set (the default), or that can contain either or both of the following values:

Table 16.5 FilterOptions values

Value	Meaning
foCaseInsensitive	Ignore case when comparing strings.
foNoPartialCompare	Disable partial string matching (i.e., do not match strings ending with an asterisk (*)).

For example, the following statements set up a filter that ignores case when comparing values in the *State* field:

```
FilterOptions := [foCaseInsensitive];
Filter := 'State = ' + QuotedStr('CA');
```

Navigating records in a filtered dataset

There are four dataset methods that enable you to navigate among records in a filtered dataset. The following table lists these methods and describes what they do:

Table 16.6 Filtered dataset navigational methods

Method	Purpose
FindFirst	Move to the first record in the dataset that matches the current filter criteria. The search for the first matching record always begins at the first record in the unfiltered dataset.
FindLast	Move to the last record in the dataset that matches the current filter criteria.
FindNext	Moves from the current record in the filtered dataset to the next one.
FindPrior	Move from the current record in the filtered dataset to the previous one.

For example, the following statement finds the first filtered record in a dataset:

```
DataSet1.FindFirst;
```

Provided that you set the *Filter* property or create an *OnFilterRecord* event handler for your application, these methods position the cursor on the specified record regardless of whether filtering is currently enabled. If you call these methods when filtering is not enabled, then they

- Temporarily enable filtering.
- Position the cursor on a matching record if one is found.
- Disable filtering.

Note If filtering is disabled and you do not set the *Filter* property or create an *OnFilterRecord* event handler, these methods do the same thing as *First*, *Last*, *Next*, and *Prior*.

All navigational filter methods position the cursor on a matching record (if one is found), make that record the current one, and return *True*. If a matching record is not found, the cursor position is unchanged, and these methods return *False*. You can check the status of the *Found* property to wrap these calls, and only take action when *Found* is *True*. For example, if the cursor is already on the last matching record in the dataset, and you call *FindNext*, the method returns *False*, and the current record is unchanged.

Modifying data

You can use the dataset methods listed below to insert, update, and delete data if the read-only *CanModify* property for the dataset is *True*. *CanModify* is *True* unless the dataset is unidirectional, the database underlying the dataset does not permit read and write privileges, or some other factor (such as a client dataset's *ReadOnly* property or a dataset provider's *poReadOnly* option) intervenes:

Table 16.7 Dataset methods for inserting, updating, and deleting data

Method	Description
Edit	Puts the dataset into *dsEdit* state if it is not already in *dsEdit* or *dsInsert* states.
Append	Posts any pending data, moves current record to the end of the dataset, and puts the dataset in *dsInsert* state.
Insert	Posts any pending data, and puts the dataset in *dsInsert* state.
Post	Attempts to post the new or altered record to the database. If successful, the dataset is put in *dsBrowse* state; if unsuccessful, the dataset remains in its current state.
Cancel	Cancels the current operation and puts the dataset in *dsBrowse* state.
Delete	Deletes the current record and puts the dataset in *dsBrowse* state.

Editing records

A dataset must be in *dsEdit* mode before an application can modify records. In your code you can use the *Edit* method to put a dataset into *dsEdit* mode if the read-only *CanModify* property for the dataset is *True*.

On forms in your application, some data-aware controls can automatically put a dataset into *dsEdit* state if

- The control's *ReadOnly* property is *False* (the default),
- The *AutoEdit* property of the data source for the control is *True*, and
- *CanModify* is *True* for the dataset.

Note Even if a dataset is in *dsEdit* state, editing records may not succeed if your application's user does not have proper SQL access privileges.

Once a dataset is in *dsEdit* mode, a user can modify the field values for the current record that appears in any data-aware controls on a form. Data-aware controls for which editing is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid).

If you provide a navigator component on your forms, users can cancel edits by clicking the navigator's Cancel button. Canceling edits returns a dataset to *dsBrowse* state.

In code, you must write or cancel edits by calling the appropriate methods. You write changes by calling *Post*. You cancel them by calling *Cancel*. In code, *Edit* and *Post* are often used together. For example,

```
with CustTable do
begin
  Edit;
  FieldValues['CustNo'] := 1234;
  Post;
end;
```

In the previous example, the first line of code places the dataset in *dsEdit* mode. The next line of code assigns the number 1234 to the *CustNo* field of the current record. Finally, the last line writes, or posts, the modified record to the change log.

Note When you want to write the edits in the change log back to the database, you must call the *ApplyUpdates* method of a client dataset. For more information about editing in client datasets, see "Editing data" on page 20-14.

Adding new records

A dataset must be in *dsInsert* mode before an application can add new records. In code, you can use the *Insert* or *Append* methods to put a dataset into *dsInsert* mode if the read-only *CanModify* property for the dataset is *True*.

On forms in your application, the data-aware grid and navigator controls can put a dataset into *dsInsert* state if

- The control's *ReadOnly* property is *False* (the default), and
- *CanModify* is *True* for the dataset.

Once a dataset is in *dsInsert* mode, a user or application can enter values into the fields associated with the new record. Except for the grid and navigational controls, there is no visible difference to a user between *Insert* and *Append*. On a call to *Insert*, an empty row appears in a grid above what was the current record. On a call to *Append*, the grid is scrolled to the last record in the dataset, an empty row appears at the bottom of the grid, and the *Next* and *Last* buttons are dimmed on any navigator component associated with the dataset.

Data-aware controls for which inserting is enabled automatically call *Post* when a user executes any action that changes which record is current (such as moving to a different record in a grid). Otherwise you must call *Post* in your code.

Post writes the new record to the change log. To write inserts and appends from the change log to the database, call the *ApplyUpdates* method of a client dataset.

Inserting records

Insert opens a new, empty record before the current record, and makes the empty record the current record so that field values for the record can be entered either by a user or by your application code.

When an application calls *Post*, a newly inserted record is written to the change log as follows:

- For indexed datasets, the record is inserted into the dataset in a position based on its index.
- For unindexed tables, the record is inserted into the dataset at its current position.

Appending records

Append opens a new, empty record at the end of the dataset, and makes the empty record the current one so that field values for the record can be entered either by a user or by your application code.

When an application calls *Post*, a newly appended record is written to the change log as follows:

- For indexed datasets, the record is inserted into the dataset in a position based on its index.
- For unindexed tables, the record is added to the end of the dataset.

Note When the dataset applies the updates in the change log, the physical location of inserted and appended records in the underlying SQL database is implementation-specific. If the table is indexed, the index is updated with the new record information.

Deleting records

A dataset must be active before an application can delete records. *Delete* removes the current record from the dataset into the change log and puts the dataset in *dsBrowse* mode. The record that followed the deleted record becomes the current record. A deleted record is not removed from the change log until you call *ApplyUpdates*.

If you provide a navigator component on your forms, users can delete the current record by clicking the navigator's Delete button. In code, you must call *Delete* explicitly to remove the current record.

Posting data to the database

The *Post* method is central to a database application's ability to edit. *Post* writes changes to the current record to the change log, but it behaves differently depending on a dataset's state.

- In *dsEdit* state, *Post* writes a modified record to the change log.
- In *dsInsert* state, *Post* writes a new record to the change log.
- In *dsSetKey* state, *Post* returns the dataset to *dsBrowse* state.

Posting can be done explicitly, or implicitly as part of another procedure. When an application moves off the current record, *Post* is called implicitly. Calls to the *First*, *Next*, *Prior*, and *Last* methods perform a *Post* if the table is in *dsEdit* or *dsInsert* modes. The *Append* and *Insert* methods also implicitly post any pending data.

Warning The *Close* method does not call *Post* implicitly. Use the *BeforeClose* event to post any pending edits explicitly.

Canceling changes

An application can undo changes made to the current record at any time, if it has not yet directly or indirectly called *Post*. For example, if a dataset is in *dsEdit* mode, and a user has changed the data in one or more fields, the application can return the record back to its original values by calling the *Cancel* method for the dataset. A call to *Cancel* always returns a dataset to *dsBrowse* state.

On forms, you can allow users to cancel edit, insert, or append operations by including the Cancel button on a navigator component associated with the dataset, or you can provide code for your own Cancel button on the form.

Client datasets introduce additional methods to cancel edits on non-current records. See "Undoing changes" on page 20-15 for details.

Modifying entire records

On forms, all data-aware controls except for grids and the navigator provide access to a single field in a record.

In code, however, you can use the following methods that work with entire record structures provided that the structure of the database tables underlying the dataset is stable and does not change. The following table summarizes the methods available for working with entire records rather than individual fields in those records:

Table 16.8 Methods that work with entire records

Method	Description
AppendRecord([array of values])	Appends a record with the specified column values at the end of a table; analogous to *Append*. Performs an implicit *Post*.
InsertRecord([array of values])	Inserts the specified values as a record before the current cursor position of a table; analogous to *Insert*. Performs an implicit *Post*.
SetFields([array of values])	Sets the values of the corresponding fields; analogous to assigning values to *TField*s. Application must perform an explicit *Post*.

These methods each take an array of values as an argument, where each value corresponds to a column in the underlying dataset. The values can be literals, variables, or NULL. If the number of values in an argument is less than the number of columns in a dataset, then the remaining values are assumed to be NULL.

For unindexed datasets, *AppendRecord* adds a record to the end of the dataset and *InsertRecord* inserts a record after the current cursor position. For indexed tables, both methods place the record in the correct position in the table, based on the index. In both cases, the methods move the cursor to the record's position.

SetFields assigns the values specified in the array of parameters to fields in the dataset. To use *SetFields*, an application must first call *Edit* to put the dataset in *dsEdit* mode. To write out the changes to the current record, it must perform a *Post*.

If you use *SetFields* to modify some, but not all fields in an existing record, you can pass NULL values for fields you do not want to change. If you do not supply enough values for all fields in a record, SetFields assigns NULL values to them. NULL values overwrite any existing values already in those fields.

For example, consider a client dataset called CountryTable with columns for Name, Capital, Continent, Area, and Population. The following statement would insert a record into the client dataset:

```
CountryTable.InsertRecord(['Japan', 'Tokyo', 'Asia']);
```

This statement does not specify values for Area and Population, so NULL values are inserted for them. The dataset is indexed on Name, so the statement would insert the record based on the alphabetic collation of "Japan".

To update the record, an application could use the following code:

```
with CountryTable do
begin
  if Locate('Name', 'Japan', loCaseInsensitive) then;
  begin
    Edit;
    SetFields(nil, nil, nil, 344567, 164700000);
    Post;
  end;
end;
```

This code assigns values to the Area and Population fields and then posts them to the change log. The three nil arguments act as place holders for the first three columns to preserve their current contents.

Using dataset events

TDataSet defines a number of events that enable an application to perform validation, compute totals, and perform other tasks. The events are listed in the following table.

Table 16.9 Dataset events

Event	Description
BeforeOpen, AfterOpen	Called before/after a dataset is opened.
BeforeClose, AfterClose	Called before/after a dataset is closed.
BeforeInsert, AfterInsert	Called before/after a dataset enters Insert state.
BeforeEdit, AfterEdit	Called before/after a dataset enters Edit state.
BeforePost, AfterPost	Called before/after changes to a table are posted.
BeforeCancel, AfterCancel	Called before/after the previous state is canceled.
BeforeDelete, AfterDelete	Called before/after a record is deleted.
OnNewRecord	Called when a new record is created; used to set default values.
OnCalcFields	Called when calculated fields are calculated.

Aborting a method

To abort a method such as an *Open* or *Insert*, call the *Abort* procedure in any of the *Before* event handlers (*BeforeOpen*, *BeforeInsert*, and so on). For example, the following code requests a user to confirm a delete operation or else it aborts the call to *Delete*:

```
procedure TForm1.TableBeforeDelete (Dataset: TDataset)begin
  if MessageDlg('Delete This Record?', mtConfirmation, mbYesNoCancel, 0) <> mrYes then
    Abort;
end;
```

Using OnCalcFields

The *OnCalcFields* event is used to set the values of calculated fields. The *AutoCalcFields* property determines when *OnCalcFields* is called. If *AutoCalcFields* is *True*, *OnCalcFields* is called when

- A dataset is opened.
- Focus moves from one visual component to another, or from one column to another in a data-aware grid control and the current record has been modified.
- A record is retrieved from the database.

OnCalcFields is always called whenever a value in a non-calculated field changes, regardless of the setting of *AutoCalcFields*.

Caution *OnCalcFields* is called frequently, so the code you write for it should be kept short. Also, if *AutoCalcFields* is *True*, *OnCalcFields* should not perform any actions that modify the dataset (or the linked dataset if it is part of a master-detail relationship), because this can lead to recursion. For example, if *OnCalcFields* performs a *Post*, and *AutoCalcFields* is *True*, then *OnCalcFields* is called again, leading to another *Post*, and so on.

If *AutoCalcFields* is *False*, then *OnCalcFields* is not called when individual fields within a single record are modified.

When *OnCalcFields* executes, a dataset is in *dsCalcFields* mode, so you cannot set the values of any fields other than calculated fields. After *OnCalcFields* is completed, the dataset returns to *dsBrowse* state.

Chapter

17

Working with field components

This chapter describes the properties, events, and methods common to the *TField* object and its descendants. Field components represent individual fields (columns) in datasets. This chapter also describes how to use field components to control the display and editing of data in your applications.

Field components are always associated with a dataset. You never use a *TField* object directly in your applications. Instead, each field component in your application is a *TField* descendant specific to the datatype of a column in a dataset. Field components provide data-aware controls such as *TDBEdit* and *TDBGrid* access to the data in a particular column of the associated dataset.

Generally speaking, a single field component represents the characteristics of a single column, or field, in a dataset, such as its data type and size. It also represents the field's display characteristics, such as alignment, display format, and edit format. For example, a *TFloatField* component has four properties that directly affect the appearance of its data:

Table 17.1 TFloatField properties that affect data display

Property	Purpose
Alignment	Specifies whether data is displayed left-aligned, centered, or right-aligned.
DisplayWidth	Specifies the number of digits to display in a control at one time.
DisplayFormat	Specifies data formatting for display (such as how many decimal places to show).
EditFormat	Specifies how to display a value during editing.

As you scroll from record to record in a dataset, a field component lets you view and change the value for that field in the current record.

Field components have many properties in common with one another (such as *DisplayWidth* and *Alignment*), and they have properties specific to their data types (such as *Precision* for *TFloatField*). Each of these properties affect how data appears to an application's users on a form. Some properties, such as *Precision*, can also affect what data values the user can enter in a control when modifying or entering data.

All field components for a dataset are either *dynamic* (automatically generated for you based on the underlying structure of database tables), or *persistent* (generated based on specific field names and properties you set in the Fields editor). Dynamic and persistent fields have different strengths and are appropriate for different types of applications. The following sections describe dynamic and persistent fields in more detail and offer advice on choosing between them.

Dynamic field components

Dynamically generated field components are the default. In fact, all field components for any dataset start out as dynamic fields the first time you place a dataset on a data module, specify how that dataset fetches its data, and open it. A field component is *dynamic* if it is created automatically based on the underlying physical characteristics of the data represented by a dataset. Datasets generate one field component for each column in the underlying data. The exact *TField* descendant created for each column is determined by field type information received from the database or (for client datasets) from a provider component.

Dynamic fields are temporary. They exist only as long as a dataset is open. Each time you reopen a dataset that uses dynamic fields, it rebuilds a completely new set of dynamic field components based on the current structure of the data underlying the dataset. If the columns in the underlying data change, then the next time you open a dataset that uses dynamic field components, the automatically generated field components are also changed to match.

Use dynamic fields in applications that must be flexible about data display and editing. For example, to create a database browsing tool, you must use dynamic fields because every database table has different numbers and types of columns. You might also want to use dynamic fields in applications where user interaction with data mostly takes place inside grid components and you know that the datasets used by the application change frequently.

To use dynamic fields in an application:

1. Place datasets and data sources in a data module.
2. Associate the datasets with data. This involves using a connection component or provider to connect to the source of the data and setting any properties that specify what data the dataset represents.
3. Associate the data sources with the datasets.
4. Place data-aware controls in the application's forms, add the data module to each uses clause for each form's unit, and associate each data-aware control with a data source in the module. In addition, associate a field with each data-aware control that requires one. Note that because you are using dynamic field components, there is no guarantee that any fieldname you specify will exist when the dataset is opened.
5. Open the datasets.

Aside from ease of use, dynamic fields can be limiting. Without writing code, you cannot change the display and editing defaults for dynamic fields, you cannot safely change the order in which dynamic fields are displayed, and you cannot prevent access to any fields in the dataset. You cannot create additional fields for the dataset, such as calculated fields or lookup fields, and you cannot override a dynamic field's default data type. To gain control and flexibility over fields in your database applications, you need to invoke the Fields editor to create persistent field components for your datasets.

Persistent field components

By default, dataset fields are dynamic. Their properties and availability are automatically set and cannot be changed in any way. To gain control over a field's properties and events you must create persistent fields for the dataset. Persistent fields let you

- Set or change the field's display or edit characteristics at design time or runtime.
- Create new fields, such as lookup fields, calculated fields, and aggregated fields, that base their values on existing fields in a dataset.
- Validate data entry.
- Remove field components from the list of persistent components to prevent your application from accessing particular columns in an underlying database.
- Define new fields to replace existing fields, based on columns in the table or query underlying a dataset.

At design time, you can—and should—use the Fields editor to create persistent lists of the field components used by the datasets in your application. Persistent field component lists are stored in your application, and do not change even if the structure of a database underlying a dataset is changed. Once you create persistent fields with the Fields editor, you can also create event handlers for them that respond to changes in data values and that validate data entries.

Note When you create persistent fields for a dataset, only those fields you select are available to your application at design time and runtime. At design time, you can always use the Fields editor to add or remove persistent fields for a dataset.

All fields used by a single dataset are either persistent or dynamic. You cannot mix field types in a single dataset. If you create persistent fields for a dataset, and then want to revert to dynamic fields, you must remove all persistent fields from the dataset. For more information about dynamic fields, see "Dynamic field components" on page 17-2.

Note One of the primary uses of persistent fields is to gain control over the appearance and display of data. You can also control the appearance of columns in data-aware grids. To learn about controlling column appearance in grids, see "Creating a customized grid" on page 15-15.

Creating persistent fields

Persistent field components created with the Fields editor provide efficient, readable, and type-safe programmatic access to underlying data. Using persistent field components guarantees that each time your application runs, it always uses and displays the same columns, in the same order even if the physical structure of the underlying data changes. Data-aware components and program code that rely on specific fields always work as expected. If a column on which a persistent field component is based is deleted or changed, your application raises an exception rather than running the application against a nonexistent column or mismatched data.

To create persistent fields for a dataset:

1. Place a dataset in a data module.
2. Bind the dataset to its underlying data. This typically involves associating the dataset with a connection component or provider and specifying any properties to describe the data. For example, If you are using *TSQLDataSet*, you can set the *SQLConnection* property to a properly configured *TSQLConnection* component and set the *CommandText* property to a valid query.
3. Double-click the dataset component in the data module to invoke the Fields editor. The Fields editor contains a title bar, navigator buttons, and a list box.

   The title bar of the Fields editor displays both the name of the data module or form containing the dataset, and the name of the dataset itself. For example, if you open the *Customers* dataset in the *CustomerData* data module, the title bar displays 'CustomerData.Customers,' or as much of the name as fits.

   Below the title bar is a set of navigation buttons that let you scroll one-by-one through the records in an active dataset at design time, and to jump to the first or last record. The navigation buttons are dimmed if the dataset is not active or if the dataset is empty. If the dataset is unidirectional, the buttons for moving to the last record and the previous record are always dimmed.

   The list box displays the names of persistent field components for the dataset. The first time you invoke the Fields editor for a new dataset, the list is empty because the field components for the dataset are dynamic, not persistent. If you invoke the Fields editor for a dataset that already has persistent field components, you see the field component names in the list box.
4. Choose Add Fields from the Fields editor context menu.
5. Select the fields to make persistent in the Add Fields dialog box. By default, all fields are selected when the dialog box opens. Any fields you select become persistent fields.

The Add Fields dialog box closes, and the fields you selected appear in the Fields editor list box. Fields in the Fields editor list box are persistent. If the dataset is active, note, too, that the Next and (if the dataset is not unidirectional) Last navigation buttons above the list box are enabled.

From now on, each time you open the dataset, it no longer creates dynamic field components for every column in the underlying database. Instead it only creates persistent components for the fields you specified.

Each time you open the dataset, it verifies that each non-calculated persistent field exists or can be created from data in the database. If it cannot, the dataset raises an exception warning you that the field is not valid, and does not open.

Arranging persistent fields

The order in which persistent field components are listed in the Fields editor list box is the default order in which the fields appear in a data-aware grid component. You can change field order by dragging and dropping fields in the list box.

To change the order of fields:

1 Select the fields. You can select and order one or more fields at a time.

2 Drag the fields to a new location.

If you select a noncontiguous set of fields and drag them to a new location, they are inserted as a contiguous block. Within the block, the order of fields does not change.

Alternatively, you can select the field, and use *Ctrl+Up* and *Ctrl+Dn* to change an individual field's order in the list.

Defining new persistent fields

Besides making existing dataset fields into persistent fields, you can also create special persistent fields as additions to or replacements for the other persistent fields in a dataset.

New persistent fields that you create are only for display purposes. The data they contain at runtime are not retained either because they already exist elsewhere in the database, or because they are temporary. The physical structure of the data underlying the dataset is not changed in any way.

To create a new persistent field component, invoke the context menu for the Fields editor and choose New field. The New Field dialog box appears.

The New Field dialog box contains three group boxes: Field properties, Field type, and Lookup definition.

- The Field properties group box lets you enter general field component information. Enter the field name in the Name edit box. The name you enter here corresponds to the field component's *FieldName* property. The New Field dialog uses this name to build a component name in the Component edit box. The name that appears in the Component edit box corresponds to the field component's *Name* property and is only provided for informational purposes (*Name* is the identifier by which you refer to the field component in your source code). The dialog discards anything you enter directly in the Component edit box.

- The Type combo box in the Field properties group lets you specify the field component's data type. You must supply a data type for any new field component you create. For example, to display floating-point currency values in a field, select *Currency* from the drop-down list. Use the Size edit box to specify the maximum number of characters that can be displayed or entered in a string-based field, or the size of *Bytes* and *VarBytes* fields. For all other data types, Size is meaningless.
- The Field type radio group lets you specify the type of new field component to create. The default type is Data. If you choose Lookup, the Dataset and Source Fields edit boxes in the Lookup definition group box are enabled. You can also create Calculated fields, and if you are working with a client dataset, you can create InternalCalc fields or Aggregate fields. The following table describes these types of fields you can create:

Table 17.2 Special persistent field kinds

Field kind	Purpose
Data	Replaces an existing field (for example to change its data type)
Calculated	Displays values calculated at runtime by a dataset's *OnCalcFields* event handler.
Lookup	Retrieve values from a specified dataset at runtime based on search criteria you specify. (not supported by unidirectional datasets)
InternalCalc	Displays values calculated at runtime by a client dataset and stored with its data.
Aggregate	Displays a value summarizing the data in a set of records from a client dataset.

The Lookup definition group box is only used to create lookup fields. This is described more fully in "Defining a lookup field" on page 17-8.

Defining a data field

A data field replaces an existing field in a dataset. For example, for programmatic reasons you might want to replace a *TSmallIntField* with a *TIntegerField*. Because you cannot change a field's data type directly, you must define a new field to replace it.

Important Even though you define a new field to replace an existing field, the field you define must derive its data values from an existing column in a table underlying a dataset.

To create a replacement data field for a field in a table underlying a dataset, follow these steps:

1. Remove the field from the list of persistent fields assigned for the dataset, and then choose New Field from the context menu.
2. In the New Field dialog box, enter the name of an existing field in the database table in the Name edit box. Do not enter a new field name. You are actually specifying the name of the field from which your new field will derive its data.
3. Choose a new data type for the field from the Type combo box. The data type you choose should be different from the data type of the field you are replacing. You cannot replace a string field of one size with a string field of another size. Note that while the data type should be different, it must be compatible with the actual data type of the field in the underlying table.

4 Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.

5 Select Data in the Field type radio group if it is not already selected.

6 Choose OK. The New Field dialog box closes, the newly defined data field replaces the existing field you specified in Step 1, and the component declaration in the data module or form's **type** declaration is updated.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector. For more information about editing field component properties and events, see "Setting persistent field properties and events" on page 17-10.

Defining a calculated field

A calculated field displays values calculated at runtime by a dataset's *OnCalcFields* event handler. For example, you might create a string field that displays concatenated values from other fields.

To create a calculated field in the New Field dialog box:

1 Enter a name for the calculated field in the Name edit box. Do not enter the name of an existing field.

2 Choose a data type for the field from the Type combo box.

3 Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.

4 Select Calculated or InternalCalc in the Field type radio group. InternalCalc is only available if you are working with a client dataset. The significant difference between these types of calculated fields is that the values calculated for an InternalCalc field are stored and retrieved as part of the client dataset's data.

5 Choose OK. The newly defined calculated field is automatically added to the end of the list of persistent fields in the Field editor list box, and the component declaration is automatically added to the form's or data module's **type** declaration.

6 Place code that calculates values for the field in the *OnCalcFields* event handler for the dataset. For more information about writing code to calculate field values, see "Programming a calculated field" on page 17-7.

Note To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector. For more information about editing field component properties and events, see "Setting persistent field properties and events" on page 17-10.

Programming a calculated field

After you define a calculated field, you must write code to calculate its value. Otherwise, it always has a null value. Code for a calculated field is placed in the *OnCalcFields* event for its dataset.

To program a value for a calculated field:

1. Select the dataset component from the Object Inspector drop-down list.
2. Choose the Object Inspector Events page.
3. Double-click the *OnCalcFields* property to bring up or create a *CalcFields* event handler for the dataset component.
4. Write the code that sets the values and other properties of the calculated field as desired.

For example, suppose you have created a *CityStateZip* calculated field for the *Customers* table on the *CustomerData* data module. *CityStateZip* should display a company's city, state, and zip code on a single line in a data-aware control.

To add code to the *CalcFields* procedure for the *Customers* table, select the *Customers* table from the Object Inspector drop-down list, switch to the Events page, and double-click the *OnCalcFields* property.

The *TCustomerData.CustomersCalcFields* procedure appears in the unit's source code window. Add the following code to the procedure to calculate the field:

```
CustomersCityStateZip.Value := CustomersCity.Value + ', ' + CustomersState.Value
  + ' ' + CustomersZip.Value;
```

Note When writing the *OnCalcFields* event handler for an internally calculated field, you can improve performance by checking the client dataset's *State* property and only recomputing the value when *State* is *dsInternalCalc*. See "Using internally calculated fields in client datasets" on page 20-19 for details.

Defining a lookup field

A lookup field is a read-only field that displays values at runtime based on search criteria you specify. In its simplest form, a lookup field is passed the name of an existing field to search on, a field value to search for, and a different field in a lookup dataset whose value it should display.

For example, consider a mail-order application that enables an operator to use a lookup field to determine automatically the city and state that correspond to the zip code a customer provides. The column to search on might be called *ZipTable.Zip*, the value to search for is the customer's zip code as entered in *Order.CustZip*, and the values to return would be those for the *ZipTable.City* and *ZipTable.State* columns of the record where the value of *ZipTable.Zip* matches the current value in the *Order.CustZip* field.

Note Unidirectional datasets do not support lookup fields.

To create a lookup field in the New Field dialog box:

1. Enter a name for the lookup field in the Name edit box. Do not enter the name of an existing field.
2. Choose a data type for the field from the Type combo box.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.

4 Select Lookup in the Field type radio group. Selecting Lookup enables the Dataset and Key Fields combo boxes.

5 Choose from the Dataset combo box drop-down list the dataset in which to look up field values. The lookup dataset must be different from the dataset for the field component itself, or a circular reference exception is raised at runtime. Specifying a lookup dataset enables the Lookup Keys and Result Field combo boxes.

6 Choose from the Key Fields drop-down list a field in the current dataset for which to match values. To match more than one field, enter field names directly instead of choosing from the drop-down list. Separate multiple field names with semicolons. If you are using more than one field, you must use persistent field components.

7 Choose from the Lookup Keys drop-down list a field in the lookup dataset to match against the Source Fields field you specified in step 6. If you specified more than one key field, you must specify the same number of lookup keys. To specify more than one field, enter field names directly, separating multiple field names with semicolons.

8 Choose from the Result Field drop-down list a field in the lookup dataset to return as the value of the lookup field you are creating.

When you design and run your application, lookup field values are determined before calculated field values are calculated. You can base calculated fields on lookup fields, but you cannot base lookup fields on calculated fields.

You can use the *LookupCache* property to hone the way lookup fields are determined. *LookupCache* determines whether the values of a lookup field are cached in memory when a dataset is first opened, or looked up dynamically every time the current record in the dataset changes. Set *LookupCache* to *True* to cache the values of a lookup field when the *LookupDataSet* is unlikely to change and the number of distinct lookup values is small. Caching lookup values can speed performance, because the lookup values for every set of *LookupKeyFields* values are preloaded when the *DataSet* is opened. When the current record in the *DataSet* changes, the field object can locate its *Value* in the cache, rather than accessing the *LookupDataSet*. This performance improvement is especially dramatic if the *LookupDataSet* is on a network where access is slow.

Tip You can use a lookup cache to provide lookup values programmatically rather than from a secondary dataset. Be sure that the *LookupDataSet* property is nil. Then, use the *LookupList* property's *Add* method to fill it with lookup values. Set the *LookupCache* property to *True*. The field will use the supplied lookup list without overwriting it with values from a lookup dataset.

If every record of *DataSet* has different values for *KeyFields*, the overhead of locating values in the cache can be greater than any performance benefit provided by the cache. The overhead of locating values in the cache increases with the number of distinct values that can be taken by *KeyFields*.

If *LookupDataSet* is volatile, caching lookup values can lead to inaccurate results. Call *RefreshLookupList* to update the values in the lookup cache. *RefreshLookupList* regenerates the *LookupList* property, which contains the value of the *LookupResultField* for every set of *LookupKeyFields* values.

When setting *LookupCache* at runtime, call *RefreshLookupList* to initialize the cache.

Defining an aggregate field

An aggregate field displays values from a maintained aggregate in a client dataset. An aggregate is a calculation that summarizes the data in a set of records. See "Using maintained aggregates" on page 20-20 for details about maintained aggregates.

To create an aggregate field in the New Field dialog box:

1. Enter a name for the aggregate field in the Name edit box. Do not enter the name of an existing field.
2. Choose aggregate data type for the field from the Type combo box.
3. Select Aggregate in the Field type radio group.
4. Choose OK. The newly defined aggregate field is automatically added and the client dataset's *Aggregates* property is automatically updated to include the appropriate aggregate specification.
5. Place the calculation for the aggregate in the *ExprText* property of the newly created aggregate field. For more information about defining an aggregate, see "Specifying aggregates" on page 20-20.

Once a persistent *TAggregateField* is created, a *TDBText* control can be bound to the aggregate field. The *TDBText* control will then display the value of the aggregate field that is relevant to the current record of the client data set.

Deleting persistent field components

Deleting a persistent field component is useful for accessing a subset of available columns in a table, and for defining your own persistent fields to replace a column in a table. To remove one or more persistent field components for a dataset:

1. Select the field(s) to remove in the Fields editor list box.
2. Press *Del*.

Note You can also delete selected fields by invoking the context menu and choosing Delete.

Fields you remove are no longer available to the dataset and cannot be displayed by data-aware controls. You can always recreate a persistent field component that you delete by accident, but any changes previously made to its properties or events is lost. For more information, see "Creating persistent fields" on page 17-4.

Note If you remove all persistent field components for a dataset, the dataset reverts to using dynamic field components for every column in the underlying database table.

Setting persistent field properties and events

You can set properties and customize events for persistent field components at design time. Properties control the way a field is displayed by a data-aware component, for example, whether it can appear in a *TDBGrid*, or whether its value can be modified. Events control what happens when data in a field is fetched, changed, set, or validated.

To set the properties of a field component or write customized event handlers for it, select the component in the Fields editor, or select it from the component list in the Object Inspector.

Setting display and edit properties at design time

To edit the display properties of a selected field component, switch to the Properties page on the Object Inspector window. The following table summarizes display properties that can be edited.

Table 17.3 Field component properties

Property	Purpose
Alignment	Left justifies, right justifies, or centers a field contents within a data-aware component.
ConstraintErrorMessage	Specifies the text to display when edits clash with a constraint condition.
CustomConstraint	Specifies a local constraint to apply to data during editing.
Currency	Numeric fields only. *True*: displays monetary values. *False* (default): does not display monetary values.
DisplayFormat	Specifies the format of data displayed in a data-aware control.
DisplayLabel	Specifies the column name for a field in a data-aware grid.
DisplayWidth	Specifies the width, in characters, of a grid column that displays this field.
EditFormat	Specifies the edit format of data in a data-aware control.
EditMask	Limits data-entry in an editable field to specified types and ranges of characters, and specifies any special, non-editable characters that appear within the field (hyphens, parentheses, and so on).
FieldKind	Specifies the type of field (data, lookup, calculated, or aggregate).
FieldName	Specifies the name of a column in the table from which the field derives its value and data type.
HasConstraints	Indicates whether there are constraint conditions imposed on a field.
ImportedConstraint	Specifies an SQL constraint imported from the database server. This is only used when the field gets its value from an application server running on another platform.
Index	Specifies the order of the field in a dataset.
LookupDataSet	Specifies the dataset used to look up field values when *Lookup* is *True*.
LookupKeyFields	Specifies the field(s) in the lookup dataset to match when doing a lookup.
LookupResultField	Specifies the field in the lookup dataset from which to copy values into this field.
MaxValue	Numeric fields only. Specifies the maximum value a user can enter for the field.
MinValue	Numeric fields only. Specifies the minimum value a user can enter for the field.
Name	Specifies the name used to refer to the field component in code.
Origin	Specifies the name of the field as it appears in the underlying database.
Precision	Numeric fields only. Specifies the number of significant digits.

Table 17.3 Field component properties (continued)

Property	Purpose
ReadOnly	*True*: Displays field values in data-aware components, but prevents editing. *False* (the default): Permits display and editing of field values.
Size	Specifies the maximum number of characters that can be displayed or entered in a string-based field, or the size, in bytes, of *TBytesField* and *TVarBytesField* fields.
Tag	Long integer available for programmer use in every component as needed.
Transliterate	*True* (default): specifies that translation to and from the respective locales will occur as data is transferred between a dataset and a database. *False*: Locale translation does not occur.
Visible	*True* (the default): Permits display of field in a data-aware grid. *False*: Prevents display of field in a data-aware grid component. User-defined components can make display decisions based on this property.

Not all properties are available for all field components. For example, a field component of type *TStringField* does not have *Currency*, *MaxValue*, or *DisplayFormat* properties, and a component of type *TFloatField* does not have a *Size* property.

While the purpose of most properties is straightforward, some properties, such as *Calculated*, require additional programming steps to be useful. Others, such as *DisplayFormat*, *EditFormat*, and *EditMask*, are interrelated; their settings must be coordinated. For more information about using *DisplayFormat*, *EditFormat*, and *EditMask*, see "Controlling and masking user input" on page 17-12.

Setting field component properties at runtime

You can use and manipulate the properties of field component at runtime. For example, the following code sets the *ReadOnly* property for the *CityStateZip* field in the *Customers* table to *True*:

```
CustomersCityStateZip.ReadOnly := True;
```

And this statement changes field ordering by setting the *Index* property of the *CityStateZip* field in the *Customers* table to 3:

```
CustomersCityStateZip.Index := 3;
```

Controlling and masking user input

The *EditMask* property provides a way to control the type and range of values a user can enter into a data-aware component associated with *TStringField*, *TDateField*, *TTimeField*, *TDateTimeField*, and *TSQLTimeStampField* components. You can use existing masks, or create your own. The easiest way to use and create edit masks is with the Input Mask editor. You can, however, enter masks directly into the *EditMask* field in the Object Inspector.

Note For *TStringField* components, the *EditMask* property is also its display format.

To invoke the Input Mask editor for a field component:

1 Select the component in the Fields editor or Object Inspector.
2 Click the Properties page in the Object Inspector.
3 Double-click the values column for the EditMask field in the Object Inspector, or click the ellipsis button. The Input Mask editor opens.

The Input Mask edit box lets you create and edit a mask format. The Sample Masks grid lets you select from predefined masks. If you select a sample mask, the mask format appears in the Input Mask edit box where you can modify it or use it as is. You can test the allowable user input for a mask in the Test Input edit box.

The Masks button lets you load a custom set of masks—if you have created one—into the Sample Masks grid for easy selection.

Using default formatting for numeric, date, and time fields

Kylix provides built-in display and edit format routines and intelligent default formatting for *TFloatField*, *TCurrencyField*, *TBCDField*, *TFMTBCDField*, *TIntegerField*, *TSmallIntField*, *TWordField*, *TDateField*, *TDateTimeField*, *TTimeField*, and *TSQLTimeStampField* components. To use these routines, you need do nothing.

Default formatting is performed by the following routines:

Table 17.4 Field component formatting routines

Routine	Used by . . .
FormatFloat	*TFloatField*, *TCurrencyField*
FormatDateTime	*TDateField*, *TTimeField*, *TDateTimeField*, *TSQLTimeStampField*
SQLTimeStampToString	*TSQLTimeStampField*
FormatCurr	*TCurrencyField*, *TBCDField*
BcdToStrF	*TFMTBcdField*

Only format properties appropriate to the data type of a field component are available for a given component.

Default formatting conventions for date, time, currency, and numeric values are based on the system locale. For example, using the default settings for the United States, a *TFloatField* column with the *Currency* property set to *True* sets the *DisplayFormat* property for the value 1234.56 to $1234.56, while the *EditFormat* is 1234.56.

At design time or runtime, you can edit the *DisplayFormat* and *EditFormat* properties of a field component to override the default display settings for that field. You can also write *OnGetText* and *OnSetText* event handlers to do custom formatting for field components at runtime.

Handling events

Like most components, field components have events associated with them. Methods can be assigned as handlers for these events. By writing these handlers you can react to the occurrence of events that affect data entered in fields through data-aware controls and perform actions of your own design. The following table lists the events associated with field components:

Table 17.5 Field component events

Event	Purpose
OnChange	Called when the value for a field changes.
OnGetText	Called when the value for a field component is retrieved for display or editing.
OnSetText	Called when the value for a field component is set.
OnValidate	Called to validate the value for a field component whenever the value is changed because of an edit or insert operation.

OnGetText and *OnSetText* events are primarily useful to programmers who want to do custom formatting that goes beyond the built-in formatting functions. *OnChange* is useful for performing application-specific tasks associated with data change, such as enabling or disabling menus or visual controls. *OnValidate* is useful when you want to control data-entry validation in your application before returning values to a database server.

To write an event handler for a field component:

1. Select the component.
2. Select the Events page in the Object Inspector.
3. Double-click the Value field for the event handler to display its source code window.
4. Create or edit the handler code.

Working with field component methods at runtime

Field components methods available at runtime let you convert field values from one data type to another, and enable you to set focus to the first data-aware control in a form that is associated with a field component.

Controlling the focus of data-aware components associated with a field is important when your application performs record-oriented data validation in a dataset event handler (such as *BeforePost*). Validation may be performed on the fields in a record whether or not its associated data-aware control has focus. Should validation fail for a particular field in the record, you want the data-aware control containing the faulty data to have focus so that the user can enter corrections.

You control focus for a field's data-aware components with a field's *FocusControl* method. *FocusControl* sets focus to the first data-aware control in a form that is associated with a field. An event handler can call a field's *FocusControl* method before validating the field. The following code illustrates how to call the *FocusControl* method for the *Company* field in the *Customers* table:

```
CustomersCompany.FocusControl;
```

The following table lists some other field component methods and their uses. For a complete list and detailed information about using each method, see the entries for *TField* and its descendants in the online *VCL Reference*.

Table 17.6 Selected field component methods

Method	Purpose
AssignValue	Sets a field value to a specified value using an automatic conversion function based on the field's type.
Clear	Clears the field and sets its value to NULL.
GetData	Retrieves unformatted data from the field.
IsValidChar	Determines if a character entered by a user in a data-aware control to set a value is allowed for this field.
SetData	Assigns unformatted data to this field.

Displaying, converting, and accessing field values

Data-aware controls such as *TDBEdit* and *TDBGrid* automatically display the values associated with field components. If editing is enabled for the dataset and the controls, data-aware controls can also send new and changed values to the database. In general, the built-in properties and methods of data-aware controls enable them to connect to datasets, display values, and make updates without requiring extra programming on your part. Use them whenever possible in your database applications. For more information about data-aware control, see Chapter 15, "Using data controls."

Standard controls can also display and edit database values associated with field components. Using standard controls, however, may require additional programming on your part. For example, when using standard controls, your application is responsible for tracking when to update controls because field values change. If the dataset has a datasource component, you can use its events to help you do this. In particular, the *OnDataChange* event lets you know when you may need to update a control's value and the *OnStateChange* event can help you determine when to enable or disable controls. For more information on these events, see "Responding to changes mediated by the data source" on page 15-4.

The following topics discuss how to work with field values so that you can display them in standard controls.

Displaying field component values in standard controls

An application can access the value of a dataset column through the *Value* property of a field component. For example, the following *OnDataChange* event handler updates the text in a *TEdit* control because the value of the *CustomersCompany* field may have changed:

```
procedure TForm1.CustomersDataChange(Sender: TObject, Field: TField);
begin
  Edit3.Text := CustomersCompany.Value;
end;
```

This method works well for string values, but may require additional programming to handle conversions for other data types. Fortunately, field components have built-in functions for handling conversions.

Note You can also use Variants to access and set field values. For more information about using variants to access and set field values, see "Accessing field values with the default dataset property" on page 17-17.

Converting field values

Conversion properties attempt to convert one data type to another. For example, the *AsString* property converts numeric and Boolean values to string representations. The following table lists field component conversion properties, and which properties are recommended for field components by field-component class:

	AsVariant	AsString	AsInteger	AsFloat AsCurrency AsBCD	AsDateTime AsSQLTimeStamp	AsBoolean
TStringField	yes	NA	yes	yes	yes	yes
TWideStringField	yes	yes	yes	yes	yes	yes
TIntegerField	yes	yes	NA	yes		
TSmallIntField	yes	yes	yes	yes		
TWordField	yes	yes	yes	yes		
TLargeintField	yes	yes	yes	yes		
TFloatField	yes	yes	yes	yes		
TCurrencyField	yes	yes	yes	yes		
TBCDField	yes	yes	yes	yes		
TFMTBCDField	yes	yes	yes	yes		
TDateTimeField	yes	yes		yes	yes	
TDateField	yes	yes		yes	yes	
TTimeField	yes	yes		yes	yes	
TSQLTimeStampField	yes	yes		yes	yes	
TBooleanField	yes	yes				
TBytesField	yes	yes				
TVarBytesField	yes	yes				
TBlobField	yes	yes				
TMemoField	yes	yes				
TGraphicField	yes	yes				
TVariantField	NA	yes	yes	yes	yes	yes
TAggregateField	yes	yes				

Note that some columns in the table refer to more than one conversion property (such as *AsFloat*, *AsCurrency*, and *AsBCD*). This is because all field data types that support one of those properties always support the others as well.

Note also that the *AsVariant* property can translate among all data types. For any datatypes not listed above, *AsVariant* is also available (and is, in fact, the only option). When in doubt, use *AsVariant*.

In some cases, conversions are not always possible. For example, *AsDateTime* can be used to convert a string to a date, time, or datetime format only if the string value is in a recognizable datetime format. A failed conversion attempt raises an exception.

In some other cases, conversion is possible, but the results of the conversion are not always intuitive. For example, what does it mean to convert a *TDateTimeField* value into a float format? *AsFloat* converts the date portion of the field to the number of days since 12/31/1899, and it converts the time portion of the field to a fraction of 24 hours. Table 17.7 lists permissible conversions that produce special results:

Table 17.7 Special conversion results

Conversion	Result
String to Boolean	Converts “True,” “False,” “Yes,” and “No” to Boolean. Other values raise exceptions.
Float to Integer	Rounds float value to nearest integer value.
DateTime or SQLTimeStamp to Float	Converts date to number of days since 12/31/1899, time to a fraction of 24 hours.
Boolean to String	Converts any Boolean value to “True” or “False.”

In other cases, conversions are not possible at all. In these cases, attempting a conversion also raises an exception.

Conversion always occurs before an actual assignment is made. For example, the following statement converts the value of *CustomersCustNo* to a string and assigns the string to the text of an edit control:

```
Edit1.Text := CustomersCustNo.AsString;
```

Conversely, the next statement assigns the text of an edit control to the *CustomersCustNo* field as an integer:

```
MyTableMyField.AsInteger := StrToInt(Edit1.Text);
```

Accessing field values with the default dataset property

The most general method for accessing a field’s value is to use Variants with the *FieldValues* property. For example, the following statement puts the value of an edit box into the *CustNo* field in the *Customers* table:

```
Customers.FieldValues['CustNo'] := Edit2.Text;
```

Because the *FieldValues* property is of type Variant, it automatically converts other datatypes into a Variant value.

For more information about Variants, see the online help.

Accessing field values with a dataset's Fields property

You can access the value of a field with the *Fields* property of the dataset component to which the field belongs. *Fields* maintains an indexed list of all the fields in the dataset. Accessing field values with the *Fields* property is useful when you need to iterate over a number of columns, or if your application works with tables that are not available to you at design time.

To use the *Fields* property you must know the order and data types of fields in the dataset. You use an ordinal number to specify the field to access. The first field in a dataset is numbered 0. Field values must be converted as appropriate using each field component's conversion properties. For more information about field component conversion properties, see "Converting field values" on page 17-16.

For example, the following statement assigns the current value of the seventh column (Country) in the *Customers* table to an edit control:

```
Edit1.Text := CustTable.Fields[6].AsString;
```

Conversely, you can assign a value to a field by setting the *Fields* property of the dataset to the desired field. For example:

```
begin
  Customers.Edit;
  Customers.Fields[6].AsString := Edit1.Text;
  Customers.Post;
end;
```

Accessing field values with a dataset's FieldByName method

You can also access the value of a field with a dataset's *FieldByName* method. This method is useful when you know the name of the field you want to access, but do not have access to the underlying table at design time.

To use *FieldByName*, you must know the dataset and name of the field you want to access. You pass the field's name as an argument to the method. To access or change the field's value, convert the result with the appropriate field component conversion property, such as *AsString* or *AsInteger*. For example, the following statement assigns the value of the *CustNo* field in the *Customers* dataset to an edit control:

```
Edit2.Text := Customers.FieldByName('CustNo').AsString;
```

Conversely, you can assign a value to a field:

```
begin
  Customers.Edit;
  Customers.FieldByName('CustNo').AsString := Edit2.Text;
  Customers.Post;
end;
```

Checking a field's current value

If your application uses a client datasetto update data from a database server, and you encounter difficulties when updating records, you can use the *CurValue* property to examine the field value in the record causing problems. *CurValue* represents the current value of the field on the database server, reflecting any changes made by other users of the database.

CurValue is only available inside an *OnUpdateError* or *OnReconcileError* event handler. In these event handlers, you can compare *CurValue* (the value on the server) to *NewValue* (the unposted value that caused the problem) and *OldValue* (the value that was originally assigned to the field before any edits were made). *CurValue* differs from *OldValue* only if another user changed the value of the field after *OldValue* was read.

Setting a default value for a field

You can specify how a default value for a field in a client dataset should be calculated at runtime using the *DefaultExpression* property. *DefaultExpression* can be any valid SQL value expression that does not refer to field values. If the expression contains literals other than numeric values, they must appear in quotes. For example, a default value of noon for a time field would be

```
'12:00:00'
```

including the quotes around the literal value.

Note If the underlying database table defines a default value for the field, the default you specify in *DefaultExpression* takes precedence. That is because *DefaultExpression* is applied when the client dataset posts the record containing the field, which occurs before the edited record is applied to the database server.

Specifying constraints

Most production SQL databases use constraints to impose conditions on the possible values for a field. For example, a field may not permit NULL values, may require that its value be unique for that column, or that its values be greater than *0* and less than *150*. These constraints are enforced when your application applies updates to the database server.

In addition to these server-enforced constraints, you can create and use custom constraints that are applied locally to the fields in client datasets. Custom constraints can duplicate the server constraints (so that you detect errors immediately when posting edits to the change log rather than later when you attempt to apply updates), or they can impose additional, application-defined limits. Custom constraints provide validation of data entry, but they cannot be applied against data received from or sent to a server application.

To create a custom constraint, set the *CustomConstraint* property to specify a constraint condition, and set *ConstraintErrorMessage* to the message to display when a user violates the constraint at runtime.

CustomConstraint is an SQL string that specifies any application-specific constraints imposed on the field's value. Set *CustomConstraint* to limit the values that the user can enter into a field. *CustomConstraint* can be any valid SQL search expression such as

```
x > 0 and x < 100
```

The name used to refer to the value of the field can be any string that is not a reserved SQL keyword, as long as it is used consistently throughout the constraint expression.

Using object fields

Object fields are fields that represent a composite of other, simpler datatypes. These include ADT (Abstract Data Type) fields, array fields, DataSet fields, and Reference fields. All of these field types either contain or reference child fields or other data sets.

ADT fields and array fields are fields that contain child fields. The child fields of an ADT field can be any scalar or object type (that is, any other field type). These child fields may differ in type from each other. An array field contains an array of child fields, all of the same type.

Dataset and reference fields map to fields that access other data sets. A dataset field provides access to a nested (detail) dataset and a reference field stores a pointer (reference) to another persistent object (ADT).

Table 17.8 Types of object field components

Component name	Purpose
TADTField	Represents an ADT (Abstract Data Type) field.
TArrayField	Represents an array field.
TDataSetField	Represents a field that contains a nested data set reference.
TReferenceField	Represents a REF field, a pointer to an ADT.

When you add fields with the Fields editor to a dataset that contains object fields, persistent object fields of the correct type are automatically created for you. Adding persistent object fields to a dataset automatically sets the dataset's *ObjectView* property to *True*, which instructs the dataset to store these fields hierarchically, rather than flattening them out as if the constituent child fields were separate, independent fields.

The following properties are common to all object fields and provide the functionality to handle child fields and datasets.

Table 17.9 Common object field descendant properties

Property	Purpose
Fields	Contains the child fields belonging to the object field.
ObjectType	Classifies the object field.
FieldCount	Number of child fields that comprise the object field.
FieldValues	Provides access to the values of the child fields.

Displaying ADT and array fields

Both ADT and array fields contain child fields that can be displayed through data-aware controls.

Data-aware controls such as *TDBEdit* that represent a single field value display child field values in an uneditable comma delimited string. In addition, if you set the control's *DataField* property to the child field instead of the object field itself, the child field can be viewed an edited just like any other normal data field.

A *TDBGrid* control displays ADT and array field data differently, depending on the value of the dataset's *ObjectView* property. When *ObjectView* is *False*, each child field appears in a single column. When *ObjectView* is *True*, an ADT or array field can be expanded and collapsed by clicking on the arrow in the title bar of the column. When the field is expanded, each child field appears in its own column and title bar, all below the title bar of the ADT or array itself. When the ADT or array is collapsed, only one column appears with an uneditable comma delimited string containing the child fields.

Working with ADT fields

ADTs are user-defined types created on the server, and are similar to the record type. An ADT can contain most scalar field types, array fields, reference fields, and nested ADTs.

There are a variety of ways to access the data in ADT field types. These are illustrated in the following examples, which assign a child field value to an edit box called *CityEdit*, and use the following ADT structure,

```
Address
  Street
  City
  State
  Zip
```

Using persistent field components

The easiest way to access ADT field values is to use persistent field components. For the ADT structure above, the following persistent fields can be added to the *Customer* table using the Fields editor:

```
CustomerAddress: TADTField;
CustomerAddrStreet: TStringField;
CustomerAddrCity: TStringField;
CustomerAddrState: TStringField;
CustomerAddrZip: TStringField;
```

Given these persistent fields, you can simply access the child fields of an ADT field by name:

```
CityEdit.Text := CustomerAddrCity.AsString;
```

Although persistent fields are the easiest way to access ADT child fields, it is not possible to use them if the structure of the dataset is not known at design time. When accessing ADT child fields without using persistent fields, you must set the dataset's *ObjectView* property to *True*.

Using the dataset's FieldByName method

You can access the children of an ADT field using the dataset's *FieldByName* method by qualifying the name of the child field with the ADT field's name:

```
CityEdit.Text := Customer.FieldByName(‘Address.City’).AsString;
```

Using the dateset's FieldValues property

You can also use qualified field names with a dataset's *FieldValues* property:

```
CityEdit.Text := Customer['Address.City'];
```

Note that you can omit the property name (FieldValues) because FieldValues is the dataset's default property.

Note Unlike other runtime methods for accessing ADT child field values, the *FieldValues* property works even if the dataset's *ObjectView* property is *False*.

Using the ADT field's FieldValues property

You can access the value of a child field with the TADTField's *FieldValues* property. *FieldValues* accepts and returns a *Variant*, so it can handle and convert fields of any type. The index parameter is an integer value that specifies the offset of the field.

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).FieldValues[1];
```

Because FieldValues is the default property of *TADTField*, the property name (FieldValues) can be omitted. Thus, the following statement is equivalent to the one above:

```
CityEdit.Text := TADTField(Customer.FieldByName('Address'))[1];
```

Using the ADT field's Fields property

Each ADT field has a *Fields* property that is analogous to the *Fields* property of a dataset. Like the *Fields* property of a dataset, you can use it to access child fields by position:

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).Fields[1].AsString;
```

or by name:

```
CityEdit.Text :=
TADTField(Customer.FieldByName('Address')).Fields.FieldByName('City').AsString;
```

Working with array fields

Array fields consist of a set of fields of the same type. The field types can be scalar (for example, float, string), or non-scalar (an ADT), but an array field of arrays is not permitted. The *SparseArrays* property of *TDataSet* determines whether a unique *TField* object is created for each element of the array field.

There are a variety of ways to access the data in array field types. If you are not using persistent fields, the dataset's *ObjectView* property must be set to *True* before you can access the elements of an array field.

Using persistent fields

You can map persistent fields to the individual array elements in an array field. For example, consider an array field *TelNos_Array*, which is a six element array of strings. The following persistent fields created for the *Customer* table component represent the *TelNos_Array* field and its six elements:

```
CustomerTelNos_Array: TArrayField;
CustomerTelNos_Array0: TStringField;
CustomerTelNos_Array1: TStringField;
CustomerTelNos_Array2: TStringField;
CustomerTelNos_Array3: TStringField;
CustomerTelNos_Array4: TStringField;
CustomerTelNos_Array5: TStringField;
```

Given these persistent fields, the following code uses a persistent field to assign an array element value to an edit box named *TelEdit*.

```
TelEdit.Text := CustomerTelNos_Array0.AsString;
```

Using the array field's FieldValues property

You can access the value of a child field with the array field's *FieldValues* property. *FieldValues* accepts and returns a *Variant*, so it can handle and convert child fields of any type. For example,

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array')).FieldValues[1];
```

Because *FieldValues* is the default property of *TArrayField*, this can also be written

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array'))[1];
```

Using the array field's Fields property

TArrayField has a *Fields* property that you can use to access individual sub-fields. This is illustrated below, where an array field (*OrderDates*) is used to populate a list box with all non-null array elements:

```
for I := 0 to OrderDates.Size - 1 do
begin
  if not OrderDates.Fields[I].IsNull then
    OrderDateListBox.Items.Add(OrderDates[I]);
end;
```

Working with dataset fields

Dataset fields provide access to data stored in a nested dataset. The *NestedDataSet* property references the nested dataset. The data in the nested dataset is then accessed through the fields of the nested dataset.

Displaying dataset fields

TDBGrid controls enable the display of data stored in data set fields. In a *TDBGrid* control, a dataset field is indicated in each cell of a dataset column with a "(DataSet)", and at runtime an ellipsis button also exists to the right. Clicking on the ellipsis brings up a new form with a grid displaying the dataset associated with the current record's dataset field. This form can also be brought up programmatically with the DB grid's *ShowPopupEditor* method. For example, if the seventh column in the grid represents a dataset field, the following code displays the dataset associated with that field for the current record.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

Accessing data in a nested dataset

A dataset field is not normally bound directly to a data aware control. Rather, since a nested dataset is another dataset, you use another dataset component to access its data. The type of dataset you use is determined by the parent dataset (the one with the dataset field.) For example, a dataset field in a *TClientDataSet* object must use another *TClientDataSet* to represent its value.

To access the data in a dataset field,

1. Create a persistent *TDataSetField* object by invoking the Fields editor for the parent dataset.
2. Create a dataset to represent the values in that dataset field.
3. Set that *DataSetField* property of the dataset created in step 2 to the persistent dataset field you created in step 1.

If the nested dataset field for the current record has a value, the detail dataset component will contain records with the nested data; otherwise, the detail dataset will be empty.

Working with reference fields

Reference fields store a pointer or reference to another ADT object. This ADT object is a single record of another object table. Reference fields always refer to a single record in a dataset (object table). The data in the referenced object is returned in a nested dataset, but can also be accessed via the *Fields* property on the *TReferenceField*.

Displaying reference fields

In a *TDBGrid* control a reference field is designated in each cell of the dataset column, with (Reference) and, at runtime, an ellipsis button to the right. At runtime, clicking on the ellipsis brings up a new form with a grid displaying the object associated with the current record's reference field.

This form can also be brought up programmatically with the DB grid's *ShowPopupEditor* method. For example, if the seventh column in the grid represents a reference field, the following code will display the object associated with that field for the current record.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

Accessing data in a reference field

You can access the data in a reference field in the same way you access a nested dataset:

1 Create a persistent *TDataSetField* object by invoking the Fields editor for the parent dataset.
2 Create a dataset to represent the value of that dataset field.
3 Set that *DataSetField* property of the dataset created in step 2 to the persistent dataset field you created in step 1.

If the reference is assigned, the reference dataset will contain a single record with the referenced data. If the reference is null, the reference dataset will be empty.

You can also use the reference field's Fields property to access the data in a reference field. For example, the following lines are equivalent and assign data from the reference field *CustomerRefCity* to an edit box called *CityEdit*:

```
CityEdit.Text := CustomerRefCity.Fields[1].AsString;
CityEdit.Text := CustomerRefCity.NestedDataSet.Fields[1].AsString;
```

Chapter

18

Using unidirectional datasets

Unidirectional datasets provide the mechanism by which an application reads data from an SQL database table. They are designed for quick lightweight access to database information, with minimal overhead. Unidirectional datasets send an SQL command to the database server, and if the command returns a set of records, obtain a unidirectional cursor for accessing those records. They do not buffer data in memory, which makes them faster and less resource-intensive than other types of dataset. However, because there are no buffered records, unidirectional datasets are also less flexible than other datasets. Many of the capabilities introduced by *TDataSet* are either unimplemented in unidirectional datasets, or cause them to raise exceptions. For example:

- The only supported navigation methods are the *First* and *Next* methods. Most others raise exceptions. Some, such as the methods involved in bookmark support, simply do nothing.
- There is no built-in support for editing because editing requires a buffer to hold the edits. The *CanModify* property is always *False*, so attempts to put the dataset into edit mode always fail. You can, however, use them to update data using an SQL UPDATE command or provide conventional editing support by connecting them to a client dataset (as described in “Using a client dataset to buffer records” on page 14-9).
- There is no support for filters, because they need to work with multiple records, which requires buffering. If you try to filter a unidirectional dataset, it raises an exception. Instead, all limits on what data appears must be imposed using the SQL command that defines the data for the dataset.
- There is no support for lookup fields, which require buffering to hold multiple records containing lookup values. If you define a lookup field on a unidirectional dataset, it does not work properly.

Despite these limitations, unidirectional datasets are a powerful way to access data. They are fast, and very simple to use and deploy.

Types of unidirectional datasets

The *dbExpress* page of the component palette contains four types of unidirectional dataset: *TSQLDataSet*, *TSQLQuery*, *TSQLTable*, and *TSQLStoredProc*.

TSQLDataSet is the most general of the four. You can use an SQL dataset to represent any data available through *dbExpress*, or to send commands to a database accessed through *dbExpress*. This is the recommended component to use for working with database tables in new database applications.

You can use *TSQLQuery* for almost everything you can accomplish with *TSQLDataSet*. *TSQLQuery* differs primarily in that it can't be used for stored procedures. (Many servers support an extension to SQL that lets you create queries to execute stored procedures, but there is no standard syntax for this) *TSQLQuery* is intended for compatibility with Windows applications, so that it is easier to port applications that use query components to Linux.

TSQLTable is a special-purpose dataset when you want to represent all of the fields and all of the records in a single database table. *TSQLTable* is intended for compatibility with Windows applications so that it is easier to port applications that use table components to Linux.

TSQLStoredProc is a special-purpose dataset for executing a stored procedure. You can also use *TSQLDataSet* to execute stored procedures: *TSQLStoredProc* is intended for compatibility with Windows applications so that it is easier to port applications that use stored procedure components to Linux.

Connecting to the Server

The first step when working with a unidirectional dataset is to connect it to a database server. At design time, once a dataset has an active connection to a database server, the Object Inspector can provide drop-down lists of values for other properties. For example, when representing a stored procedure, you must have an active connection before the Object Inspector can list what stored procedures are available on the server.

The connection to a database server is represented by a separate *TSQLConnection* component. *TSQLConnection* identifies the database server and several connection parameters (including which database to use on the server, the host name of the machine running the server, the username, password, and so on). For information about setting up a connection using *TSQLConnection*, see Chapter 19, "Connecting to databases."

To connect a unidirectional dataset, you must specify the *TSQLConnection* that forms the connection. Do this using the *SQLConnection* property. At design time, you can choose the SQL connection component from a drop-down list in the Object Inspector. If you make this assignment at runtime, be sure that the connection is active:

```
SQLDataSet1.SQLConnection := SQLConnection1;
SQLConnection1.Connected := True;
```

Typically, all unidirectional datasets in an application share the same connection component, unless you are working with data from multiple database servers or with a database server such as MySQL that does not support multiple statements.

Specifying what data to display

There are a number of ways to specify what data a unidirectional dataset represents. Which method you choose depends on the type of unidirectional dataset you are using and whether the information comes from a single database table, the results of a query, or from a stored procedure.

When you work with a *TSQLDataSet* component, use the *CommandType* property to indicate where the dataset gets its data. *CommandType* can take any of the following values:

- *ctQuery*: When *CommandType* is *ctQuery*, *TSQLDataSet* executes a query you specify. If the query is a SELECT command, the dataset contains the resulting set of records.
- *ctTable*: When *CommandType* is *ctTable*, *TSQLDataSet* retrieves all of the records from a specified table.
- *ctStoredProc*: When *CommandType* is *ctStoredProc*, *TSQLDataSet* executes a stored procedure. If the stored procedure returns a cursor, the dataset contains the returned records.

Note You can also populate the unidirectional dataset with metadata about what is available on the server. For information on how to do this, see "Accessing schema information" on page 18-15.

Representing the results of a query

Using a query is the most general way to specify a set of records. Queries are simply commands written in SQL. They need not return a set of records; SQL defines queries such as UPDATE queries that perform actions on the server. Queries that do not return records are discussed in greater detail in "Executing commands that do not return records" on page 18-11.

Most queries that return records are SELECT commands. Typically, they define the fields to include, the tables from which to select those fields, conditions that limit what records to include, and the order of the resulting dataset. For example:

```
SELECT CustNo, OrderNo, SaleDate
FROM Orders
WHERE CustNo = 1225
ORDER BY SaleDate
```

You can use either *TSQLDataSet* or *TSQLQuery* to represent the result of a query.

Specifying a query using TSQLDataSet

When using *TSQLDataSet*, assign the text of the query statement to the *CommandText* property:

```
SQLDataSet1.CommandText := 'SELECT CustName, Address FROM Customer';
```

At design time, you can type the query directly into the Object Inspector, or, if the SQL dataset already has an active connection to the database, you can click the elipsis button by the *CommandText* property to display the Command Text editor. The Command Text editor lists the available tables, and the fields in those tables, to make it easier to compose your queries.

Specifying a query using TSQLQuery

When using *TSQLQuery*, assign the query to the *SQL* property instead. Unlike the *CommandText* property of *TSQLDataSet*, which is a string, the *SQL* property is a *TStrings* object. Each separate string in this *TStrings* object is a separate line of the query. Using multiple lines does not affect the way the query executes on the server, but can make it easier to assemble a query from multiple sources:

```
SQLQuery1.SQL.Clear;
SQLQuery1.SQL.Add('SELECT ' + Edit1.Text + ' FROM ' + Edit2.Text);
if Length(Edit3.Text) <> 0 then
  SQLQuery1.SQL.Add('ORDER BY ' + Edit3.Text)
```

At design time, use the String List editor to specify the query. Click the elipsis button by the *SQL* property in the Object Inspector to display the String List editor.

One advantage of using *TSQLQuery* is that, because the *SQL* property is a *TStrings* object, you can load the text of the query from a file by calling the *TStrings.LoadFromFile* method:

```
SQLQuery1.SQL.LoadFromFile('/usr/queries/custquery.sql');
```

Using parameters in queries

The previous topic showed how to build a query dynamically at runtime. However, you can accomplish the same type of thing for queries written at design time by using parameters.

A parameterized SQL statement contains parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values that appear in the SQL statement. For example, in the following SELECT statement, a parameter is used to specify a selection criterion:

```
SELECT CustNo, OrderNo, SaleDate FROM Orders
WHERE CustNo = :CustNumber
```

In this SQL statement, *:CustNumber* is a placeholder for the actual value supplied to the statement at runtime by your application. Note that the name, *:CustNumber*, begins with a colon. Parameter names must start with a colon (:). You can also include unnamed parameters by adding a question mark (?) to your query. Unnamed parameters are identified by position, because they do not have unique names.

Before the dataset can execute the query, you must supply values for any parameters in the query text. The dataset uses its *Params* property to store these values. *Params* is a collection of *TParam* objects, where each *TParam* object represents a single parameter. When you specify the text for the query (using the *CommandText* or *SQL* property), the dataset generates this set of *TParam* objects, and initializes any of their properties that it can deduce from the query.

Note You can suppress the automatic generation of *TParam* objects in response to changing the query text by setting the *ParamCheck* property to *False*. See “Creating and modifying server metadata” on page 18-12 for an example where this is useful.

Note You do not need to explicitly assign parameter values if the dataset uses a datasource to obtain parameter values from another dataset. This process is described in “Setting up master/detail relationships” on page 18-13.

Setting up parameters at design time

At design time, you can specify parameter values using the parameter collection editor. To display the parameter collection editor, click on the ellipsis button for the *Params* property in the Object Inspector. If the SQL statement does not contain any parameters, no *TParam* objects are listed in the collection editor.

Note The parameter collection editor is the same collection editor that appears for other collection properties. Because the editor is shared with other properties, its right-click context menu contains the Add and Delete commands. However, they are never enabled for dataset parameters. The only way to add or delete parameters is in the SQL statement itself.

For each parameter, select it in the parameter collection editor. Then use the Object Inspector to modify its properties:

- The *DataType* property lists the data type for the parameter’s value. This value may be correctly initialized, if the dataset could deduce the value type from the query. If the dataset could not deduce the type, *DataType* is *ftUnknown*, and you must change it to indicate the type of the parameter value.
- The *ParamType* property lists the type of the selected parameter. For queries, this is always initialized to *ptInput*, because queries can only contain input parameters.
- The *Value* property specifies a value for the selected parameter. You can leave *Value* blank if your application supplies parameter values at runtime.

Setting parameters at runtime

To create parameters at runtime, you can use the

- *ParamByName* method to assign values to a parameter based on its name.
- *Params* property to assign values to a parameter based on the parameter’s ordinal position in the SQL statement.
- *Params.ParamValues* property to assign values to one or more parameters in a single command line, based on the name of each parameter set.

The following code uses *ParamByName* to assign the text of an edit box to the :Capital parameter:

```
SQLDataSet1.ParamByName('Capital').AsString := Edit1.Text;
```

The same code can be rewritten using the *Params* property, using an index of 0 (assuming the :Capital parameter is the first parameter in the SQL statement):

```
SQLDataSet1.Params[0].AsString := Edit1.Text;
```

The command line below sets three parameters at once, using the *Params.ParamValues* property:

```
SQLDataSet1.Params.ParamValues['Name;Capital;Continent'] :=
  VarArrayOf([Edit1.Text, Edit2.Text, Edit3.Text]);
```

Note that *ParamValues* uses Variants, avoiding the need to cast values.

Representing the records in a table

When you want to represent all of the fields and all of the records in a single underlying database table, you can use either *TSQLDataSet* or *TSQLTable* to generate the query for you rather than writing the SQL yourself.

Note If server performance is a concern, you may want to compose the query explicitly rather than relying on an automatically-generated query. Automatically-generated queries use wildcards rather than explicitly listing all of the fields in the table. This results in slightly slower performance on the server. The wildcard (*) in automatically-generated queries is more robust to changes in the fields on the server.

Representing a table using TSQLDataSet

To make *TSQLDataSet* generate a query to fetch all fields and all records of a single database table, set the *CommandType* property to *ctTable*.

When *CommandType* is *ctTable*, *TSQLDataSet* generates a query based on the values of two properties:

- *CommandText* specifies the name of the database table that the *TSQLDataSet* object should represent.
- *SortFieldNames* lists the names of any fields to use to sort the data, in the order of significance.

For example, if you specify the following:

```
SQLDataSet1.CommandType := ctTable;
SQLDataSet1.CommandText := 'Employee';
SQLDataSet1.SortFieldNames := 'HireDate,Salary'
```

TSQLDataSet generates the following query, which lists all the records in the Employee table, sorted by HireDate and, within HireDate, by Salary:

```
select * from Employee order by HireDate, Salary
```

Representing a table using TSQLTable

When using *TSQLTable*, specify the table you want using the *TableName* property.

To specify the order of fields in the dataset, you must specify an index. There are two ways to do this:

- Set the *IndexName* property to the name of an index defined on the server that imposes the order you want.
- Set the *IndexFieldNames* property to a semicolon-delimited list of field names on which to sort. *IndexFieldNames* works like the *SortFieldNames* property of *TSQLDataSet*, except that it uses a semicolon instead of a comma as a delimiter.

Representing the results of a stored procedure

Stored procedures are sets of SQL statements that are named and stored on an SQL server. They typically handle frequently-repeated database-related tasks. They are especially useful for operations that act upon large numbers of rows in database tables or that use aggregate or mathematical functions. Using stored procedures typically improves the performance of a database application by:

- Taking advantage of the server's usually greater processing power and speed.
- Reducing network traffic by moving processing to the server.

Stored procedures may or may not return data. Those that return data may return it as a cursor (similar to the results of a SELECT query), as multiple cursors (effectively returning multiple datasets), or they may return data in output parameters. These differences depend in part on the server: Some servers do not allow stored procedures to return data, or only allow output parameters. Some servers do not support stored procedures at all. See your server documentation to determine what is available.

How you indicate the stored procedure you want to execute depends on the type of unidirectional dataset you are using.

Specifying a stored procedure using TSQLDataSet

When using *TSQLDataSet*, to specify a stored procedure:

- Set the *CommandType* property to *ctStoredProc*.
- Specify the name of the stored procedure as the value of the *CommandText* property:

```
SQLDataSet1.CommandType := ctStoredProc;
SQLDataSet1.CommandText := 'MyStoredProcName';
```

Specifying a stored procedure using TSQLStoredProc

When using *TSQLStoredProc*, you need only specify the name of the stored procedure as the value of the *StoredProcName* property.

```
SQLStoredProc1.StoredProcName := 'MyStoredProcName';
```

Working with stored procedure parameters

There are four types of parameters that can be associated with stored procedures:

- *Input parameters*, used to pass values to a stored procedure for processing.

- *Output parameters*, used by a stored procedure to pass return values to an application.
- *Input/output parameters*, used to pass values to a stored procedure for processing, and used by the stored procedure to pass return values to the application.
- A *result parameter*, used by some stored procedures to return an error or status value to an application. A stored procedure can only return one result parameter.

Whether a stored procedure uses a particular type of parameter depends both on the general language implementation of stored procedures on your database server and on a specific instance of a stored procedure. For any server, individual stored procedures may or may not use input parameters. On the other hand, some uses of parameters are server-specific. For example, the InterBase implementation of stored procedures never returns a result parameter.

Access to stored procedure parameters is provided by *TParam* objects in the *Params* property. As with query parameters, *TSQLDataSet* and *TSQLStoredProc* automatically generate the *TParam* objects for each parameter when you set the *CommandText* (*TSQLDataSet*) or *StoredProcName* (*TSQLStoredProc*) property.

Setting up parameters at design time

As with query parameters, you can specify stored procedure parameter values at design time using the parameter collection editor. To display the parameter collection editor, click on the ellipsis button for the *Params* property in the Object Inspector.

Important You can assign values to input parameters by selecting them in the parameter collection editor and using the Object Inspector to set the *Value* property. However, do not change the names or data types for input parameters reported by the server. Otherwise, when you execute the stored procedure an exception is raised.

Some servers do not report parameter names or data types. In these cases, you must set up the parameters manually using the parameter collection editor. Right click and choose Add to add parameters. For each parameter you add,

- Assign the name (defined by the procedure on the server) as the value of the *Name* property.
- Indicate whether it is an input, output, input/output, or result parameter by setting the *ParamType* property.
- Indicate the data type of the parameter's value by setting the *DataType* property.

Some servers provide parameter names, but do not return all parameter information. Check the *DataType* and *ParamType* properties for each parameter and fix any that are set to *ftUnknown* or *ptUnknown*.

Note You can never set values for output and result parameters. These types of parameters have values set by the execution of the stored procedure.

Using parameters at runtime

At runtime, you can access individual parameter objects to assign values to input parameters and to retrieve values from output parameters. As with query parameters, you can use the *ParamByName* method to access individual parameters based on their names. For example, the following code sets the value of an input/ output parameter, executes the stored procedure, and retrieves the returned value:

```
with SQLDataSet1 do
begin
  ParamByName('IN_OUTVAR').AsInteger := 103;
  ExecProc;
  IntegerVar := ParamByName('IN_OUTVAR').AsInteger;
end;
```

Because some servers do not report parameter names or data types, you may need to set up parameters in code if you do not specify the name of the stored procedure until runtime. To check whether this is necessary, try specifying a stored procedure on the target database at design time, and use the parameter collection editor to check whether the dataset automatically sets up parameters.

The following example illustrates how to set up parameters at runtime if they are not supplied automatically:

```
var
  P1, P2: TParam;
begin
  with SQLDataSet1 do
  begin
    CommandType := ctStoredProc;
    CommandText := 'GET_EMP_PROJ';
    Params.Clear;
    P1 := TParam.Create(Params, ptInput);
    P2 := TParam.Create(Params, ptOutput);
    try
      Params[0].Name := ‘EMP_NO’;
      Params[0].DataType := ftSmallint;
      Params[1].Name := ‘PROJ_ID’;
      Params[1].DataType := ftString;
      ParamByname(‘EMP_NO’).AsSmallInt := 52;
      ExecProc;
      Edit1.Text := ParamByname(‘PROJ_ID’).AsString;
    finally
      P1.Free;
      P2.Free;
    end;
  end;
end;
```

Fetching the data

Once you have specified the source of the data, you must fetch the data before your application can access it. Once the dataset has fetched the data, data-aware controls linked to the dataset through a data source automatically display data values and client datasets linked to the dataset through a provider can be populated with records.

As with any dataset, there are two ways to fetch the data for a unidirectional dataset:

- Set the *Active* property to *True*, either at design time in the Object Inspector, or in code at runtime:

  ```
  CustQuery.Active := True;
  ```

- Call the *Open* method at runtime,

  ```
  CustQuery.Open;
  ```

Use the *Active* property or the *Open* method with any unidirectional dataset that obtains records from the server. It does not matter whether these records come from a SELECT query (including automatically-generated queries when the *CommandType* is *ctTable*) or a stored procedure.

Preparing the dataset

Before a query or stored procedure can execute on the server, it must first be "prepared". Preparing the dataset means that *dbExpress* and the server allocate resources for the statement and its parameters. If *CommandType* is *ctTable*, this is when the dataset generates its SELECT query. Any parameters that are not bound by the server are folded into a query at this point.

Unidirectional datasets are automatically prepared when you set *Active* to *True* or call the *Open* method. When you close the dataset, the resources allocated for executing the statement are freed. If you intend to execute the query or stored procedure more than once, you can improve performance by explicitly preparing the dataset before you open it the first time. To explicitly prepare a dataset, set its *Prepared* property to *True*.

```
CustQuery.Prepared := True;
```

When you explicitly prepare the dataset, the resources allocated for executing the statement are not freed until you set *Prepared* to *False*.

Set the *Prepared* property to *False* if you want to ensure that the dataset is re-prepared before it executes (for example, if you change a parameter value or the *SortFieldNames* property).

Fetching multiple datasets

Some stored procedures return multiple sets of records. The dataset only fetches the first set when you open it. In order to access the other sets of records, call the *NextRecordSet* method:

```
var
  DataSet2: TCustomSQLDataSet;
begin
  DataSet2 := SQLStoredProc1.NextRecordSet;
  ...
```

NextRecordSet returns a newly created *TCustomSQLDataSet* component that provides access to the next set of records. That is, the first time you call *NextRecordSet*, it returns a dataset for the second set of records. Calling *NextRecordSet* again returns a third dataset, and so on, until there are no more sets of records. When there are no additional datasets, *NextRecordSet* returns **nil**.

Executing commands that do not return records

You can use a unidirectional dataset even if the query or stored procedure it represents does not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language.

The SQL command you execute must be acceptable to the server you are using. Unidirectional datasets neither evaluate the SQL nor execute it. They merely pass the command to the server for execution.

Note If the command does not return any records, you do not need to use a unidirectional dataset at all, because there is no need for the dataset methods that provide access to a set of records. The SQL connection component that connects to the database server can be used directly to execute a command on the server. See "Sending commands to the server" on page 19-12 for details.

Specifying the command to execute

With unidirectional datasets, the way you specify the command to execute is the same whether the command results in a dataset or not. That is:

When using *TSQLDataSet*, use the *CommandType* and *CommandText* properties to specify the command:

- If *CommandType* is *ctQuery*, *CommandText* is the SQL statement to pass to the server.
- If *CommandType* is *ctStoredProc*, *CommandText* is the name of a stored procedure to execute.

When using *TSQLQuery*, use the *SQL* property to specify the SQL statement to pass to the server.

When using *TSQLStoredProc*, use the *StoredProcName* property to specify the name of the stored procedure to execute.

Just as you specify the command in the same way as when you are retrieving records, you work with query parameters or stored procedure parameters the same way as with queries and stored procedures that return records. See "Using parameters in queries" on page 18-4 and "Working with stored procedure parameters" on page 18-7 for details.

Executing the command

To execute a query or stored procedure that does not return any records, you do not use the *Active* property or the *Open* method. Instead, you must use

- The *ExecSQL* method if the dataset is an instance of *TSQLDataSet* or *TSQLQuery*.

  ```
  FixTicket.CommandText := 'DELETE FROM TrafficViolations WHERE (TicketID = 1099)';
  FixTicket.ExecSQL;
  ```

- The *ExecProc* method if the dataset is an instance of *TSQLStoredProc*.

  ```
  SQLStoredProc1.StoredProcName := 'MyCommandWithNoResults';
  SQLStoredProc1.ExecProc;
  ```

Tip If you are executing the query or stored procedure multiple times, it is a good idea to set the *Prepared* property to *True*.

Creating and modifying server metadata

Most of the commands that do not return data fall into two categories: those that you use to edit data (such as INSERT, DELETE, and UPDATE commands), and those that you use to create or modify entities on the server such as tables, indexes, and stored procedures.

If you don't want to use explicit SQL commands for editing, you can link your unidirectional dataset to a client dataset and let it handle all the generation of all SQL commands concerned with editing (see "Using a client dataset to buffer records" on page 14-9). In fact, this is the recommended approach because data-aware controls are designed to perform edits through a client dataset (or custom dataset that enables editing).

The only way your application can create or modify metadata on the server, however, is to send a command. Not all database drivers support the same SQL syntax. It is beyond the scope of this topic to describe the SQL syntax supported by each database type and the differences between the database types. For a comprehensive and up-to-date discussion of the SQL implementation for a given database system, see the documentation that comes with that system.

In general, use the CREATE TABLE statement to create tables in a database and CREATE INDEX to create new indexes for those tables. Where supported, use other CREATE statements for adding various metadata objects, such as CREATE DOMAIN, CREATE VIEW, CREATE SCHEMA, and CREATE PROCEDURE.

For each of the CREATE statements, there is a corresponding DROP statement to delete the metadata object. These statements include DROP TABLE, DROP VIEW, DROP DOMAIN, DROP SCHEMA, and DROP PROCEDURE.

To change the structure of a table, use the ALTER TABLE statement. ALTER TABLE has ADD and DROP clauses to create new elements in a table and to delete them. For example, use the ADD COLUMN clause to add a new column to the table and DROP CONSTRAINT to delete an existing constraint from the table.

For example, the following statement creates a stored procedure called GET_EMP_PROJ on an InterBase database:

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
  FOR SELECT PROJ_ID
  FROM EMPLOYEE_PROJECT
  WHERE EMP_NO = :EMP_NO
  INTO :PROJ_ID
  DO
    SUSPEND;
END
```

The following code uses a *TSQLDataSet* to create this stored procedure. Note the use of the *ParamCheck* property to prevent the dataset from confusing the parameters in the stored procedure definition (:EMP_NO and :PROJ_ID) with a parameter of the query that creates the stored procedure.

```
with SQLDataSet1 do
begin
  ParamCheck := False;
  CommandType := ctQuery;
  CommandText := 'CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT) ' +
      'RETURNS (PROJ_ID CHAR(5)) AS ' +
      'BEGIN ' +
        'FOR SELECT PROJ_ID FROM EMPLOYEE_PROJECT ' +
        'WHERE EMP_NO = :EMP_NO ' +
        'INTO :PROJ_ID ' +
          'DO SUSPEND; ' +
       END';
  ExecSQL;
end;
```

Setting up master/detail relationships

There are two ways to set up a master/detail relationship that uses a unidirectional dataset as the detail set. Which method you use depends on the type of unidirectional dataset you are using. Once you have set up such a relationship, the unidirectional dataset (the “many” in a one-to-many relationship) provides access only to those records that correspond to the current record on the master set (the “one” in the one-to-many relationship).

Setting up master/detail relationships with TSQLDataSet or TSQLQuery

To set up a master/detail relationship where the detail set is an instance of *TSQLDataSet* or *TSQLQuery*, you must specify a query that uses parameters. These parameters refer to current field values on the master dataset. Because the current field values on the master dataset change dynamically at runtime, you must rebind the detail set’s parameters every time the master record changes. Although you could write code to do this using an event handler, *TSQLDataSet* and *TSQLQuery* provide an easier mechanism using the *DataSource* property.

If parameter values for a parameterized query are not bound at design time or specified at runtime, *TSQLDataSet* and *TSQLQuery* attempt to supply values for them based on the *DataSource* property. *DataSource* identifies a different dataset that is searched for field names that match the names of unbound parameters. This search dataset can be any type of dataset, it need not be another unidirectional dataset. The search dataset must be created and populated before you create the detail dataset that uses it. If matches are found in the search dataset, the detail dataset binds the parameter values to the values of the fields in the current record pointed to by the data source.

To illustrate how this works, consider two tables: a customer table and an orders table. For every customer, the orders table contains a set of orders that the customer made. The Customer table includes an ID field that specifies a unique customer ID. The orders table includes a CustID field that specifies the ID of the customer who made an order.

The first step is to set up the Customer dataset:

1. Add a *TSQLDataSet* component to your application. Set its *CommandType* property to *ctTable* and its *CommandText* property to the name of the Customer table.
2. Add a *TDataSource* component named *CustomerSource*. Set its *DataSet* property to the dataset added in step 1. This data source now represents the Customer dataset.
3. Add another *TSQLDataSet* component and set its *CommandType* property to *ctQuery*. Set its *CommandText* property to

   ```
   SELECT CustID, OrderNo, SaleDate
   FROM Orders
   WHERE CustID = :ID
   ```

   Note that the name of the parameter is the same as the name of the field in the master (Customer) table.
4. Set the detail dataset's *DataSource* property to *CustomerSource*. Setting this property makes the detail dataset a linked query.

At runtime the *:ID* parameter in the SQL statement for the detail dataset is not assigned a value, so the dataset tries to match the parameter by name against a column in the dataset identified by *CustomersSource*. *CustomersSource* gets its data from the master dataset, which, in turn, derives its data from the Customer table. Because the Customer table contains a column called "ID," the value from the *ID* field in the current record of the master dataset is assigned to the *:ID* parameter for the detail dataset's SQL statement. The datasets are linked in a master-detail relationship. Each time the current record changes in the Customers dataset, the detail dataset's SELECT statement executes to retrieve all orders based on the current customer id.

Setting up master/detail relationships with TSQLTable

To set up a master/detail relationship where the detail set is an instance of *TSQLTable*, use the *MasterSource* and *MasterFields* properties.

The *MasterSource* property specifies a data source bound to the master table (like the *DataSource* property you use when the detail is a query). The *MasterFields* property lists the fields in the master table that correspond to the fields in the *TSQLTable* object's index.

Before you set the *MasterFields* property, first specify the index that starts with the corresponding fields. You can use either the *IndexName* or the *IndexFieldNames* property.

Once you have specified the index to use, use the *MasterFields* property to indicate the column(s) in the master dataset that correspond to the indexed fields in the (detail) SQL table. To link datasets based on multiple column names, use a semicolon delimited list:

```
SQLTable1.MasterFields := 'OrderNo;ItemNo';
```

Tip If you double click on the *MasterFields* property in the Object Inspector after you have assigned a *MasterSource*, the Field Link editor appears to help you select link fields in the master dataset and in *TSQLTable*.

Accessing schema information

You can populate a unidirectional dataset with information about what is available on the server instead of the results of a query or stored procedure. This information, called metadata, includes information about what tables and stored procedures are available on the server and information about these tables and stored procedures (such as the fields a table contains, the indexes that are defined, and the parameters a stored procedure uses).

To fetch metadata from the database server, you must first indicate what data you want to see, using the *SetSchemaInfo* method. *SetSchemaInfo* takes three parameters:

- The type of schema information (metadata) you want to fetch. This can be a list of tables (*stTables*), a list of system tables (*stSysTables*), a list of stored procedures (*stProcedures*), a list of fields in a table (*stColumns*), a list of indexes (*stIndexes*), or a list of parameters used by a stored procedure (*stProcedureParams*). Each type of information uses a different set of fields to describe the items in the list. For details on the structures of these datasets, see "The structure of metadata datasets" on page 18-16.
- If you are fetching information about fields, indexes, or stored procedure parameters, the name of the table or stored procedure to which they apply. If you are fetching any other type of schema information, this parameter is nil.
- A pattern that must be matched for every name returned. This pattern is an SQL pattern such as 'Cust%', which uses the wildcards '%' (to match a string of arbitrary characters of any length) and '_' (to match a single arbitrary character). To use a literal percent or underscore in a pattern, the character is doubled (%% or __). If you do not want to use a pattern, this parameter can be nil.

Note If you are fetching schema information about tables (*stTables*), the resulting schema information can describe ordinary tables, system tables, views, and/or synonyms, depending on the value of the SQL connection's *TableScope* property.

The following call requests a table listing all system tables (server tables that contain metadata):

```
SQLDataSet1.SetSchemaInfo(stSysTable, '', '');
```

When you open the dataset after this call to *SetSchemaInfo*, the resulting dataset has a record for each table, with columns giving the table name, type, schema name, and so on. If the server does not use system tables to store metadata (for example MySQL), when you open the dataset it contains no records.

The previous example used only the first parameter. Suppose, Instead, you want to obtain a list of input parameters for a stored procedure named 'MyProc'. Suppose, further, that the person who wrote that stored procedure named all parameters using a prefix to indicate whether they were input or output parameters ('inName', 'outValue' and so on). You would call *SetSchemaInfo* as follows:

```
SQLDataSet1.SetSchemaInfo(stProcedureParams, 'MyProc', 'in%');
```

The resulting dataset is a table of input parameters with columns to describe the properties of each parameter.

Note You can also fetch metadata from the database server into a list object rather than using a dataset. This alternate approach, which uses *TSQLConnection* rather than a dataset, provides less information and has no support for pattern matching strings. For details, see "Accessing server metadata" on page 19-10.

Fetching data after using the dataset for metadata

There are two ways to return to executing queries or stored procedures with the dataset after a call to *SetSchemaInfo*:

- Change the *CommandText* property, specifying the query, table, or stored procedure from which you want to fetch data.
- Call *SetSchemaInfo*, setting the first parameter to *stNoSchema*. In this case, the dataset reverts to fetching the data specified by the current value of *CommandText*.

The structure of metadata datasets

For each type of metadata you can access using *TSQLDataSet*, there is a predefined set of columns (fields) that are populated with information about the items of the requested type.

Information about tables

When you request information about tables (*stTables* or *stSysTables*), the resulting dataset includes a record for each table. It has the following columns:

Table 18.1 Columns in tables of metadata listing tables

Column Name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the table. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the table.
TABLE_NAME	ftString	The name of the table. This field determines the sort order of the dataset.
TABLE_TYPE	ftInteger	Identifies the type of table. It is a sum of one or more of the following values: 1: Table 2: View 4: System table 8: Synonym 16: Temporary table 32: Local table.

Information about stored procedures

When you request information about stored procedures (*stProcedures*), the resulting dataset includes a record for each stored procedure. It has following columns:

Table 18.2 Columns in tables of metadata listing stored procedures

Column Name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the stored procedure. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the stored procedure.
PROC_NAME	ftString	The name of the stored procedure. This field determines the sort order of the dataset.
PROC_TYPE	ftInteger	Identifies the type of stored procedure. It is a sum of one or more of the following values: 1: Procedure 2: Function 4: Package 8: System procedure
IN_PARAMS	ftSmallint	The number of input parameters
OUT_PARAMS	ftSmallint	The number of output parameters.

Information about fields

When you request information about the fields in a specified table (*stColumns*), the resulting dataset includes a record for each field. It includes the following columns:

Table 18.3 Columns in tables of metadata listing fields

Column Name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the table whose fields you listing. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the field.
TABLE_NAME	ftString	The name of the table that contains the fields.
COLUMN_NAME	ftString	The name of the field. This value determines the sort order of the dataset.
COLUMN_POSITION	ftSmallint	The position of the column in its table.
COLUMN_TYPE	ftInteger	Identifies the type of value in the field. It is a sum of one or more of the following: 1: Row ID 2: Row Version 4: Auto increment field 8: Field with a default value
COLUMN_DATATYPE	ftSmallint	The datatype of the column. This is one of the logical field type constants defined in sqllinks.pas.
COLUMN_TYPENAME	ftString	A string describing the datatype. This is the same information as contained in COLUMN_DATATYPE and COLUMN_SUBTYPE, but in a form used in some DDL statements.
COLUMN_SUBTYPE	ftSmallint	A subtype for the column’s datatype. This is one of the logical subtype constants defined in sqllinks.pas.
COLUMN_PRECISION	ftInteger	The size of the field type (number of characters in a string, bytes in a bytes field, significant digits in a BCD value, members of an ADT field, and so on)
COLUMN_SCALE	ftSmallint	The number of digits to the right of the decimal on BCD values, or descendants on ADT and array fields.
COLUMN_LENGTH	ftInteger	The number of bytes required to store field values
COLUMN_NULLABLE	ftSmallint	A Boolean that indicates whether the field can be left blank. (0 means the field requires a value)

Information about indexes

When you request information about the indexes on a table (stIndexes), the resulting dataset includes a record for each field in each record. (Multi-record indexes are described using multiple records) The dataset has the following columns:

Table 18.4 Columns in tables of metadata listing indexes

Column Name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the index. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the index.
TABLE_NAME	ftString	The name of the table for which the index is defined.
INDEX_NAME	ftString	The name of the index. This field determines the sort order of the dataset.
PKEY_NAME	ftString	Indicates the name of the primary key.
COLUMN_NAME	ftString	The name of the field (column) in the index.
COLUMN_POSITION	ftSmallint	The position of this field in the index.
INDEX_TYPE	ftSmallint	Identifies the type of index. It is a sum of one or more of the following values: 1: Non-unique 2: Unique 4: Primary key
SORT_ORDER	ftString	Indicates that the index is ascending (a) or descending (d) on this field.
FILTER	ftString	Describes a filter condition that limits the indexed records.

Information about stored procedure parameters

When you request information about the parameters of a stored procedure (*stProcedureParams*), the resulting dataset includes a record for each parameter. It has the following columns:

Table 18.5 Columns in tables of metadata listing parameters

Column Name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the stored procedure. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the stored procedure.
PROC_NAME	ftString	The name of the stored procedure that contains the parameter.
PARAM_NAME	ftString	The name of the parameter. This field determines the sort order of the dataset.

Table 18.5 Columns in tables of metadata listing parameters (continued)

Column Name	Field type	Contents
PARAM_TYPE	ftSmallint	Identifies the type of parameter. This is the same as a *TParam* object's *ParamType* property.
PARAM_DATATYPE	ftSmallint	The datatype of the parameter. This is one of the logical field type constants defined in sqllinks.pas.
PARAM_SUBTYPE	ftSmallint	A subtype for the parameter's datatype. This is one of the logical subtype constants defined in sqllinks.pas.
PARAM_TYPENAME	ftString	A string describing the datatype. This is the same information as contained in PARAM_DATATYPE and PARAM_SUBTYPE, but in a form used in some DDL statements.
PARAM_PRECISION	ftInteger	The maximum number of digits in floating-point values or bytes (for strings and Bytes fields).
PARAM_SCALE	ftSmallint	The number of digits to the right of the decimal on floating-point values.
PARAM_LENGTH	ftInteger	The number of bytes required to store parameter values.
PARAM_NULLABLE	ftSmallint	A Boolean that indicates whether the parameter can be left blank. (0 means the parameter requires a value)

Chapter

19

Connecting to databases

Kylix applications use *dbExpress* to connect to a database. *dbExpress* is a set of lightweight database drivers that provide fast access to SQL database servers. For each supported database, *dbExpress* provides a driver that adapts the server-specific software to a set of uniform *dbExpress* interfaces. When you deploy your application, you need only include a single shared object (the server-specific driver) with the application files you build.

Each *dbExpress* connection is encapsulated by a *TSQLConnection* component. *TSQLConnection* provides all the information necessary to establish a database connection using *dbExpress*, and is designed to work with unidirectional dataset components. A single SQL connection component can be shared by multiple unidirectional datasets, or the datasets can each use their own connection.

You may want to use a separate connection component for each dataset if the server does not support multiple statements per connection. Check whether the database server requires a separate connection for each dataset by reading the *MaxStmtsPerConn* property. By default, *TSQLConnection* generates connections as needed when the server limits the number of statements that can be executed over a connection. If you want to keep stricter track of the connections you are using, set the *AutoClone* property to *False*.

TSQLConnection lets you perform a number of tasks concerned with how you connect to or use the database server. These include

- Controlling connections
- Controlling server login
- Managing transactions
- Accessing server metadata
- Working with associated datasets
- Sending commands to the server
- Debugging database applications

Controlling connections

Before you can establish a connection to a database server, your application must provide certain key pieces of information that describe the desired server connection. Once you have identified the server and provided login details, *TSQLConnection* can open or close a connection to the server.

Describing the server connection

In order to describe a database connection in sufficient detail for *TSQLConnection* to open a connection, you must identify both the driver to use and a set of connection parameters that are passed to that driver.

Identifying the driver

The driver is identified by the *DriverName* property, which is the name of an installed *dbExpress* driver, such as INTERBASE, MYSQL, ORACLE, or DB2. The driver name is associated with two files

- The *dbExpress* driver, which is a shared object file with a name like libsqlib.so, libsqlmys.so, libsqlora.so, or libsqldb2.so.
- The Shared object file provided by the database vendor.

The relationship between these two shared object files and the database name is stored in a file called "dbxdrivers," which is updated when you install a *dbExpress* driver. Typically, you do not need to worry about these files because the SQL connection component looks them up in dbxdrivers when given the value of *DriverName*. When you set the *DriverName* property, *TSQLConnection* automatically sets the *LibraryName* and *VendorLib* properties to the names of the associated shared objects. Once *LibraryName* and *VendorLib* have been set (along with *GetDriverFunc*), your application does not need to rely on dbxdrivers. (That is, you do not need to deploy the dbxdrivers file with your application.)

Specifying connection parameters

The *Params* property is a string list that lists name/value pairs. Each pair has the form *Name=Value*, where *Name* is the name of the parameter, and *Value* is the value you want to assign.

The particular parameters you need depend on the database server you are using. However, one particular parameter, *Database*, is required for all servers. Its value depends on the server you are using. For example, with InterBase, *Database* is the name of the .gdb file, with Oracle, it is the entry in TNSName.ora, with DB2, it is the client-side node, while with MySQL, it is the database name that was assigned by a CREATE DATABASE command.

Other typical parameters include the *User_Name* (the name to use when logging in), *Password* (the password for *User_Name*), *HostName* (the machine name or IP address of where the server is located), and *TransIsolation* (the degree to which transactions you introduce are, by default, aware of changes made by other transactions). When you specify a driver name, the *Params* property is preloaded with all the parameters you need for that driver type, initialized to default values.

Because *Params* is a string list, at design time you can double-click on the *Params* property in the Object Inspector to edit the parameters using the String List editor. At runtime, use the *Params.Values* property to assign values to individual parameters.

Naming a connection description

Although you can always specify a connection using only the *DatabaseName* and *Params* properties, it can be more convenient to name a specific combination and then just identify the connection by name. *dbExpress* allows you to name database and parameter combinations, which are then saved in a file called "dbxconnections". The name of each combination is called a connection name.

Once you have defined the connection name, you can identify a database connection by simply setting the *ConnectionName* property to a valid connection name. Setting *ConnectionName* automatically sets the *DriverName* and *Params* properties. Once *ConnectionName* is set, you can edit the *Params* property to create temporary differences from the saved set of parameter values, but changing the *DriverName* property clears both *Params* and *ConnectionName*.

One advantage of using connection names arises when you develop your application using one database (for example Local InterBase), but deploy it for use with another (such as ORACLE). In that case, *DriverName* and *Params* will likely differ on the system where you deploy your application from the values you use during development. You can switch between the two connection descriptions easily by using two versions of the dbxconnections file. At design-time, your application loads the *DriverName* and *Params* from the design-time version of dbxconnections. Then, when you deploy your application, it loads these values from a separate version of dbxconnections that uses the "real" database. However, for this to work, you must instruct your connection component to reload the *DriverName* and *Params* properties at runtime. There are two ways to do this:

- Set the *LoadParamsOnConnect* property to *True*. This causes *TSQLConnection* to automatically set *DriverName* and *Params* to the values associated with *ConnectionName* in the dbxconnections file when the connection is opened.
- Call the *LoadParamsFromIniFile* method. This method sets *DriverName* and *Params* to the values associated with *ConnectionName* in the dbxconnections file. You might choose to use this method if you want to then override certain parameter values before opening the connection.

Using the Connection Editor

The relationships between connection names and their associated driver and connection parameters is stored in the dbxconnections file. You can create or modify these associations using the Connection Editor.

To display the Connection Editor, double-click on the *TSQLConnection* component. The Connection Editor appears, with a drop-down list containing all available drivers, a list of connection names for the currently selected driver, and a table listing the connection parameters for the currently selected connection name.

You can use this dialog to indicate the connection to use by selecting a driver and connection name, and editing any parameter values you want to temporarily override. Once you have chosen the configuration you want, click the Test Connection button to check that you have not made any mistakes.

You can use this dialog to indicate the connection to use by selecting a driver and connection name. Once you have chosen the configuration you want, click the Test Connection button to check that you have chosen a valid configuration.

In addition, you can use this dialog to edit the named connections in dbxconnections:

- Edit the parameter values in the parameter table to change the currently selected named connection. When you exit the dialog by clicking OK, the new parameter values are saved to the dbxconnections file.
- Click the Add Connection button to define a new named connection. A dialog appears where you specify the driver to use and the name of the new connection. Once the connection is named, edit the parameters to specify the connection you want and click the OK button to save the new connection to dbxconnections.
- Click the Delete Connection button to delete the currently selected named connection from dbxconnections.
- Click the Rename Connection button to change the name of the currently selected named connection. Note that any edits you have made to the parameters are saved with the new name when you click the OK button.

Opening and closing server connections

TSQLConnection generates events when it opens or closes a connection to the server. This lets you customize the behavior of your application in response to changes in the database connection.

Opening a connection

There are two ways to connect to a database server using *TSQLConnection*:

- Call the *Open* method.
- Set the *Connected* property to *True*.

Calling the *Open* method sets *Connected* to *True*.

When you set *Connected* to *True*, *TSQLConnection* first generates a *BeforeConnect* event, where you can perform any initialization. For example, you can use this event to alter any properties that specify the server to which it will then connect. If you are altering any parameter values, however, be sure that the *LoadParamsOnConnect* property is *False*, otherwise, after the event handler exits, *TSQLConnection* replaces all the connection parameters with the values associated with *ConnectionName* in the dbxconnections file.

After the *BeforeConnect* event, *TSQLConnection* may display a default login dialog, depending on how you choose to control server login. It then passes the user name and password to the driver, opening a connection.

Once the connection is open, *TSQLConnection* generates an *AfterConnect* event, where you can perform any tasks that require an open connection, such as fetching metadata from the server.

Once a connection is established, it is maintained as long as there is at least one active dataset using it. When there are no more active datasets, *TSQLConnection* may drop the connection, depending on the value of its *KeepConnection* property.

KeepConnection determines if your application maintains a connection to a database even when all datasets associated with that database are closed. If *True*, a connection is maintained. For connections to remote database servers, or for applications that frequently open and close datasets, setting *KeepConnection* to *True* reduces network traffic and speeds up your application. If *KeepConnection* is *False,* the connection is dropped when there are no active datasets using the database. If a dataset that uses the database is later opened, the connection must be reestablished and initialized.

Disconnecting from a database server

There are two ways to disconnect from a server using *TSQLConnection*:

- Set the *Connected* property to *False*.
- Call the *Close* method.

Calling *Close* sets *Connected* to *False*.

When *Connected* is set to *False*, *TSQLConnection* generates a *BeforeDisconnect* event, where you can perform any cleanup before the connection closes. For example, you can use this event to cache information about all open datasets before they are closed.

After the *BeforeConnect* event, *TSQLConnection* closes all open datasets and disconnects from the server.

Finally, *TSQLConnection* generates an *AfterDisconnect* event, where you can respond to the change in connection status, such as enabling a Connect button in your user interface.

Note Calling *Close* or setting *Connected* to *False* disconnects from a database server even if *KeepConnection* is *True*.

Controlling server login

Most remote database servers include security features to prohibit unauthorized access. Generally, the server requires a user name and password login before permitting database access.

At design time, if a server requires a login, a standard login dialog box prompts for a user name and password when you first attempt to connect to the database.

At runtime, there are several ways you can handle a server's request for a login:

- Let the default login dialog and processes handle the login. This is the default approach. Set the *LoginPrompt* property of the *TSQLConnection* object to *True* (the default) and add DBLogDlg to the **uses** clause of the unit that declares the connection component. Your application displays the standard login dialog box when the server requests a user name and password.

- Provide the values for User_Name and Password in the dbxconnections file. You can save these parameter values from the Connection Editor under a particular *ConnectionName*. Set *LoginPrompt* to *False* to prevent the default login dialog from appearing. Typically, you only want to use this approach to avoid the need to log in during development. It is a serious breach of security to leave an unencrypted password in the dbxconnections file, where it is available for anyone to read.
- Use the *Params* property to supply login information before the attempt to log in. You can assign values to the *User_Name* and *Password* parameters at design-time through the Object Inspector or programmatically at runtime. Set *LoginPrompt* to *False* to prevent the default login dialog from appearing. For example, the following code sets the user name and password in the *BeforeConnect* event handler, decrypting an encrypted password that is stored in the dbxconnections file:

```
procedure TForm1.SQLConnectionBeforeConnect(Sender: TObject);
begin
  with Sender as TSQLConnection do
  begin
    if LoginPrompt = False then
    begin
      Params.Values['User_Name'] := 'SYSDBA';
      Params.Values['Password'] := Decrypt(Params.Values['Password']);
    end;
  end;
end;
```

  Note that setting the values in the *Params* property at design-time or using hard-coded strings in code causes the values to be embedded in the application's executable file. While not as freely available as in the dbxconnections file, this still leaves them easy to find, compromising server security.
- Write an event handler for the *OnLogin* event. Set the *LoginPrompt* property to *True* and in the *OnLogin* event handler, set the login parameters. A copy of the *User_Name*, *Password*, and *Database* parameters is passed to the event handler in its *LoginParams* parameter. Assign values to these parameters using this string list, providing whichever values are needed. On exit, the values returned in *LoginParams* are used to form the connection. The following example assigns values for the *User_Name* and *Password* database parameters using a global variable (*UserName*) and a method that returns a password given a user name (*PasswordSearch*):

```
procedure TForm1.SQLConnection1Login(Database: TDatabase; LoginParams: TStrings);
begin
  LoginParams.Values['User_Name'] := UserName;
  LoginParams.Values['Password'] := PasswordSearch(UserName);
end;
```

  As with the other methods of providing login parameters, when writing an *OnLogin* event handler, avoid hard coding the password in your application code.

It should appear only as an encrypted value, an entry in a secure database your application uses to look up the value, or be dynamically obtained from the user.

Warning The *OnLogin* event does not occur unless the *LoginPrompt* property is *True*. Having a *LoginPrompt* value of *False* and providing login information only in an *OnLogin* event handler creates a situation where it is impossible to log in to the database: The default dialog does not appear and the *OnLogin* event handler never executes.

Managing transactions

A *transaction* is a group of actions that must all be carried out successfully on one or more tables in a database before they are *committed* (made permanent). If one of the actions in the group fails, then all actions are *rolled back* (undone). By using transactions, you ensure that the database is not left in an inconsistent state when a problem occurs completing one of the actions that make up the transaction.

For example, in a banking application, transferring funds from one account to another is an operation you would want to protect with a transaction. If, after decrementing the balance in one account, an error occurred incrementing the balance in the other, you want to roll back the transaction so that the database still reflects the correct total balance.

It is always possible to manage transactions by sending SQL commands directly to the database. Each database provides its own transaction management model, although some, such as MySQL, have no transaction support at all. For servers that support it, you may want to code your own transaction management directly, taking advantage of advanced transaction management capabilities on a particular database server, such as schema caching.

If you do not need to use any advanced transaction management capabilities, *TSQLConnection* provides a set of methods and properties you can use to manage transactions without explicitly sending any SQL commands. Using these properties and methods has the advantage that you do not need to customize your application for each type of database server you use, as long as the server supports transactions. (Trying to use transactions on a database that does not support them — such as MySQL— causes *TSQLConnection* to raise an exception.)

Warning When a client dataset edits the data accessed through a *TSQLConnection* component (either directly or when it is linked, through a provider, to a unidirectional dataset that uses the *TSQLConnection* component), the provider implicitly provides transaction support for any updates. Be careful that any transactions you explicitly start do not conflict with those generated by the provider.

Starting a transaction

Start a transaction by calling the *StartTransaction* method. *StartTransaction* takes a single parameter, a transaction descriptor that lets you manage multiple simultaneous transactions and specify the transaction isolation level on a per-transaction basis. (For more information on transaction levels, see “Specifying the transaction isolation level” on page 19-9.)

```
var
  TD: TTransactionDesc;
begin
  TD.TransactionID := 1;
  TD.IsolationLevel := xilREADCOMMITTED;
  SQLConnection1.StartTransaction(TD);
```

In order to manage multiple simultaneous transactions, set the *TransactionID* field of the transaction descriptor to a unique value. *TransactionID* can be any value you choose, as long as it is unique (does not conflict with any other transaction currently underway). Depending on the server, transactions started by *TSQLConnection* can be nested or they can be overlapped. Before you create multiple simultaneous transactions, be sure they are supported by the database server.

By default, when you start a transaction, all subsequent statements that read from or write to the database occur in the context of that transaction, until the transaction is explicitly terminated or, in the case of overlapped transactions, until another transaction starts. Each statement is considered part of a group. Changes must be successfully committed to the database, or every change made in the group must be undone.

With overlapped transactions, a first transaction becomes inactive when the second transaction starts, although you can postpone committing or rolling back the first transaction until later. However, if you are using an InterBase database, you can identify each dataset in your application with a particular active transaction, by setting its *TransactionLevel* property. That is, after starting a second transaction, you can continue to work with both transactions simultaneously, simply by associating a dataset with the transaction you want.

You can determine whether a transaction is in process by checking the *InTransaction* property.

Ending a transaction

Ideally, a transaction should only last as long as necessary. The longer a transaction is active, the more simultaneous users that access the database, and the more concurrent, simultaneous transactions that start and end during the lifetime of your transaction, the greater the likelihood that your transaction will conflict with another when you attempt to commit any changes.

Ending a successful transaction

When the actions that make up the transaction have all succeeded, you can make the database changes permanent by using the *Commit* method. *Commit* takes a single parameter: the transaction descriptor you supplied to the *StartTransaction* method:

```
SQLConnection1.Commit(TD);
```

Note If you are working with nested transactions, it is possible to commit a secondary (nested) transaction, only to have it rolled back later when the primary transaction is rolled back.

Commit is usually attempted in a **try...except** statement. That way, if the transaction cannot commit successfully, you can use the **except** block to handle the error and retry the operation or to roll back the transaction.

Ending an unsuccessful transaction

If an error occurs when making the changes that are part of the transaction or when trying to commit the transaction, you will want to discard all changes that make up the transaction. To discard these changes, use the *Rollback* method. *Rollback* takes a single parameter: the transaction descriptor you supplied to the *StartTransaction* method:

```
SQLConnection1.Rollback(TD);
```

Rollback usually occurs in

- Exception handling code when you can't recover from a database error.
- Button or menu event code, such as when a user clicks a Cancel button.

Specifying the transaction isolation level

The transaction descriptor you provide when you start a transaction includes an *IsolationLevel* field, which allows you to control how the transaction interacts with other simultaneous transactions when they work with the same tables. In particular, it affects how much a transaction "sees" of other transactions' changes to a table.

Each server type supports a different set of possible values for *IsolationLevel*, and some do not support different isolation levels at all. If the server supports different isolation levels, the set of allowable values includes two or more of the following, depending on the server:

- *DirtyRead*: When *TransIsolation* is *DirtyRead*, your transaction sees all changes made by other transactions, even if they have not been committed. Uncommitted changes are not permanent, and might be rolled back at any time. This value provides the least isolation, and is not available for many database servers (such as Oracle or InterBase).
- *ReadCommitted*: When *TransIsolation* is *ReadCommitted*, only committed changes made by other transactions are visible. Although this setting protects your transaction from seeing uncommitted changes that may be rolled back, you may still receive an inconsistent view of the database state if another transaction is committed while you are in the process of reading.
- *RepeatableRead*: When *TransIsolation* is *RepeatableRead*, your transaction is guaranteed to see a consistent state of the database data. Your transaction sees a single snapshot of the data. It cannot see any subsequent changes to data by other simultaneous transactions, even if they are committed. This isolation level guarantees that once your transaction reads a record, its view of that record will not change. At this level your transaction is most isolated from changes made by other transactions.

Note In addition, the transaction descriptor allows you to specify a custom isolation level that is defined by the database driver. None of the dbExpress drivers provided with Kylix support custom isolation levels.

For a detailed description of how each isolation level is implemented, see your server documentation.

Accessing server metadata

Once you have an open connection, you can use *TSQLConnection* to obtain information about the entities available on the database server. This information, called metadata, includes information about what tables and stored procedures are available on the server and information about these tables and stored procedures (such as the fields a table contains, the indexes that are defined, and the parameters a stored procedure uses).

TSQLConnection has a number of methods you can call to fill a list with this metadata:

- *GetTableNames* fills a *TStrings* descendant (such a *TStringList*) with the names of all available tables on the server. Which tables are returned depends on two things: the *TableScope* property, and a boolean parameter that indicates whether you are only interested in system tables. If you only want system tables, you can use the *SystemTables* parameter, and the list will contain only system tables, regardless of the value of *TableScope*. (Note that not all servers use system tables to store metadata, so asking for system tables may result in an empty list.) If you do not request only system tables, *GetTableNames* returns the names of any tables that match the types specified by the *TableScope* property. These can include system tables, data tables, views, and synonyms.
- *GetFieldNames* fills a *TStrings* descendant with the names of all fields (columns) in a specified table.
- *GetIndexNames* fills a *TStrings* descendant with the names of all indexes defined on a specified table.
- *GetProcedureNames* fills a *TStrings* descendant with the names of all available stored procedures (if any).
- *GetProcedureParams* fills a *TList* object with pointers to parameter description records, where each record describes a parameter of a specified stored procedure, including its name, index, parameter type, field type, and so on. You can convert these parameter descriptions to the more familiar *TParams* object by calling the global *LoadParamListItems* procedure. Because *GetProcedureParams* dynamically allocates the individual records, your application must free them when it is finished with the information.

Note You can also access server metadata using *TSQLDataSet*. *TSQLDataSet* provides more detailed information and greater control over the information returned (for example, you can provide a pattern mask to limit the items returned. For more information on how to access server metadata with *TSQLDataSet*, see “Accessing schema information” on page 18-15.

Working with associated datasets

TSQLConnection maintains a list of all active datasets that use it to connect to a database. It uses this list, for example, to close all of the datasets when it closes the database connection.

You can use this list as well, to perform actions on all the datasets that use a specific *TSQLConnection* to connect to a particular database.

Closing datasets without disconnecting from the server

There may be times when you want to close all datasets without disconnecting from the database server. To close all open datasets without disconnecting from a server, follow these steps:

1. Set the *TSQLConnection* component's *KeepConnection* property to *True*.
2. Call the *TSQLConnection* component's *CloseDataSets* method.

Iterating through the associated datasets

To perform any actions (other than closing them all) on all the datasets that use a *TSQLConnection* instance, use the *DataSets* and *DataSetCount* properties. *DataSets* is an indexed array of all active datasets that are linked to the SQL connection component via their *SQLConnection* property. *DataSetCount* is the number of datasets in this array.

Note When using an SQL client dataset (*TSQLClientDataSet*), the connection component's *DataSets* property does not list the client dataset itself. Rather, it lists the internal unidirectional dataset that the client dataset uses to access the data.

DataSets lists only the active (open) datasets. If a dataset is closed, it does not appear in the list.

You can use *DataSets* with *DataSetCount* to cycle through all currently active datasets in code. For example, the following code cycles through all active datasets and disables any controls that use the data they provide:

```
var
  I: Integer;
begin
  with SQLConnection1 do
  begin
    for I := 0 to DataSetCount - 1 do
      DataSets[I].DisableControls;
  end;
end;
```

Sending commands to the server

Simple Data Definition Language (DDL) SQL statements such as CREATE INDEX, ALTER TABLE, and DROP DOMAIN statements can be executed directly from a *TSQLConnection* component using its *Execute* method. These statements do not return result sets and only operate on or create a database's metadata. The *Execute* method can also be used to execute Data Manipulation Language (DML) SQL statements, such as INSERT, DELETE, and UPDATE statements.

Execute takes three parameters, a string that specifies a single SQL statement that you want to execute, a *TParams* object that supplies any parameter values for that statement, and a pointer that can receive a dynamically created *TCustomSQLDataSet* object that is populated with the result set from executing the statement. Although you can use this third parameter to receive records that are returned as a result of executing the statement, the recommended approach is to use an SQL dataset instead. If you do use the third parameter, your application is responsible for freeing the dataset returned in the third parameter.

Note *Execute* can only execute one SQL statement at a time. It is not possible to execute multiple SQL statements with a single call to *Execute*, as you can with SQL scripting utilities. To execute more than one statement, call *Execute* repeatedly, each time with new parameters.

It is relatively easy to execute a statement that does not include any parameters. For example, the following code executes a CREATE TABLE statement (DDL) without any parameters:

```
procedure TForm1.CreateTableButtonClick(Sender: TObject);
var
  SQLstmt: String;
begin
  SQLConnection1.Connected := True;
  SQLstmt := 'CREATE TABLE NewCusts ' +
    '( ' +
    '  CustNo INTEGER, ' +
    '  Company CHAR(40), ' +
    '  State CHAR(2), ' +
    '  PRIMARY KEY (CustNo) ' +
    ')';
  SQLConnection1.Execute(SQLstmt, nil, nil);
end;
```

To use parameters, you must create a *TParams* object. For each parameter value, use the *TParams.CreateParam* method to add a *TParam* object. Then use properties of *TParam* to describe the parameter and set its value.

This process is illustrated in the following example, which executes an INSERT statement. The INSERT statement includes a single parameter named :*StateParam*. A *TParams* object (called *stmtParams*) is created to supply a value of "CA" for that parameter.

```
procedure TForm1.INSERT_WithParamsButtonClick(Sender: TObject);
var
  SQLstmt: String;
```

```
    stmtParams: TParams;
  begin
    stmtParams := TParams.Create;
    try
      SQLConnection1.Connected := True;
      stmtParams.CreateParam(ftString, 'StateParam', ptInput);
      stmtParams.ParamByName('StateParam').AsString := 'CA';
      SQLstmt := 'INSERT INTO "Custom.db" '+
        '(CustNo, Company, State) ' +
        'VALUES (7777, "Robin Dabank Consulting", :StateParam)';
      SQLConnection1.Execute(SQLstmt, stmtParams, nil);
    finally
      stmtParams.Free;
    end;
  end;
```

If the SQL statement includes a parameter but you do not supply a *TParam* object to supply a value for it, the SQL statement may cause an error when executed (this depends on the particular database back-end used). If a *TParam* object is provided but there is no corresponding parameter in the SQL statement, an exception is raised when the application attempts to use the *TParam*.

Debugging database applications

While you are debugging your database application, it may prove useful to monitor the SQL messages that are sent to and from the database server through your connection component, including those that are generated automatically for you (for example by a provider component or by the *dbExpress* driver).

Using TSQLMonitor to monitor SQL commands

TSQLConnection uses a companion component, *TSQLMonitor*, to intercept these messages and save them in a string list. To use *TSQLMonitor*,

1 Add a *TSQLMonitor* component to the form or data module containing the *TSQLConnection* component whose SQL commands you want to monitor.

2 Set its *SQLConnection* property to the *TSQLConnection* component.

3 Set the SQL monitor's *Active* property to *True*.

As SQL commands are sent to the server, the SQL monitor's *TraceList* property is automatically updated to list all the SQL commands that are intercepted.

You can save this list to a file by specifying a value for the *FileName* property and then setting the *AutoSave* property to *True*. *AutoSave* causes the SQL monitor to save the contents of the *TraceList* property to a file every time is logs a new message.

If you do not want the overhead of saving a file every time a message is logged, you can use the *OnLogTrace* event handler to only save files after a number of messages have been logged. For example, the following event handler saves the contents of

TraceList every 10th message, clearing the log after saving it so that the list never gets too long:

```
procedure TForm1.SQLMonitor1LogTrace(Sender: TObject; CBInfo: Pointer);
var
  LogFileName: string;
begin
  with Sender as TSQLMonitor do
  begin
    if TraceCount = 10 then
    begin
      LogFileName := 'c:\log' + IntToStr(Tag) + '.txt';
      Tag := Tag + 1; {ensure next log file has a different name }
      SaveToFile(LogFileName);
      TraceList.Clear; { clear list }
    end;
  end;
end;
```

Note If you were to use the previous event handler, you would also want to save any partial list (fewer than 10 entries) when the application shuts down.

Using a callback to monitor SQL commands

Instead of using *TSQLMonitor*, you can customize the way your application traces SQL commands by using the SQL connection component's *SetTraceCallbackEvent* method. *SetTraceCallbackEvent* takes two parameters: a callback of type *TSQLCallbackEvent*, and a user-defined value that is passed to the callback function.

The callback function takes two parameters: *CallType* and *CBInfo*:

- *CallType* is reserved for future use.
- *CBInfo* is a pointer to a record that includes the category (the same as *CallType*), the text of the SQL command, and the user-defined value that is passed to the *SetTraceCallbackEvent* method.

The callback returns a value of type *CBRType*, typically *cbrUSEDEF*.

The *dbExpress* driver calls your callback every time the SQL connection component passes a command to the server or the server returns an error message.

Warning Do not call *SetTraceCallbackEvent* if the *TSQLConnection* object has an associated *TSQLMonitor* component. *TSQLMonitor* uses the callback mechanism to work, and *TSQLConnection* can only support one callback at a time.

Chapter

20

Using client datasets

Client datasets provide all the data access, editing, navigation, data constraint, and filtering support introduced by *TDataSet*. They cache all data in memory and manipulate the in-memory cache. The support for manipulating the data they store in memory is provided by a shared object file, midas.so, which must be deployed with any application that uses client datasets.

Kylix provides two types of client datasets:

- *TClientDataSet*, which is designed to work without a specific type of database connectivity support (such as *dbExpress)*.
- *TSQLClientDataSet*, which uses *dbExpress* to fetch the data for the in-memory cache.

Both types of client dataset add to the properties and methods inherited from *TDataSet* to provide an expanded set of features for working with data. They differ primarily in how they obtain the data that is cached in memory.

Both types of client dataset support the following mechanisms for reading data and writing updates:

- Reading from and writing to a file accessed directly from the client dataset. When using only this mechanism, an application should use *TClientDataSet*, which has less overhead. However, when using the briefcase model, this mechanism is equally appropriate for both types of client dataset. Properties and methods supporting this mechanism are described in "Using a client dataset with file-based data" on page 20-37.
- Reading from another dataset. Client datasets provide a variety of mechanisms for copying data from other datasets. These are described in "Copying data from another dataset" on page 20-23.

In addition, *TClientDataSet* can also fetch data from a provider component. Provider components link *TClientDataSet* to other datasets. The provider forwards data from the source dataset to the client dataset, and sends edits from the client dataset back to the source dataset. Optionally, the provider generates an SQL command to apply updates back to the server, which the source dataset forwards to the server. Properties and methods for working with a provider are described in “Using a client dataset with a provider” on page 20-24.

TSQLClientDataSet can’t be linked to an external provider. Instead, it fetches data from a database server. The client dataset still caches updates in its in-memory cache, and includes a method to apply those cached updates back to the database server. Properties and methods for using *TSQLClientDataSet* to connect to a database server are described in “Using an SQL client dataset” on page 20-34.

Working with data using a client dataset

Like any dataset, you can use client datasets to supply the data for data-aware controls using a data source component. See Chapter 15, “Using data controls”for information on how to display database information in data-aware controls.

As descendants of *TDataSet*, client datasets inherit the power and usefulness of the properties, methods, and events defined for all dataset components. For a complete introduction to this generic dataset behavior, see Chapter 16, “Understanding datasets.”

Client datasets differ from other datasets in that they hold all their data in memory. Because of this, their support for common database functions can involve additional capabilities or considerations.

Navigating data in client datasets

If an application uses standard data-aware controls, then a user can navigate through a client dataset’s records using the built-in behavior of those controls. You can also navigate programmatically through records using standard dataset methods such as *First*, *Last*, *Next*, and *Prior*. For more information about these methods, see “Navigating datasets” on page 16-8.

Also inherited from *TDataSet* are the *Locate* and *Lookup* methods, which search for a particular record based on the values of specified fields. These methods are described in “Searching datasets” on page 16-14.

Client datasets implement the standard bookmark capabilities introduced in *TDataSet* for marking and navigating to specific records. For more information about bookmarking, see “Marking and returning to records” on page 16-12.

In addition to these methods introduced by *TDataSet*, client datasets introduce a number of additional navigation methods that take advantage of the way they store and index in-memory records.

For example, instead of using bookmarks, with the overhead of allocating and freeing the memory to store them, client datasets can position the cursor at any specific record using the *RecNo* property. While other datasets may use *RecNo* to determine the record number of the current record, client datasets can also set *RecNo* to a particular record number to make that record the current one.

However, the most powerful method introduced by *TClientDataSet* is the *Goto* and *Find* search methods that search for a record based on indexed fields. By explicitly using indexes that you define for the client dataset, client datasets can improve over the searching performance provided by the *Locate* and *Lookup* methods.

The following table summarizes the six related *Goto* and *Find* methods an application can use to search for a record:

Table 20.1 Index-based search methods

Method	Purpose
EditKey	Preserves the current contents of the search key buffer and puts the dataset into *dsSetKey* state so your application can modify existing search criteria prior to executing a search.
FindKey	Combines the *SetKey* and *GotoKey* methods in a single method.
FindNearest	Combines the *SetKey* and *GotoNearest* methods in a single method.
GotoKey	Searches for the first record in a dataset that exactly matches the search criteria, and moves the cursor to that record if one is found.
GotoNearest	Searches on string-based fields for the closest match to a record based on partial key values, and moves the cursor to that record.
SetKey	Clears the search key buffer and puts the dataset into *dsSetKey* state so your application can specify new search criteria prior to executing a search.

GotoKey and *FindKey* are boolean functions that, if successful, move the cursor to a matching record and return *True*. If the search is unsuccessful, the cursor is not moved, and these functions return *False*.

GotoNearest and *FindNearest* always reposition the cursor either on the first exact match found or, if no match is found, on the first record that is greater than the specified search criteria.

Specifying the index to use for searching

Before using the *Goto* and *Find* search methods, you must define the index, or key, that is used to speed the search. If the index has already been created for your client dataset, you must make that index current using the *IndexName* property. For example, if a client dataset has an index named “CityIndex” on which you want to search for a value, you must set the *IndexName* property to “CityIndex”:

```
ClientDataSet1.Close;
ClientDataSet1.IndexName := 'CityIndex';
ClientDataSet1.Open;
ClientDataSet1.SetKey;
ClientDataSet1['City'] := Edit1.Text;
ClientDataSet1.GotoNearest;
```

Instead of specifying an index name, you can list fields to use as a key in the *IndexFieldNames* property.

For more information on creating and using indexes in client datasets, see “Sorting and indexing” on page 20-16.

Executing a search with Goto methods

To execute a search using *Goto* methods, follow these general steps:

1. Specify the index to use for the search (as described above).
2. Open the client dataset.
3. Put the dataset in *dsSetKey* state with *SetKey*.
4. Specify the value(s) to search on in the *Fields* property. *Fields* is a *TFields* object, which maintains an indexed list of field components you can access by specifying ordinal numbers corresponding to columns. The first column number in a dataset is 0.
5. Search for and move to the first matching record found with *GotoKey* or *GotoNearest*.

For example, the following code, attached to a button's *OnClick* event, moves to the first record where the first field in the index has a value that exactly matches the text in an edit box:

```
procedure TSearchDemo.SearchExactClick(Sender: TObject);
begin
  ClientDataSet1.SetKey;
  ClientDataSet1.Fields[0].AsString := Edit1.Text;
  if not ClientDataSet1.GotoKey then
    ShowMessage('Record not found');
end;
```

GotoNearest is similar. It searches for the nearest match to a partial field value. It can be used only for string fields. For example,

```
ClientDataSet1.SetKey;
ClientDataSet1.Fields[0].AsString := 'Sm';
ClientDataSet1.GotoNearest;
```

If a record exists with "Sm" as the first two characters of the first indexed field's value, the cursor is positioned on that record. Otherwise, the position of the cursor does not change and *GotoNearest* returns *False*.

Executing a search with Find methods

The Find methods do the same thing as the *Goto* methods, except that you do not need to explicitly put the dataset in *dsSetKey* state to specify the key field values on which to search. To execute a search using *Find* methods, follow these general steps:

1. Specify the index to use for the search (as described above).
2. Open the client dataset.
3. Search for and move to the first or nearest record with *FindKey* or *FindNearest*. Both methods take a single parameter, a comma-delimited list of field values, where each value corresponds to an indexed column in the underlying table.

Note *FindNearest* can only be used for string fields.

Specifying the current record after a successful search

By default, a successful search positions the cursor on the first record that matches the search criteria. If you prefer, you can set the *KeyExclusive* property to *True* to position the cursor on the next record after the first matching record.

By default, *KeyExclusive* is *False*, meaning that successful searches position the cursor on the first matching record.

Searching on partial keys

If a client dataset has more than one key column, and you want to search for values in a subset of that key, set *KeyFieldCount* to the number of columns on which you are searching. For example, if the client dataset's current index has three columns, and you want to search for values using just the first column, set *KeyFieldCount* to 1.

For client datasets with multiple-column keys, you can search only for values in contiguous columns, beginning with the first. For example, for a three-column key you can search for values in the first column, the first and second, or the first, second, and third, but not just the first and third.

Repeating or extending a search

Each time you call *SetKey* or *FindKey* it clears any previous values in the *Fields* property. If you want to repeat a search using previously set fields, or you want to add to the fields used in a search, call *EditKey* in place of *SetKey* and *FindKey*. For example, suppose you have already executed a search based on the City field of the "CityIndex" index. Suppose further that "CityIndex" includes both the *City* and *Company* fields. To find a record with a specified company name in a specified city, use the following code:

```
ClientDataSet1.KeyFieldCount := 2;
ClientDataSet1.EditKey;
ClientDataSet1['Company'] := Edit2.Text;
ClientDataSet1.GotoNearest;
```

Limiting what records appear

To restrict users to a subset of available data on a temporary basis, applications can use ranges and filters. When you apply a range or a filter, the client dataset does not display all the data in its in-memory cache. Instead, it only displays the data that meets the range or filter conditions.

Filters are introduced by *TDataSet* class, and so potentially apply to custom datasets as well as client datasets. (They are not implemented for unidirectional datasets.) For more information about using filters, see "Displaying and editing a subset of data using filters" on page 16-15.

Ranges only apply to *TClientDataSet* components. Despite their similarities, ranges and filters have different uses. The following topics discuss the differences between ranges and filters and how to use ranges.

Understanding the differences between ranges and filters

Both ranges and filters restrict visible records to a subset of all available records, but the way they do so differs. A range is a set of contiguously indexed records that fall between specified boundary values. For example, in an employee database indexed on last name, you might apply a range to display all employees whose last names are greater than "Jones" and less than "Smith". Because ranges depend on indexes, you must set the current index to one that can be used to define the range. As with specifying an index to use for locating records, you can assign the index on which to define a range using either the *IndexName* or the *IndexFieldNames* property.

A filter, on the other hand, is any set of records that share specified data points, regardless of indexing. For example, you might filter an employee database to display all employees who live in California and who have worked for the company for five or more years. While filters can make use of indexes if they apply, filters are not dependent on them. Filters are applied record-by-record as an application scrolls through a dataset.

In general, filters are more flexible than ranges. Ranges, however, can be more efficient when datasets are large and the records of interest to an application are already blocked in contiguously indexed groups. For very large datasets, it may be still more efficient to use the WHERE clause of a query to select data before it ever fills the client dataset's in-memory cache.

Specifying ranges

There are two mutually exclusive ways to specify a range for a client dataset:

- Specify the beginning and ending separately using *SetRangeStart* and *SetRangeEnd*.
- Specify both endpoints at once using *SetRange*.

Setting the beginning of a range

Call the *SetRangeStart* procedure to put the dataset into *dsSetKey* state and begin creating a list of starting values for the range. Once you call *SetRangeStart*, subsequent assignments to the *Fields* property are treated as starting index values to use when applying the range. Fields specified must apply to the current index.

For example, suppose your application uses a client dataset named *Customers*, linked to the CUSTOMER table, and that you have created persistent field components for each field in the *Customers* dataset. CUSTOMER is indexed on its first column (*CustNo*). A form in the application has two edit components named *StartVal* and *EndVal*, used to specify start and ending values for a range. The following code can be used to create and apply a range:

```
with Customers do
begin
  SetRangeStart;
  FieldByName('CustNo').AsString := StartVal.Text;
  SetRangeEnd;
  if (Length(EndVal.Text) > 0) then
    FieldByName('CustNo').AsString := EndVal.Text;
  ApplyRange;
end;
```

This code checks that the text entered in *EndVal* is not null before assigning any values to *Fields*. If the text entered for *StartVal* is null, then all records from the beginning of the dataset are included, since all values are greater than null. However, if the text entered for *EndVal* is null, then no records are included, since none are less than null.

For a multi-column index, you can specify a starting value for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. If you try to set a value for a field that is not in the index, the dataset raises an exception.

Tip To start at the beginning of the dataset, omit the call to *SetRangeStart*.

To finish specifying the start of a range, call *SetRangeEnd* or apply or cancel the range. For information about applying and canceling ranges, see “Applying or canceling a range” on page 20-9.

Setting the end of a range

Call the *SetRangeEnd* procedure to put the dataset into *dsSetKey* state and start creating a list of ending values for the range. Once you call *SetRangeEnd*, subsequent assignments to the *Fields* property are treated as ending index values to use when applying the range. Fields specified must apply to the current index.

Warning Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, Kylix assumes the ending value of the range is a null value. A range with null ending values is always empty.

The easiest way to assign ending values is to call the *FieldByName* method. For example,

```
with ClientDataSet1 do
begin
  SetRangeStart;
  FieldByName('LastName').AsString := Edit1.Text;
  SetRangeEnd;
  FieldByName('LastName').AsString := Edit2.Text;
  ApplyRange;
end;
```

As with specifying start of range values, if you try to set a value for a field that is not in the index, the dataset raises an exception.

To finish specifying the end of a range, apply or cancel the range. For information about applying and canceling ranges, see “Applying or canceling a range” on page 20-9.

Setting start- and end-range values

Instead of using separate calls to *SetRangeStart* and *SetRangeEnd* to specify range boundaries, you can call the *SetRange* procedure to put the dataset into *dsSetKey* state and set the starting and ending values for a range with a single call.

SetRange takes two constant array parameters: a set of starting values, and a set of ending values. For example, the following statements establish a range based on a two-column index:

```
SetRange([Edit1.Text, Edit2.Text], [Edit3.Text, Edit4.Text]);
```

For a multi-column index, you can specify starting and ending values for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. To omit a value for the first field in an index, and specify values for successive fields, pass a null value for the omitted field.

Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, the dataset assumes the ending value of the range is a null value. A range with null ending values is always empty because the starting range is greater than or equal to the ending range.

Specifying a range based on partial keys

If a key is composed of one or more string fields, the *SetRange* methods support partial keys. For example, if an index is based on the *LastName* and *FirstName* columns, the following range specifications are valid:

```
ClientDataSet1.SetRangeStart;
ClientDataSet1['LastName'] := 'Smith';
ClientDataSet1.SetRangeEnd;
ClientDataSet1['LastName'] := 'Zzzzzz';
ClientDataSet1.ApplyRange;
```

This code includes all records in a range where *LastName* is greater than or equal to “Smith.” The value specification could also be:

```
ClientDataSet1['LastName'] := 'Sm';
```

This statement includes records that have *LastName* greater than or equal to “Sm.”

Including or excluding records that match boundary values

By default, a range includes all records that are greater than or equal to the specified starting range, and less than or equal to the specified ending range. This behavior is controlled by the *KeyExclusive* property. *KeyExclusive* is *False* by default.

If you prefer, you can set the *KeyExclusive* property for a client dataset to *True* to exclude records equal to ending range. For example,

```
ClientDataSet1.KeyExclusive := True;
ClientDataSet1.SetRangeStart;
ClientDataSet1['LastName'] := 'Smith';
ClientDataSet1.SetRangeEnd;
ClientDataSet1['LastName'] := 'Tyler';
ClientDataSet1.ApplyRange;
```

This code includes all records in a range where *LastName* is greater than or equal to “Smith” and less than “Tyler”.

Modifying a range

Two functions enable you to modify the existing boundary conditions for a range: *EditRangeStart*, for changing the starting values for a range; and *EditRangeEnd*, for changing the ending values for the range.

The process for editing and applying a range involves these general steps:

1. Putting the dataset into *dsSetKey* state and modifying the starting index value for the range.
2. Modifying the ending index value for the range.
3. Applying the range to the dataset.

You can modify either the starting or ending values of the range, or you can modify both boundary conditions. If you modify the boundary conditions for a range that is currently applied to the dataset, the changes you make are not applied until you call *ApplyRange* again.

Editing the start of a range

Call the *EditRangeStart* procedure to put the dataset into *dsSetKey* state and begin modifying the current list of starting values for the range. Once you call *EditRangeStart*, subsequent assignments to the *Fields* property overwrite the current index values to use when applying the range.

Tip If you initially created a start range based on a partial key, you can use *EditRangeStart* to extend the starting value for a range. For more information about ranges based on partial keys, see "Specifying a range based on partial keys" on page 20-8.

Editing the end of a range

Call the *EditRangeEnd* procedure to put the dataset into *dsSetKey* state and start creating a list of ending values for the range. Once you call *EditRangeEnd*, subsequent assignments to the *Fields* property are treated as ending index values to use when applying the range.

Applying or canceling a range

When you call *SetRangeStart* or *EditRangeStart* to specify the start of a range, or *SetRangeEnd* or *EditRangeEnd* to specify the end of a range, the dataset enters the *dsSetKey* state. It stays in that state until you apply or cancel the range.

Applying a range

When you specify a range, the boundary conditions you define are not put into effect until you apply the range. To make a range take effect, call the *ApplyRange* procedure. *ApplyRange* immediately restricts a user's view of and access to data in the specified subset of the dataset.

Canceling a range

The *CancelRange* method ends application of a range and restores access to the full dataset. Even though canceling a range restores access to all records in the dataset, the boundary conditions for that range are still available so that you can reapply the range at a later time. Range boundaries are preserved until you provide new range boundaries or modify the existing boundaries. For example, the following code is valid:

```
⋮
ClientDataSet1.CancelRange;
⋮
{later on, use the same range again. No need to call SetRangeStart, etc.}
ClientDataSet1.ApplyRange;
⋮
```

Representing master/detail relationships

Client datasets can be linked into master/detail relationships. When you set up a master/detail relationship, you link two datasets so that all the records of one (the detail) always correspond to the single current record in the other (the master).

Client datasets support master/detail relationships in two very distinct ways:

- Make the client dataset the detail of another dataset by linking cursors. This process is described in “Making the client dataset a detail of another dataset” below.
- Make the client dataset the master in a master/detail relationship using nested detail tables. This process is described in “Using nested detail tables” on page 20-12.

Each of these approaches has its unique advantages. Linking cursors lets you create master/detail relationships where the master table is not a client dataset. With nested details both datasets must be client datasets, but they provide for more options in how to display the data and are better suited for applying edits back to the server.

Making the client dataset a detail of another dataset

A client dataset’s *MasterSource* and *MasterFields* properties can be used to establish one-to-many relationships between two datasets.

The *MasterSource* property is used to specify a data source from which the client dataset gets data from the master table. This data source can be linked to any type of dataset, it need not be another client dataset. For instance, you can link a client dataset to a unidirectional dataset, so that the client dataset tracks the events occurring in the unidirectional dataset, by specifying the unidirectional dataset’s data source component in this property.

The client dataset is linked to the master table based on its current index. Before you specify the fields in the master dataset that are tracked by the (detail) client dataset, first specify the index in the client dataset that starts with the corresponding fields. You can use either the *IndexName* or the *IndexFieldNames* property.

Once you have specified the index to use, use the *MasterFields* property to indicate the column(s) in the master dataset that correspond to the indexed fields in the (detail) client dataset. To link datasets based on multiple column names, use a semicolon delimited list:

```
ClientDataSet1.MasterFields := 'OrderNo;ItemNo';
```

To help create meaningful links between two datasets, you can use the Field Link designer. To use the Field Link designer, double click on the *MasterFields* property in the Object Inspector after you have assigned a *MasterSource* and an index.

The following steps create a simple form in which a user can step through customer records and display all orders for the current customer. The master dataset is called *CustomersTable*, and the detail dataset is *OrdersTable*.

1. Place a *TSQLConnection*, a *TSQLDataSet* component, a *TSQLClientDataSet*, and two *TDataSource* components in a data module.
2. Set the properties of the *TSQLConnection* component as follows:
   - *DriverName*: INTERBASE
   - *Params*:
     - *Database*: the full path name for employee.gdb. (This database should be installed when you install interbase.)
     - *UserName*: your user name.
     - *Password*: your password.
3. Set the properties of the *TSQLDataSet* component as follows:
   - *SQLConnection*: *SQLConnection1*
   - *CommandType*: *ctTable*
   - *CommandText*: CUSTOMER
   - *Name*: CustomersTable
4. Set the properties of the first *TDataSource* component as follows:
   - *Name*: CustSource
   - *DataSet*: CustomersTable
5. Set the properties of the *TSQLClientDataSet* component as follows:
   - *DBConnection*: *SQLConnection1*
   - *CommandText*: 'Select * from SALES'
   - *Name*: OrdersTable
   - *IndexFieldNames*: 'Cust_No'
   - *MasterSource*: CustSource.

   The last two properties (*IndexFieldNames* and *MasterSource*) link the CUSTOMER table (the master table) to the ORDERS table (the detail table) using the CUST_No field on the orders table.
6. Double-click the *MasterFields* property value box in the Object Inspector to invoke the Field Link Designer to set the following properties:
   - Select *Cust_No* in both the Detail Fields and Master Fields field lists.
   - Click the Add button to add this join condition. In the Joined Fields list, “CustNo -> CustNo” appears.
   - Choose OK to commit your selections and exit the Field Link Designer.

7 Set the properties of the second *TDataSource* component as follows:

- *Name*: OrdersSource
- *DataSet*: OrdersTable

8 Place a *TDBText*, a *TDBGrid*, and a *TButton* on a form.

9 Choose File | Use Unit to specify that the form should use the data module.

10 Set the *DataSource* property of the DB grid to "OrdersSource".

11 Set the following properties on the DB text control:

- *DataSource*: CustSource
- *DataField*: CUSTOMER.

12 Double-click on the button, and in its *OnClick* event handler type

```
CustomersTable.Next;
```

13 Set the *Active* properties of *CustomersTable* and *OrdersTable* to *True* to display data in the form.

14 Compile and run the application.

If you run the application now, you will see that the datasets are linked together, and that when you press the button to display a new customer name in the DB text control, you see only those records in the ORDERS table that belong to the current customer.

Using nested detail tables

There are two ways to set up master/detail relationships in client datasets using nested tables:

- Obtain records that contain nested details from a provider component. When a provider component represents the master table of a master/detail relationship, it automatically creates a nested dataset field to represent the details for each record. This method only applies to *TClientDataSet*, because you do can't set up a separate provider component to represent a master table when using *TSQLClientDataSet*.
- Define nested details using the Fields Editor. At design time, right click the client dataset and choose Fields Editor. Add a new persistent field to your client dataset by right-clicking and choosing Add Fields. Define your new field with type DataSet Field. In the Fields Editor, define the structure of your detail table.

Note To use nested detail sets, the *ObjectView* property of the client dataset must be *True*.

When your client dataset contains nested detail datasets, *TDBGrid* provides support for displaying the nested details in a popup window. For more information on how this works, see "Displaying dataset fields" on page 17-24.

Alternately, you can display and edit these datasets in data-aware controls by using a separate client dataset for the detail set. At design time, create persistent fields for the fields in your (master) client dataset, including a DataSet field for the nested detail set.

You can now create a client dataset to represent the nested detail set. Set this detail dataset's *DataSetField* property to the persistent dataSet field in the master dataset.

Using nested detail sets is necessary if you want to apply updates from master and detail tables to a database server. In file-based database applications, using nested detail sets lets you save the details with the master records in one file, rather than requiring you load two datasets separately, and then recreate the indexes to re-establish the master/detail relationship.

Constraining data values

Client datasets let you specify limits on the values a user can enter when editing the data. There are two types of constraints you can impose on user edits: field-level constraints and record-level constraints.

Each field component has two properties that you can use to specify constraints:

- The *DefaultExpression* property defines a default value that is assigned to the field if the user does not enter a value. Note that if the database server or source dataset also assigns a default expression for the field, the client dataset's version takes precedence because it is assigned before the update is returned to the provider.
- The *CustomConstraint* property lets you assign a constraint condition that must be met before a field value can be posted. Custom constraints are useful for validating data before it is sent to the server. For example, you may want to duplicate constraint conditions that are imposed by the database server. That way, user edits that would violate server constraints are enforced on the client side, and are never passed to the database server where they would be rejected. This means that fewer updates generate error conditions during the updating process. For more information about working with custom constraints on field components, see "Specifying constraints" on page 17-19.

At the record level, you can specify constraints using the client dataset's *Constraints* property. *Constraints* is a collection of *TCheckConstraint* objects, where each object represents a separate condition. Use the *CustomConstraint* property of a *TCheckConstraint* object to add your own constraints that are checked when you post records.

When fetching data from a database server (using *TSQLClientDataSet* or using *TClientDataSet* with provider component), there may be times when you want to turn off enforcement of data constraints, especially when the client dataset does not contain all of the records from the source dataset (or database server). For example, if a server constraint is based on the current maximum value in a field, but the client dataset fetches multiple packets of records, the current maximum value for a field in the client dataset may differ from the maximum value on the database server, and constraints may be invoked differently. In another case, if a client dataset applies a filter to records when constraints are enabled, the filter may interfere in unintended ways with constraint conditions. In each of these cases, an application may disable constraint-checking.

To disable constraints temporarily, call a client dataset's *DisableConstraints* method. Each time *DisableConstraints* is called, a reference count is incremented. While the reference count is greater than zero, constraints are not enforced on the client dataset.

To reenable constraints for the client dataset, call the dataset's *EnableConstraints* method. Each call to *EnableConstraints* decrements the reference count. When the reference count is zero, constraints are enabled again.

Tip Always call *DisableConstraints* and *EnableConstraints* in paired blocks to ensure that constraints are enabled when you intend them to be.

Making data read-only

TDataSet introduces the *CanModify* property so that applications can determine whether the data in a dataset can be edited. Applications can't change the *CanModify* property, because some *TDataSet* descendants, such as unidirectional datasets, introduce no built-in support for posting edits.

However, because client datasets represent in-memory data, your application can always control whether users can edit that data. (It is possible, however, that the application can't post updates to a database server if the server table is read-only). To prevent users from modifying data, set the *ReadOnly* property of the client dataset to *True*. Setting *ReadOnly* to *True* sets the *CanModify* property to *False.*

You do not need to close a client dataset to change its read-only status. An application can make a client dataset read-only or not on a temporary basis at any time merely by changing the current setting of the *ReadOnly* property.

Editing data

Client datasets represent their data as an in-memory data packet. This packet is the value of the client dataset's *Data* property. By default, however, edits are not stored in the *Data* property. Instead the insertions, deletions, and modifications (made by users or programmatically) are stored in an internal change log, represented by the *Delta* property. Using a change log serves two purposes:

- When working with a provider, the change log is required by the mechanism for applying updates to the server.
- In any application, the change log provides sophisticated support for undoing changes.

The *LogChanges* property lets you disable logging temporarily. When *LogChanges* is *True*, changes are recorded in the log. When *LogChanges* is *False*, changes are made directly to the *Data* property. You can disable the change log in file-based applications when you do not need the undo support.

Edits in the change log remain there until they are removed by the application. Applications remove edits when

- Undoing changes
- Saving changes

Note Saving the client dataset to a file does not remove edits from the change log. When you reload the dataset, the *Data* and *Delta* properties are the same as they were when the data was saved.

Undoing changes

Even though a record's original version remains unchanged in *Data*, each time a user edits a record, leaves it, and returns to it, the user sees the last changed version of the record. If a user or application edits a record a number of times, each changed version of the record is stored in the change log as a separate entry.

Storing each change to a record makes it possible to support multiple levels of undo operations should it be necessary to restore a record's previous state:

- To remove the last change to a record, call *UndoLastChange*. *UndoLastChange* takes a Boolean parameter, *FollowChange*, that indicates whether to reposition the cursor on the restored record (*True*), or to leave the cursor on the current record (*False*). If there are several changes to a single record, each call to *UndoLastChange* removes another layer of edits. *UndoLastChange* returns a Boolean value indicating success or failure to remove a change. If the removal occurs, *UndoLastChange* returns *False*. Use the *ChangeCount* property to determine whether there are any more changes to undo. *ChangeCount* indicates the number of changes stored in the change log.
- Instead of removing each layer of changes to a single record, you can remove them all at once. To remove all changes to a record, select the record, and call *RevertRecord*. *RevertRecord* removes any changes to the current record from the change log.
- At any point during edits, you can save the current state of the change log using the *SavePoint* property. Reading *SavePoint* returns a marker into the current position in the change log. Later, if you want to undo all changes that occurred since you read the save point, set *SavePoint* to the value you read previously. Your application can obtain values for multiple save points. However, once you back up the change log to a save point, the values of all save points that your application read after that one are invalid.
- You can abandon all changes recorded in the change log by calling *CancelUpdates*. *CancelUpdates* clears the change log, effectively discarding all edits to all records. Be careful when you call *CancelUpdates*. After you call *CancelUpdates*, you cannot recover any changes that were in the log.

Saving changes

Client datasets use different mechanisms for incorporating changes from the change log, depending on whether the client dataset stores its data in a file or represents data from a server. Whichever mechanism is used, the change log is automatically emptied when all updates have been incorporated.

File-based applications can simply merge the changes into the local cache represented by the *Data* property. They do not need to worry about resolving local edits with changes made by other users. To merge the change log into the *Data* property, call the *MergeChangeLog* method. "Merging changes into data" on page 20-40 describes this process.

You can't use *MergeChangeLog* if you are using the client dataset to represent server data. The information in the change log is required so that the updated records can be resolved with the data stored in the database (or source dataset). Instead, you call *ApplyUpdates*, which sends the changes to the database server (or source dataset) and updates the *Data* property only when the modifications have been successfully posted. See "Applying updates" on page 20-31 for more information about this process.

Sorting and indexing

Using indexes provides several benefits to your applications:

- They allow client datasets to locate data quickly.
- They let you apply ranges to limit the available records.
- They let your application to set up relationships between client datasets such as lookup tables or master/detail links.
- They specify the order in which records appear.

If a client dataset uses a provider (including the internal provider used by *TSQLClientDataSet*), it inherits a default index and sort order based on the data it receives from the provider. The default index is called DEFAULT_ORDER. You can use this ordering, but you cannot change or delete the index.

In addition to the default index, the client dataset maintains a second index, called CHANGEINDEX, on the changed records stored in the change log (*Delta* property). CHANGEINDEX orders all records in the client dataset as they would appear if the changes specified in *Delta* were applied. CHANGEINDEX is based on the ordering inherited from DEFAULT_ORDER. As with DEFAULT_ORDER, you cannot change or delete the CHANGEINDEX index.

You can use other existing indexes for a dataset, and you can create your own indexes. The following sections describe how to create and use indexes with client datasets.

Adding a new index

There are three ways to add indexes to a client dataset:

- To create a temporary index at runtime that sorts the records in the client dataset, you can use the *IndexFieldNames* property. Specify field names, separated by semicolons. Ordering of field names in the list determines their order in the index.

  This is the least powerful method of adding indexes. You can't specify a descending or case-insensitive index, and the resulting indexes do not support grouping. These indexes do not persist when you close the dataset, and are not saved when you save the client dataset to a file.

- To create an index at runtime that can be used for grouping, call *AddIndex*. *AddIndex* lets you specify the properties of the index, including

  - The name of the index. This can be used for switching indexes at runtime.

- The fields that make up the index. The index uses these fields to sort records and to locate records that have specific values on these fields.
- How the index sorts records. By default, indexes impose an ascending sort order (based on the machine's locale). This default sort order is case-sensitive. You can specify options to make the entire index case-insensitive or to sort in descending order. Alternately, you can provide a list of fields to be sorted case-insensitively and a list of fields to be sorted in descending order.
- The default level of grouping support for the index.

Indexes created with *AddIndex* do not persist when the client dataset is closed. (That is, they are lost when you reopen the client dataset). You can't call *AddIndex* when the dataset is closed. Indexes you add using *AddIndex* are not saved when you save the client dataset to a file.

- The third way to create an index is at the time the client dataset is created. Before creating the client dataset, specify the desired indexes using the *IndexDefs* property. The indexes are then created along with the underlying dataset when you call *CreateDataSet*. See "Creating a dataset using field and index definitions" on page 20-38 for details.

  As with *AddIndex*, indexes you create with the dataset support grouping, can sort in ascending order on some fields and descending order on others, and can be case insensitive on some fields and case sensitive on others. Indexes created this way always persist and are saved when you save the client dataset to a file.

Tip You can index and sort on internally calculated fields with client datasets.

Deleting and switching indexes

To remove an index you created for a client dataset, call *DeleteIndex* and specify the name of the index to remove. You cannot remove the DEFAULT_ORDER and CHANGEINDEX indexes.

To use a different index with a client dataset when more than one index is available, use the *IndexName* property to select the index to use. At design time, you can select from available indexes in *IndexName* property drop-down box in the Object Inspector.

Obtaining information about indexes

At runtime, your application can ask the client dataset for information about the available indexes.

The *GetIndexNames* method retrieves a list of available indexes for a table. *GetIndexNames* fills a string list with valid index names. For example, the following code fetches the list of indexes available for *ClientDatSet1*:

```
var
  IndexList: TList;
begin
:
IndexList := TStringList.Create;
ClientDataSet1.GetIndexNames(IndexList);
```

You can also examine a list of fields in the current index. To iterate through all the fields in the current index, use two properties:

- *IndexFieldCount* property, which indicates the number of columns in the index.
- *IndexFields* property, which lists the fields that comprise the index.

IndexFields is an array of *TField* components, one for each field in the index. The following code fragment illustrates how you can use *IndexFieldCount* and *IndexFields* to iterate through the fields in the current index:

```
var
  I: Integer;
  ListOfIndexFields: array[0 to 20} of string;
begin
with ClientDataSet1 do
  begin
  for I := 0 to IndexFieldCount - 1 do
     ListOfIndexFields[I] := IndexFields[I].FieldName;
  end;
end;
```

Using indexes to group data

When you use an index in your client dataset, it automatically imposes a sort order on the records. Because of this order, adjacent records usually contain duplicate values on the fields that make up the index. For example, consider the following fragment from an orders table that is indexed on the SalesRep and Customer fields:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

Because of the sort order, adjacent values in the SalesRep column are duplicated. Within the records for SalesRep 1, adjacent values in the Customer column are duplicated. That is, the data is grouped by SalesRep, and within the SalesRep group it is grouped by Customer. Each grouping has an associated level. In this case, the SalesRep group has level 1 (because it is not nested in any other groups) and the Customer group has level 2 (because it is nested in the group with level 1). Grouping level corresponds to the order of fields in the index.

Client datasets let you determine where the current record lies within any given grouping level. This allows your application to display records differently, depending on whether they are the first record in the group, in the middle of a group, or the last record in a group. For example, you might want to only display a field value if it is on the first record of the group, eliminating the duplicate values. To do this with the previous table results in the following:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
		2	50
	2	3	200
		6	75
2	1	1	10
	3	4	200

To determine where the current record falls within any group, use the *GetGroupState* method. *GetGroupState* takes an integer giving the level of the group and returns a value indicating where the current record falls the group (first record, last record, or neither).

When you create an index, you can specify the level of grouping it supports (up to the number of fields in the index). *GetGroupState* can't provide information about groups beyond that level, even if the index sorts records on additional fields.

Representing calculated values

As with any dataset, you can add calculated fields to your client dataset. These are fields whose values you calculate dynamically, usually based on the values of other fields in the same record. For more information about using calculated fields, see "Defining a calculated field" on page 17-7.

Client datasets, however, let you optimize when fields are calculated by using internally calculated fields. For more information on internally calculated fields, see "Using internally calculated fields in client datasets" below.

You can also tell client datasets to create calculated values that summarize the data in several records using maintained aggregates. For more information on maintained aggregates, see "Using maintained aggregates" on page 20-20.

Using internally calculated fields in client datasets

In other datasets, your application must compute the value of calculated fields every time the record changes or the user edits any fields in the current record. It does this in an *OnCalcFields* event handler.

While you can still do this, client datasets let you minimize the number of times calculated fields must be recomputed by saving calculated values in the client dataset's data. When calculated values are saved with the client dataset, they must still be recomputed when the user edits the current record, but your application need not recompute values every time the current record changes. To save calculated values in the client dataset's data, use internally calculated fields instead of calculated fields.

Internally calculated fields, just like calculated fields, are calculated in an *OnCalcFields* event handler. However, you can optimize your event handler by checking the *State* property of your client dataset. When *State* is *dsInternalCalc*, you must recompute internally calculated fields. When *State* is *dsCalcFields*, you need only recompute regular calculated fields.

To use internally calculated fields, you must define the fields as internally calculated before you create the client dataset. If you are creating the client dataset using persistent fields, define fields as internally calculated by selecting InternalCalc in the Fields editor. If you are creating the client dataset using field definitions, set the *InternalCalcField* property of the relevant field definition to *True*.

Note Other types of datasets use internally calculated fields. However, with other datasets, you do not calculate these values in an *OnCalcFields* event handler. Instead, they are computed automatically by the remote database server.

Using maintained aggregates

Client datasets provide support for summarizing data over groups of records. Because these summaries are automatically updated as you edit the data in the dataset, this summarized data is called a "maintained aggregate."

In their simplest form, maintained aggregates let you obtain information such as the sum of all values in a column of the client dataset. They are flexible enough, however, to support a variety of summary calculations and to provide subtotals over groups of records defined by a field in an index that supports grouping.

Specifying aggregates

To specify that you want to calculate summaries over the records in a client dataset, use the *Aggregates* property. *Aggregates* is a collection of aggregate specifications (*TAggregate*). You can add aggregate specifications to your client dataset using the Collection Editor at design time, or using the *Add* method of *Aggregates* at runtime. If you want to create field components for the aggregates, create persistent fields for the aggregated values in the Fields Editor.

Note When you create aggregated fields, the appropriate aggregate objects are added to the client dataset's *Aggregates* property automatically. Do not add them explicitly when creating aggregated persistent fields. For details on creating aggregated persistent fields, see "Defining an aggregate field" on page 17-10.

For each aggregate, the *Expression* property indicates the summary calculation it represents. *Expression* can contain a simple summary expression such as

```
Sum(Field1)
```

or a complex expression that combines information from several fields, such as

```
Sum(Qty * Price) - Sum(AmountPaid)
```

Aggregate expressions include one or more of the summary operators in Table 20.2

Table 20.2 Summary operators for maintained aggregates

Operator	Use
Sum	Totals the values for a numeric field or expression
Avg	Computes the average value for a numeric or date-time field or expression
Count	Specifies the number of non-blank values for a field or expression
Min	Indicates the minimum value for a string, numeric, or date-time field or expression
Max	Indicates the maximum value for a string, numeric, or date-time field or expression

The summary operators act on field values or on expressions built from field values using the same operators you use to create filters. (You can't nest summary operators, however.) You can create expressions by using operators on summarized values with other summarized values, or on summarized values and constants. However, you can't combine summarized values with field values, because such expressions are ambiguous (there is no indication of which record should supply the field value.) These rules are illustrated in the following expressions:

Sum(Qty * Price)	{**legal** -- summary of an expression on fields}
Max(Field1) - Max(Field2)	{**legal** -- expression on summaries}
Avg(DiscountRate) * 100	{**legal** -- expression of summary and constant}
Min(Sum(Field1))	{**illegal** -- nested summaries}
Count(Field1) - Field2	{**illegal** -- expression of summary and field}

Aggregating over groups of records

By default, maintained aggregates are calculated so that they summarize all the records in a client dataset. However, you can specify that you want to summarize over the records in a group instead. This allows you to provide intermediate summaries such as subtotals for groups of records that share a common field value.

Before you can specify a maintained aggregate over a group of records, you must use an index that supports the appropriate grouping. See "Using indexes to group data" on page 20-18 for information on grouping support.

Once you have an index that groups the data in the way you want it summarized, specify the *IndexName* and *GroupingLevel* properties of the aggregate to indicate what index it uses, and which group or subgroup on that index defines the records it summarizes.

For example, consider the following fragment from an orders table that is grouped by SalesRep and, within SalesRep, by Customer:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

The following code sets up a maintained aggregate that indicates the total amount for each sales representative:

```
Agg.Expression := 'Sum(Amount)';
Agg.IndexName := 'SalesCust';
Agg.GroupingLevel := 1;
Agg.AggregateName := 'Total for Rep';
```

To add an aggregate that summarizes for each customer within a given sales representative, create a maintained aggregate with level 2.

Maintained aggregates that summarize over a group of records are associated with a specific index. The *Aggregates* property can include aggregates that use different indexes. However, only the aggregates that summarize over the entire dataset and those that use the current index are valid. Changing the current index changes which aggregates are valid. To determine which aggregates are valid at any time, use the *ActiveAggs* property.

Obtaining aggregate values

To get the value of a maintained aggregate, call the *Value* method of the *TAggregate* object that represents the aggregate. *Value* returns the maintained aggregate for the group that contains the current record of the client dataset.

When you are summarizing over the entire client dataset, you can call *Value* at any time to obtain the maintained aggregate. However, when you are summarizing over grouped information, you must be careful to ensure that the current record is in the group whose summary you want. Because of this, it is a good idea to obtain aggregate values at clearly specified times, such as when you move to the first record of a group or when you move to the last record of a group. Use the *GetGroupState* method to determine where the current record falls within a group.

To display maintained aggregates in data-aware controls, use the Fields editor to create a persistent aggregate field component. When you specify an aggregate field in the Fields editor, the client dataset's *Aggregates* is automatically updated to include the appropriate aggregate specification. The *AggFields* property contains the new aggregated field component, and the *FindField* method returns it.

Adding application-specific information to the data

Application developers can add custom information to the client dataset's *Data* property. Because this information is bundled with the data packet, it is included when you save the data to a file or stream. It is copied when you copy the data to another dataset. Optionally, it can be included with the *Delta* property so that a provider can read this information when it receives updates from the client dataset.

To save application-specific information with the *Data* property, use the *SetOptionalParam* method. This method lets you store an OleVariant that contains the data under a specific name.

To retrieve this application-specific information, use the *GetOptionalParam* method, passing in the name that was used when the information was stored.

Copying data from another dataset

To copy the data from another dataset at design time, right click the dataset and choose Assign Local Data. A dialog appears listing all the datasets available in your project. Select the one from which you want to copy and choose OK. When you copy the source dataset, your client dataset is automatically activated.

To copy from another dataset at runtime, you can assign its data directly or, if the source is another client dataset, you can clone the cursor.

Assigning data directly

You can use the client dataset's *Data* property to assign data to a client dataset from another dataset. *Data* is a data packet in the form of an OleVariant. A data packet can come from another client dataset or from any other dataset by using a provider. Once a data packet is assigned to *Data*, its contents are displayed automatically in data-aware controls connected to the client dataset by a data source component.

When you open a *TSQLClientDataSet* object or a *TClientDataSet* object that is linked to a provider, data packets are automatically assigned to *Data*. See "Using an SQL client dataset" on page 20-34 for information on using *TSQLClientDataSet*. See "Using a client dataset with a provider" on page 20-24 for information on using *TClientDataSet* with a provider component.

When you are not using the data from the database or provider, you can copy the data from another client dataset as follows:

```
ClientDataSet1.Data := ClientDataSet2.Data;
```

Note When you copy the *Data* property of another client dataset, you copy the change log as well, but the copy does not reflect any filters or ranges that have been applied. To include filters or ranges, you must clone the source dataset's cursor instead.

You can copy data to *TClientDataSet* from a dataset that is not a client dataset by creating a dataset provider component, linking it to the source dataset, and then copying its data:

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := SourceDataSet;
ClientDataSet1.Data := TempProvider.Data;
TempProvider.Free;
```

Note When you assign directly to the *Data* property, the new data packet is not merged into the existing data. Instead, all previous data is replaced.

With any client dataset, you can use a provider component to merge changes from another client dataset. Create a dataset provider as in the previous example, but attach it to the destination dataset and instead of copying the data property, use the *ApplyUpdates* method:

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := SQLClientDataSet1;
TempProvider.ApplyUpdates(SourceDataSet.Delta, -1, ErrCount);
TempProvider.Free;
```

Note When you merge data this way, it is merged into the destination client dataset's in-memory cache. You must still call that client dataset's *ApplyUpdates* method to then apply those changes back to the database server.

Cloning a client dataset cursor

Client datasets use the *CloneCursor* method to let you work with a second view of the data at runtime. *CloneCursor* lets a second client dataset share the original client dataset's data. This is less expensive than copying all the original data, but, because the data is shared, the second client dataset can't modify the data without affecting the original client dataset.

CloneCursor takes three parameters: *Source* specifies the client dataset to clone. The last two parameters (*Reset* and *KeepSettings*) indicate whether to copy information other than the data. This information includes any filters, the current index, links to a master table (when the source dataset is a detail set), the *ReadOnly* property, and any links to a connection component or provider interface.

When *Reset* and *KeepSettings* are *False*, a cloned client dataset is opened, and the settings of the source client dataset are used to set the properties of the destination. When *Reset* is *True*, the destination dataset's properties are given the default values (no index or filters, no master table, *ReadOnly* is *False*, and no connection component or provider is specified). When *KeepSettings* is *True*, the destination dataset's properties are not changed.

Using a client dataset with a provider

When using a client dataset to represent data from a database server or another dataset, the client dataset uses a provider component. The provider component passes data from a source dataset to the client dataset. (When using *TSQLClientDataSet*, this source dataset is an internal query component for accessing the data; with *TClientDataSet*, it is a dataset component you add to a form or data module.) After editing the data in memory, the client dataset applies updates, through the provider, back to the source dataset or directly to a remote database server.

Because with *TSQLClientDataSet* components the provider is internal, it can't be accessed directly. However, you can still use the client dataset's properties and methods to affect the communication between the client dataset and its internal provider (and hence between the client dataset and the database server).

With *TClientDataSet*, the provider can reside in the same application as the client dataset, or it can be part of a separate application running on another system.

For more information about provider components, see Chapter 21, "Using provider components."

The following steps describe how to use a client dataset with a provider:

1 If you are using *TClientDataSet*, specify a data provider.
2 Optionally, Get parameters from the source dataset or pass parameters to the source dataset.
3 Depending on the client dataset and provider, you may specify the command to execute on the server.
4 Request data from the source dataset.
5 Update records.
6 Refresh records.

In addition, *TClientDataSet* lets you communicate with the provider using custom events.

Specifying a data provider

Before *TClientDataSet* can receive data from another dataset and apply updates to that dataset or its database server, it must be associated with a dataset provider. The way you associate *TClientDataSet* with a provider depends on whether the provider is in the same application as the client dataset or on a remote application server running on another system.

- If the provider is in the same application as the client dataset, you can associate it with a provider by choosing a provider from the drop-down list for the *ProviderName* property in the Object Inspector. This works as long as the provider has the same *Owner* as the client dataset. (The client dataset and the provider have the same *Owner* if they are placed in the same form or data module.) To use a local provider that has a different *Owner*, you must form the association at runtime using the client dataset's *SetProvider* method.
- If the provider is on a remote application server, you need to use both the *RemoteServer* and *ProviderName* properties. *RemoteServer* specifies the name of a connection component from which to get a list of providers. The connection component resides in the same data module as the client dataset. It establishes and maintains a connection to an application server, sometimes called a "data broker".

  At design time, after you specify *RemoteServer*, you can select a provider from the drop-down list for the *ProviderName* property in the Object Inspector. This list includes both local providers (in the same form or data module) and remote providers that can be accessed through the connection component.

Note The connection component that connects *TClientDataSet* to a provider on a remote application server must be purchased separately.

At runtime, you can switch among available providers (both local and remote) by setting *ProviderName* in code.

Getting parameters from the source dataset

There are two circumstances when a client dataset needs to obtain parameter values from its source dataset:

- The client needs to know the value of output parameters on a stored procedure.
- The client wants to initialize the input parameters of a query or stored procedure to the current values on the source dataset.

A client dataset stores parameter values in its *Params* property. These values are refreshed with any output parameters whenever the client dataset fetches data from the source dataset. However, there may be times when a *TClientDataSet* component in a client application needs output parameters when it is not fetching data.

To fetch output parameters when not fetching records, or to initialize input parameters, the client dataset can request parameter values from the source dataset by calling the *FetchParams* method. The parameters are returned in a data packet from the provider and assigned to the client dataset's *Params* property.

At design time, the *Params* property can be initialized by right-clicking the client dataset and choosing Fetch Params.

Note There is never any need to call *FetchParams* when working with *TSQLClientDataSet*, because the *Params* property always reflects the parameters on the internal source dataset. With *TClientDataSet*, The *FetchParams* method (or the Fetch Params command) only works if the client dataset is connected to a provider whose associated dataset can supply parameters. For example, if the source dataset is a *TSQLDataSet* object with a *CommandType* of *ctTable*, there are no parameters to fetch.

If the provider is on a separate system as part of a stateless application server, you can't use *FetchParams* to retrieve output parameters. In a stateless application server, other clients can change and rerun the query or stored procedure, changing output parameters before the call to *FetchParams*. To retrieve output parameters from a stateless application server, use the *Execute* method. If the provider is associated with a query or stored procedure, *Execute* tells the provider to execute the query or stored procedure and return any output parameters. These returned parameters are then used to automatically update the *Params* property.

Passing parameters to the source dataset

Client datasets can pass parameters to the source dataset to specify what data they want provided in the data packets it sends. These parameters can specify

- Parameter values for a query or stored procedure that is run on the application server
- Field values that limit the records sent in data packets

You can specify parameter values that your client dataset sends to the provider at design time or at runtime. At design time, select the client dataset, and then double-click the *Params* property in the Object Inspector. This brings up the collection editor, where you can add, delete, or rearrange parameters. By selecting a parameter in the collection editor, you can use the Object Inspector to edit the properties of that parameter.

At runtime, use the *CreateParam* method of the *Params* property to add parameters to your client dataset. *CreateParam* returns a parameter object, given a specified name, parameter type, and datatype. You can then use the properties of that parameter object to assign a value to the parameter.

For example, the following code sets the value of a parameter named CustNo to 605:

```
with ClientDataSet1.Params.CreateParam(ftInteger, 'CustNo', ptInput) do
    AsInteger := 605;
```

If the client dataset is not active, you can send the parameters to the provider and retrieve a data packet that reflects those parameter values simply by setting the *Active* property to *True*.

Sending query or stored procedure parameters

When a provider represents the results of a query or stored procedure, you can use the *Params* property to specify parameter values. When the client dataset requests data from the source dataset or uses its *Execute* method to run a query or stored procedure that does not return a dataset, it passes these parameter values along with the request for data or the execute command. When the provider receives these parameter values, it assigns them to its associated dataset. It then instructs the dataset to execute its query or stored procedure using these parameter values, and, if the client dataset requested data, begins providing data, starting with the first record in the result set.

Note Parameter names should match the names of the corresponding parameters on the source dataset.

Limiting records with parameters

When a provider represents a *TSQLTable* component, you can use *Params* property to limit the records that are provided to the *Data* property.

Each parameter name must match the name of a field in the *TSQLTable* component associated with the provider. The provider then sends only those records whose values on the corresponding fields match the values assigned to the parameters.

For example, consider an application that displays the orders for a single customer. When the user identifies the customer, the client dataset sets its *Params* property to include a single parameter named CustID (or whatever field in the source table is called) whose value identifies the customer whose orders should be displayed. When the client dataset requests data from the source dataset, it passes this parameter value. The provider then sends only the records for the identified customer. This is more efficient than letting the provider send all the orders records to the client application and then filtering the records using the client dataset.

Specifying the command to execute on the server

When using *TSQLClientDataSet*, the application does not have direct access to the source dataset. Instead, the client dataset must specify an SQL statement it executes to produce data packets. It does this using the *CommandText* property. The *CommandType* property indicates whether *CommandText* represents an SQL statement for the server to execute, the name of a table, or the name of a stored procedure.

When using *TClientDataSet*, the source dataset typically determines what data is supplied to clients. This dataset may have a property that specifies an SQL statement to generate the data, or it may represent a specific database table or stored procedure. For example, the *CommandText* property of *TSQLDataSet* specifies an SQL statement, a table name, or a stored procedure name, depending on the value of the *CommandType* property, while the *ProcedureName* property of *TSQLStoredProc* specifies the stored procedure it executes.

If the provider allows, *TClientDataSet* can override the property that indicates what data the dataset represents. This enables the client dataset to specify dynamically what data it wants to see. As with *TSQLClientDataSet*, you can set *CommandText* to an SQL statement that replaces the SQL on the provider's dataset, the name of a table or the name of a stored procedure. Which type of value to use is determined by the type of dataset associated with the provider.

By default, the internal provider used by *TSQLClientDataSet* always allows its client dataset to specify a *CommandText* value, because there is no other way to specify what data to fetch. For external provider components, however, the default is not to allow client datasets to specify *CommandText* in this way. To allow *TClientDataSet* to use its *CommandText* property, you must add *poAllowCommandText* to the *Options* property of the provider. Otherwise, the value of *CommandText* is ignored.

Warning Do not remove *poAllowCommandText* from the *Options* property of *TSQLClientDataSet* before opening the dataset.

The client dataset sends its *CommandText* string to the provider at two times:

- When the client dataset first opens. After it has retrieved the first data packet from the provider, the client dataset does not send *CommandText* when fetching subsequent data packets.
- When the client dataset sends an *Execute* command to the provider.

To send an SQL command or to change a table or stored procedure name at any other time, you must explicitly use the *IAppServer* interface that is available as the *AppServer* property.

Requesting data from the source dataset

The following table lists the properties and methods of client datasets that determine how data is fetched from a provider:

Table 20.3 Client datasets properties and method for handling data requests

Property or method	Purpose
FetchOnDemand property	Determines whether the client dataset automatically fetches data as needed, or relies on the application to call its *GetNextPacket*, *FetchBlobs*, and *FetchDetails* functions to retrieve additional data.
PacketRecords property	Specifies the type or number of records to include in each data packet.
GetNextPacket method	Fetches the next data packet from the provider.
FetchBlobs method	Fetches any BLOB fields for the current record when the provider does not include BLOB data automatically.
FetchDetails method	Fetches nested detail datasets for the current record when the provider does not include these in data packets automatically.

By default, a client dataset retrieves all records from the source dataset. You can change this using *PacketRecords* and *FetchOnDemand*.

Incremental fetching

PacketRecords specifies either how many records to fetch at a time, or the type of records to return. By default, *PacketRecords* is set to *-1*, which means that all available records are fetched at once, either when the client dataset is first opened, or when the application explicitly calls *GetNextPacket*. When *PacketRecords* is *-1*, then after the client dataset first fetches data, it never needs to fetch more data because it already has all available records.

To fetch records in small batches, set *PacketRecords* to the number of records to fetch. For example, the following statement sets the size of each data packet to ten records:

```
ClientDataSet1.PacketRecords := 10;
```

This process of fetching records in batches is called “incremental fetching”. Client datasets use incremental fetching when *PacketRecords* is greater than zero. By default, the client dataset automatically calls *GetNextPacket* to fetch data as needed. Newly fetched packets are appended to the end of the data already in the client dataset.

GetNextPacket returns the number of records it fetches. If the return value is the same as *PacketRecords*, the end of available records was not encountered. If the return value is greater than *0* but less than *PacketRecords*, the last record was reached during the fetch operation. If *GetNextPacket* returns *0*, then there are no more records to fetch.

Warning Incremental fetching does not work if you are fetching data from a remote provider on a stateless application server.

You can also use *PacketRecords* to fetch metadata information about the source dataset. To retrieve metadata information, set *PacketRecords* to *0*.

Fetch-on-demand

Automatic fetching of records is controlled by the *FetchOnDemand* property. When *FetchOnDemand* is *True* (the default), automatic fetching is enabled. To prevent the client dataset from automatically fetching records as needed, set *FetchOnDemand* to *False*. When *FetchOnDemand* is *False*, the application must explicitly call *GetNextPacket* to fetch records.

For example, applications that need to represent extremely large read-only datasets can turn off *FetchOnDemand* to ensure that the client datasets do not try to load more data than can fit into memory. Between fetches, the client dataset frees its cache using the *EmptyDataSet* method. This approach, however, does not work well when the client must post updates to the server.

When using *TSQLClientDataSet*, the *Options* property controls whether records in data packets include BLOB data and nested detail datasets. When using *TClientDataSet*, the external provider controls whether this information is included in data packets. If data packets do not include this information, the *FetchOnDemand* property causes the client dataset to automatically fetch BLOB data and detail datasets on an as-needed basis. If *FetchOnDemand* is *False,* and BLOB data and detail datasets are not included in data packets, you must explicitly call the *FetchBlobs* or *FetchDetails* method to retrieve this information.

Updating records

Client datasets work with a local copy of data. The user sees and edits this copy in the client application's data-aware controls. User changes are temporarily stored by the client dataset in an internally maintained change log. The contents of the change log are stored as a data packet in the *Delta* property. To make the changes in *Delta* permanent, the client dataset must apply them to the database.

When a client applies updates to the server, the following steps occur:

1. The application calls the *ApplyUpdates* method of a client dataset object. This method passes the contents of the client dataset's *Delta* property to the provider. *Delta* is a data packet that contains a client dataset's updated, inserted, and deleted records.
2. The provider applies the updates to the database (or source dataset), caching any problem records that it can't resolve itself. See "Responding to client update requests" on page 21-7 for details on how the provider applies updates.
3. The provider returns all unresolved records to the client dataset in a *Result* data packet. The *Result* data packet contains all records that were not updated. It also contains error information, such as error messages and error codes.
4. The client dataset attempts to reconcile update errors returned in the *Result* data packet on a record-by-record basis.

Applying updates

Changes made to the client dataset's local copy of data are not sent to the database server until the client application calls the *ApplyUpdates* method for the dataset. *ApplyUpdates* takes the changes in the change log, and sends them as a data packet (called *Delta*) to the provider.

ApplyUpdates takes a single parameter, *MaxErrors*, which indicates the maximum number of errors that the provider should tolerate before aborting the update process. If *MaxErrors* is *0*, then as soon as an update error occurs, the entire update process is terminated. No changes are written to the database, and the client dataset's change log remains intact. If *MaxErrors* is *-1*, any number of errors is tolerated, and the change log ends up containing all records that could not be successfully applied. If *MaxErrors* is a positive value, and more errors occur than are permitted by *MaxErrors*, all updates are aborted. If fewer errors occur than specified by *MaxErrors*, all records successfully applied are automatically cleared from the client dataset's change log.

ApplyUpdates returns the number of actual errors encountered, which is always less than or equal to *MaxErrors* plus one. This return value indicates the number of records that could not be written to the database.

The client dataset's *ApplyUpdates* method does the following:

1. It indirectly calls the provider's *ApplyUpdates* method. The provider's *ApplyUpdates* method writes the updates to the database (or source dataset) and attempts to correct any errors it encounters. Records that it cannot apply because of error conditions are sent back to the client dataset.
2. The client dataset 's *ApplyUpdates* method then attempts to reconcile these problem records by calling the *Reconcile* method. *Reconcile* is an error-handling routine that calls the *OnReconcileError* event handler. You must code the *OnReconcileError* event handler to correct errors. For details about using *OnReconcileError*, see "Reconciling update errors" on page 20-31.
3. Finally, *Reconcile* removes successfully applied changes from the change log and updates *Data* to reflect the newly updated records. When *Reconcile* completes, *ApplyUpdates* reports the number of errors that occurred.

Tip If the provider is on a stateless application server, you may want to communicate with it about persistent state information before or after you apply updates. The client dataset receives a *BeforeApplyUpdates* event before the updates are sent, which lets you send persistent state information to the server. After the updates are applied (but before the reconcile process), the client dataset receives an *AfterApplyUpdates* event where you can respond to any persistent state information returned by the application server.

Reconciling update errors

The provider returns error records and error information to the client dataset in a result data packet. If the provider returns an error count greater than zero, then for each record in the result data packet, the client dataset's *OnReconcileError* event occurs.

You should always code the *OnReconcileError* event handler, even if only to discard the records returned by the provider. The *OnReconcileError* event handler is passed four parameters:

- *DataSet:* A client dataset that contains the updated record which couldn't be applied. You can use client dataset methods to obtain information about the problem record and to change the record in order to correct any problems. In particular, you will want to use the *CurValue*, *OldValue*, and *NewValue* properties of the fields in the current record to determine the cause of the update problem. However, you must not call any client dataset methods that change the current record in an *OnReconcileError* event handler.
- *E:* An *EReconcileError* object that represents the problem that occurred. You can use this exception to extract an error message or to determine the cause of the update error.
- *UpdateKind:* The type of update that generated the error. *UpdateKind* can be *ukModify* (the problem occurred updating an existing record that was modified), *ukInsert* (the problem occurred inserting a new record), or *ukDelete* (the problem occurred deleting an existing record).
- *Action:* A **var** parameter that lets you indicate what action to take when the *OnReconcileError* handler exits. On entry into the handler, *Action* is set to the action taken by the resolution process on the provider. In your event handler, you set this parameter to
  - Skip this record, leaving it in the change log. (raSkip)
  - Stop the entire reconcile operation. (raAbort)
  - Merge the modification that failed into the corresponding record from the server. (raMerge) This only works if the server record does not include any changes to fields modified in the client dataset's record.
  - Replace the current update in the change log with the value of the record in the event handler (which has presumably been corrected). (raCorrect)
  - Back out the changes for this record on the client dataset, reverting to the originally provided values. (raCancel)
  - Update the current record value to match the record on the server. (raRefresh)

The following code shows an *OnReconcileError* event handler that uses the reconcile error dialog from the RecError unit, which ships in the object repository directory. (To use this dialog, add RecError to your uses clause.)

```
procedure TForm1.ClientDataSetReconcileError(DataSet: TCustomClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind, var Action TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

Refreshing records

Client datasets work with an in-memory snapshot of data. If that data comes from a database server, then as time elapses other users may modify it. The data in the client dataset becomes a less and less accurate picture of the underlying data.

Like any other dataset, client datasets have a *Refresh* method that updates its records to match the current values on the server. However, calling *Refresh* only works if there are no edits in the change log. Calling *Refresh* when there are unapplied edits results in an exception.

Client applications can also update the data while leaving the change log intact. To do this, call the client dataset's *RefreshRecord* method. Unlike the *Refresh* method, *RefreshRecord* updates only the current record in the client dataset. *RefreshRecord* changes the record value originally obtained from the provider but leaves any changes in the change log.

Warning It may not always be appropriate to call *RefreshRecord*. If the user's edits conflict with changes to the underlying dataset made by other users, calling *RefreshRecord* will mask this conflict. When the client application applies its updates, no reconcile error will occur and the application can't resolve the conflict.

In order to avoid masking update errors, you may want to check that there are no pending updates for the current record before calling *RefreshRecord*. For example, the following code raises an exception if an attempt is made to refresh a modified record:

```
if ClientDataSet1.UpdateStatus <> usUnModified then
  raise Exception.Create('You must apply updates before refreshing the current record.');
ClientDataSet1.RefreshRecord;
```

Communicating with providers using custom events

Client datasets communicate with a provider component through a special interface called *IAppServer*. If the provider is local, *IAppServer* is the interface to an automatically-generated object that handles all communication between the client dataset and its provider. If the provider is remote, *IAppServer* is the interface to a remote data module on the application server

TClientDataSet provides many opportunities for customizing the communication that uses the *IAppServer* interface. Before and after every *IAppServer* method call that is directed at the client dataset's provider, *TClientDataSet* receives special events that allow it to communicate arbitrary information with its provider. These events are matched with similar events on the provider. Thus for example, when the client dataset calls its *ApplyUpdates* method, the following events occur:

1 The client dataset receives a *BeforeApplyUpdates* event, where it specifies arbitrary custom information in an OleVariant called *OwnerData*.

2 The provider receives a *BeforeApplyUpdates* event, where it can respond to the *OwnerData* from the client dataset and update the value of *OwnerData* to new information.

3 The provider goes through its normal process of assembling a data packet (including all the accompanying events).

4 The provider receives an *AfterApplyUpdates* event, where it can respond to the current value of *OwnerData* and update it to a value for the client dataset.

5 The client dataset receives an *AfterApplyUpdates* event, where it can respond to the returned value of *OwnerData*.

Every other *IAppServer* method call is accompanied by a similar set of *BeforeXXX* and *AfterXXX* events that let you customize the communication between *TClientDataSet* and its provider.

In addition, *TClientDataSet* has a special method, *DataRequest*, whose only purpose is to allow application-specific communication with the provider. When the client dataset calls *DataRequest*, it passes an OleVariant as a parameter that can contain any information you want. This, in turn, generates an *OnDataRequest* event on the provider, where you can respond in any application-defined way and return a value to the client dataset.

Using an SQL client dataset

TSQLClientDataSet is a special type of client dataset designed for simple two-tiered applications. Like a unidirectional dataset, it can use an SQL connection component to connect to a database server and specify an SQL statement to execute on that server. Like other client datasets, it buffers data in memory to allow full navigation and editing support.

TSQLClientDataSet works the same way as a generic client dataset (*TClientDataSet*) that is linked to a unidirectional dataset by a dataset provider. In fact, *TSQLClientDataSet* has its own, internal provider, which it uses to communicate with an internally created unidirectional dataset.

Using an SQL client dataset can simplify the process of two-tiered application development because you don't need to work with as many components. Although you can't access the internal provider directly, the SQL client dataset publishes properties that let you configure how it applies updates. Typically, however, the defaults provided by *TSQLClientDataSet* are sufficient.

When to use TSQLClientDataSet

TSQLClientDataSet is intended for use in a simple two-tiered database applications and briefcase model applications. It provides an easy-to-set up component for linking to the database server, fetching data, caching updates, and applying them back to the server. It can be used in most two-tiered applications.

There are times, however, when it is more appropriate to use *TClientDataSet*:

- If you are not using data from a database server (for example, if you are using a dedicated file on disk), then *TClientDataSet* has the advantage of less overhead.

- Only *TClientDataSet* can be used in a multi-tiered database application. Thus, if you are writing a multi-tiered application, or if you intend to scale up to a multi-tiered application eventually, you should use *TClientDataSet* with an external provider and source dataset.
- Because the source dataset is internal to the SQL client dataset component, you can't link two source datasets in a master/detail relationship to obtain nested detail sets. (You can, however, link two SQL client datasets into a master/detail relationship.)
- The SQL client dataset does not surface many of the events that occur on its internal dataset provider. However, in most cases, these events are used in multi-tiered applications, and are not needed for two-tiered applications.

Setting up an SQL client dataset

To use *TSQLClientDataSet*,

1. Place the *TSQLClientDataSet* component in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
2. Identify the database server that contains the data. There are two ways to do this:
   - If you have defined a named connection in the connections file, you can simply specify the name of that connection as the value of the *ConnectionName* property. For details on named connections, see "Naming a connection description" on page 19-3.
   - For greater control over connection properties, transaction support, login support, and the ability to use a single connection for more than one dataset, use a separate *TSQLConnection* component instead. Specify the *TSQLConnection* component as the value of the *DBConnection* property. For details on *TSQLConnection*, see Chapter 19, "Connecting to databases".
3. Indicate what data you want to fetch from the server. There are three ways to do this:
   - Set *CommandType* to *ctQuery* and set *CommandText* to an SQL statement you want to execute on the server. This statement is typically a SELECT statement. Supply the values for any parameters using the *Params* property.
   - Set *CommandType* to *ctStoredProc* and set *CommandText* to the name of the stored procedure you want to execute. Supply the values for any input parameters using the *Params* property.
   - Set *CommandType* to *ctTable* and set *CommandText* to the name of the database tables whose records you want to use.
4. If the data is to be used with visual data controls, add a data source component to the form or data module, and set its *DataSet* property to the *TSQLClientDataSet* object. The data source component forwards the data in the client dataset's in-memory cache to data-aware components for display. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.

5. If desired, configure the way the internal provider applies updates. The properties and events that allow this are described in "Configuring the internal provider" below.
6. Activate the dataset by setting the *Active* property to *True* (or, at runtime, calling the *Open* method).
7. If you executed a stored procedure, use the *Params* property to retrieve any output parameters.
8. When the user has edited the data in the SQL client dataset, you can apply those edits back to the database server by calling the *ApplyUpdates* method. Resolve any update errors in an *OnUpdateError* event handler. For more information on applying updates, see "Updating records" on page 20-30.

Configuring the internal provider

Because *TSQLClientDataSet* does not use an external provider component, you have less control over how it provides data and applies updates. Typically, this is not a problem, because *TSQLClientDataSet* publishes the properties and events of its internal provider that you are likely to need.

Because you specify the SQL statement to execute as the value of the SQL client dataset's *CommandText* property, you can control what data is included in data packets. Field properties can be set using persistent fields on the client dataset.

TSQLClientDataSet has two published properties that influence how the internal provider applies updates. These properties are the same as properties on the provider, and the *TSQLClientDataSet* component simply forwards any values you set on to the internal provider:

- *Options* controls whether nested detail sets and BLOB data are included in data packets or fetched separately, whether specific types of edits (insertions, modifications, or deletions) are disabled, whether a single update can affect multiple server records, and whether the client dataset's records are refreshed when it applies updates. *Options* is identical to the provider's *Options* property. As a result, it allows you to set options that are not relevant or appropriate to *TSQLClientDataSet*. For example, there is no reason to include *poIncFieldProps*, because the internal source dataset does not use persistent field components. Similarly, you do not want to exclude *poAllowCommandText*, which is included by default, because that would make it impossible for the client dataset to specify a query to execute. For information on the provider's *Options* property, see "Setting options that influence the data packets" on page 21-4.
- *UpdateMode* controls what fields are used to locate records in the SQL statements the provider generates for applying updates. *UpdateMode* is identical to the provider's *UpdateMode* property. For information on the provider's *UpdateMode* property, see "Influencing how updates are applied" on page 21-8.

TSQLClientDataSet also publishes several events that correspond to events on the internal provider. Some of the most important of these are

- *OnGetTableName*, which lets you supply the name of the database table to which the dataset should apply updates. This lets the internal provider generate SQL statements for updates when it can't identify the database table from the query specified by *CommandText*. For example, if the query executes a stored procedure or multi-table join that only requires updates to a single table, supplying an *OnGetTableName* event handler allows the internal provider to correctly apply updates.
- *BeforeUpdateRecord*, which occurs for every record in the delta packet. This event lets you make any last-minute changes before the record is inserted, deleted, or modified. It also provides a way for you to execute your own SQL statements to apply the update in cases where the provider can't generate correct SQL (for example, for multi-table joins where multiple tables must be updated.) To execute your own SQL statements, use the *Execute* method of the *TSQLConnection* component.
- *OnUpdateError*, which occurs every time the internal provider can't apply an update to the database server. This event lets you correct update errors during the update process so that they do not count toward the maximum number of errors allowed in the entire update operation.

Using a client dataset with file-based data

Client datasets can function independently of a provider, such as in file-based database applications and "briefcase model" applications. When it does not use a provider, however, the client dataset cannot get table definitions and data from the server, and there is no server to which it can apply updates. Instead, the client dataset must independently

- Define and create tables
- Load saved data
- Merge edits into its data
- Save data

Creating a new dataset

There are three ways to define and create client datasets that do not represent server data:

- You can define and create a new client dataset by creating persistent fields for the dataset and then choosing Create Dataset from its context menu.
- You can define and create a new client dataset based on field definitions and index definitions.
- You can copy an existing dataset.

Creating a new dataset using persistent fields

The following steps describe how to create a new client dataset using the Fields Editor:

1. From the Component palette, add a *TClientDataSet* component to your application.
2. Right-click the client dataset and select Fields Editor. In the Fields editor, right-click and choose the New Field command. Describe the basic properties of your field definition. Once the field is created, you can alter its properties in the Object Inspector by selecting the field in the Fields editor.

   Continue adding fields in the fields editor until you have described your client dataset.
3. Right-click the client dataset and choose Create DataSet. This creates an empty client dataset from the persistent fields you added in the Fields Editor.
4. Right-click the client dataset and choose Save To File. (This command is not available unless the client dataset contains a dataset.)
5. In the File Save dialog, choose a file name and save a copy of your client dataset to that file.

Note You can also create the client dataset at runtime using persistent fields that are saved with the client dataset. Simply call the *CreateDataSet* method.

Creating a dataset using field and index definitions

If you want to create persistent indexes for your client dataset as well as fields, you must use field and index definitions. Use the *FieldDefs* property to specify the fields in your dataset and the *IndexDefs* property to specify any indexes. Once the dataset is specified, right-click the client dataset and choose Create DataSet at design time, or call the *CreateDataSet* method at runtime.

When defining the index definitions for your client dataset, two properties of the index definition apply uniquely to client datasets. These are *TIndexDef.DescFields* and *TIndexDef.CaseInsFields*.

DescFields lets you define indexes that sort records in ascending order on some fields and descending order on other fields. Instead of using the *ixDescending* option to sort in descending order on all the fields in the index, list only those fields that should sort in descending order as the value of *DescFields*. For example, when defining an index that sorts on Field1, then Field2, then Field3, setting *DescFields* to

```
Field1;Field3
```

results in an index that sorts Field2 in ascending order and Field1 and Field3 in descending order.

CaseInsFields lets you define indexes that sort records case-sensitively on some fields and case-insensitively on other fields. Instead of using the *isCaseInsensitive* option to sort case-insensitively on all the fields in the index, list only those fields that should sort case-insensitively as the value of *CaseInsFields*. Like *DescFields*, *CaseInsFields* takes a semicolon-delimited list of field names.

You can specify the field and index definitions at design time using the Collection editor. Just choose the appropriate property in the Object Inspector (*FieldDefs* or *IndexDefs*), and double-click to display the Collection editor. Use the Collection editor to add, delete, and rearrange definitions. By selecting definitions in the Collection editor you can edit their properties in the Object Inspector.

You can also specify the field and index definitions in code at runtime. For example, the following code creates and activates a client dataset in the form's *OnCreate* event handler:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
with ClientDataSet1 do
begin
  with FieldDefs.AddFieldDef do
  begin
    DataType := ftInteger;
    Name := 'Field1';
  end;
  with FieldDefs.AddFieldDef do
  begin
    DataType := ftString;
    Size := 10;
    Name := 'Field2';
  end;
  with IndexDefs.AddIndexDef do
  begin
    Fields := 'Field1';
    Name := 'IntIndex';
  end;
  CreateDataSet;
end;
end;
```

Creating a dataset based on an existing table

If you are converting a server-based database application into a file-based application, you can copy existing tables and save them to a file from the IDE. The following steps indicate how to copy an existing table:

1. From the dbExpress page of the Component palette, add a *TSQLDataSet* component and a *TSQLConnection* component to your application. Set the *CommandType* and *CommandText* properties of the SQL dataset to identify the existing database table. Set its *SQLConnection* property to the *TSQLConnection* component. Set properties on the *TSQLConnection* component to attach to the database server. Set the SQL dataset's *Active* property to *True*.
2. From the Component palette, add a *TClientDataSet* component.
3. Right-click the client dataset and select Assign Local Data. In the dialog that appears, choose the SQL dataset component that you added in step 1. Choose OK.
4. Right-click the client dataset and choose Save To File. (This command is not available unless the client dataset contains data.)

5 In the File Save dialog, choose a file name and save a copy of your database table.

To copy from another dataset at runtime, you can assign its data directly or, if the source is another client dataset, you can clone the cursor.

Loading data from a file or stream

To load data from a file, call a client dataset's *LoadFromFile* method. *LoadFromFile* takes one parameter, a string that specifies the file from which to read data. The file name can be a fully qualified path name, if appropriate. If you always load the client dataset's data from the same file, you can use the *FileName* property instead. If *FileName* names an existing file, the data is automatically loaded when the client dataset is opened.

To load data from a stream, call the client dataset's *LoadFromStream* method. *LoadFromStream* takes one parameter, a stream object that supplies the data.

The data loaded by *LoadFromFile* (*LoadFromStream)* must have previously been saved in a client dataset's data format by this or another client dataset using the *SaveToFile (SaveToStream)* method. For more information about saving data to a file or stream, see "Saving data to a file or stream" on page 20-41.

When you call *LoadFromFile* or *LoadFromStream*, all data in the file is read into the *Data* property. Any edits that were in the change log when the data was saved are read into the *Delta* property.

Merging changes into data

When you edit the data in a client dataset, the changes you make are recorded in the change log, but the changes do not affect the original version of the data.

To make your changes permanent, call *MergeChangeLog*. *MergeChangeLog* overwrites records in *Data* with any changed field values in the change log.

After *MergeChangeLog* executes, *Data* contains a mix of existing data and any changes that were in the change log. This mix becomes the new *Data* baseline against which further changes can be made. *MergeChangeLog* clears the change log of all records and resets the *ChangeCount* property to *0*.

Warning Do not call *MergeChangeLog* for client datasets that represent the data from a database server or source dataset. In this case, call *ApplyUpdates* to write changes to the database or source dataset. For more information, see "Applying updates" on page 20-31.

Note It is also possible to merge changes into the data of a separate client dataset if that dataset originally provided the data in the *Data* property. To do this, you must use a dataset provider. For an example of how to do this, see "Assigning data directly" on page 20-23.

Saving data to a file or stream

If you use a client dataset in a file-based application, then when you edit data and merge it, the changes you make exist only in memory. To make a permanent record of your changes, you must write them to disk. You can save the data to disk using the *SaveToFile* method.

SaveToFile takes one parameter, a string that specifies the file into which to write data. The file name can be a fully qualified path name, if appropriate. If the file already exists, its current contents are completely overwritten.

If you always save the data to the same file, you can use the *FileName* property instead. If *FileName* is set, the data is automatically saved to the named file when the client dataset is closed.

You can also save data to a stream, using the *SaveToStream* method. *SaveToStream* takes one parameter, a stream object that receives the data.

Note If you save a client dataset while there are still edits in the change log, these are not merged with the data. When you reload the data, using the *LoadFromFile* or *LoadFromStream* method, the change log will still contain the unmerged edits. This is important for applications that support the briefcase model, where those changes will eventually have to be applied to a provider component.

Note *SaveToFile* does not preserve any indexes you add to the client dataset unless they are created with the dataset using index definitions.

Chapter

21

Using provider components

Provider components (*TDataSetProvider*) supply the mechanism by which client datasets obtain their data unless they use dedicated files on disk. Providers

- Receive data requests from a client dataset, fetch the requested data from the database server, package the data into a transportable data packet, and return the data to the client dataset. This activity is called "providing."
- Receive updated data from a client dataset, apply updates to the database or source dataset, and log any updates that cannot be applied, returning unresolved updates to the client dataset for further reconciliation. This activity is called "resolving."

Most of the work of a provider component happens automatically. You need not write any code on the provider to create data packets from the data in another dataset or apply updates to a dataset or database server. However, you may want to use the properties and events of the provider component to control its interaction with clients.

When using *TSQLClientDataSet*, the provider is internal to the client dataset, and the application has no direct access to it. When using *TClientDataSet*, however, the provider is a separate component that you can use to control what information is packaged for clients and for responding to events that occur around the process of providing and resolving. *TSQLClientDataSet* surfaces some of the provider's properties and events as its own properties and events, but for the greatest amount of control, you may want to use *TClientDataSet* with a separate provider component.

When using a separate provider component, it can reside in the same application as the client dataset, or it can reside on an application server as part of a multi-tiered application.

Note The components that connect a client dataset to a provider on a remote application server must be purchased separately.

This chapter describes how to use a provider component to control the interaction with client datasets.

Determining the source of data

When you use a provider component, you must specify a dataset that it can use to fetch the data it assembles into data packets. You specify this dataset by setting the *DataSet* property of the provider. At design time, select from available datasets in the *DataSet* property drop-down list in the Object Inspector.

TDataSetProvider interacts with the source dataset using the *IProviderSupport* interface. This interface is introduced by *TDataSet*, so it is available for all datasets. However, the *IProviderSupport* methods implemented in *TDataSet* are mostly stubs that don't do anything or that raise exceptions.

The unidirectional dataset classes that ship with Kylix override these methods to implement the *IProviderSupport* interface in a more useful fashion. Client datasets use the default implementation inherited from *TDataSet*, but can still be used as a source dataset as long as the provider's *ResolveToDataSet* property is *True*.

Component writers that create their own custom descendants from *TDataSet* must override all appropriate *IProviderSupport* methods if their datasets are to supply data to a provider. If the provider need only provide data packets on a read-only basis (that is, if it does not need to apply updates), the *IProviderSupport* methods implemented in *TDataSet* may be sufficient.

Communicating with the client dataset

All communication between a provider and a client dataset takes place through an *IAppServer* interface. If the provider is in the same application as the client, this interface is implemented by a hidden object generated automatically for you. If the provider is part of a multi-tiered application, this is the interface for the application server's remote data module.

IAppServer provides the bridge between client dataset and the provider. Most applications do not use *IAppServer* directly, but invoke it indirectly through the properties and methods of the client dataset. However, when necessary, you can make direct calls to the *IAppServer* interface by using the *AppServer* property of the client dataset.

Table 21.1 lists the methods of the *IAppServer* interface, as well as the corresponding methods and events on the provider component and the client dataset. These *IAppServer* methods include a *Provider* parameter. In multi-tiered applications, this parameter indicates the provider on the application server with which the client dataset communicates. Most methods also include an OleVariant parameter called *OwnerData* that allows a client application and an application server to pass custom information back and forth. *OwnerData* is not used by default, but is passed to all event handlers so that you can write code that allows your provider to adjust to application-defined information before and after each call from a client dataset.

Table 21.1 AppServer interface members

IAppServer	TDataSetProvider	TClientDataSet
AS_ApplyUpdates method	ApplyUpdates method, BeforeApplyUpdates event, AfterApplyUpdates event	ApplyUpdates method, BeforeApplyUpdates event, AfterApplyUpdates event
AS_DataRequest method	DataRequest method, OnDataRequest event	DataRequest method
AS_Execute method	Execute method, BeforeExecute event, AfterExecute event	Execute method, BeforeExecute event, AfterExecute event
AS_GetParams method	GetParams method, BeforeGetParams event, AfterGetParams event	FetchParams method, BeforeGetParams event, AfterGetParams event
AS_GetProviderNames method	Used to identify all available providers	Used to create a design-time list for ProviderName property
AS_GetRecords method	GetRecords method, BeforeGetRecords event, AfterGetRecords event	GetNextPacket method, Data property, BeforeGetRecords event, AfterGetRecords event
AS_RowRequest method	RowRequest method, BeforeRowRequest event, AfterRowRequest event	FetchBlobs method, FetchDetails method, RefreshRecord method, BeforeRowRequest event, AfterRowRequest event

Choosing how to apply updates

By default, when *TDataSetProvider* components apply updates and resolve update errors, they communicate directly with the database server using dynamically generated SQL statements. This approach has the advantage that your application does not need to merge updates twice (first to the source dataset, and then to the remote database server).

However, you may not always want to take this approach. For example, you may be using a dataset that does not get its data from an SQL server (for example if you are providing from a *TClientDataSet* component). Alternately, you may be using a custom dataset that can apply updates to an SQL server, but you want to use some of the dataset events.

TDataSetProvider lets you decide whether to apply updates to the database server using SQL or to the source dataset by setting the *ResolveToDataSet* property. When this property is *True*, updates are applied to the dataset. When it is *False*, the provider generates SQL statements to apply updates and the source dataset forwards them to the database server.

Controlling what information is included in data packets

There are a number of ways to control what information is included in data packets that are sent to and from the client. These include

- Specifying what fields appear in data packets
- Setting options that influence the data packets
- Adding custom information to data packets

Specifying what fields appear in data packets

To control what fields are included in data packets, create persistent fields on the dataset that the provider uses to build the packets. The provider then includes only these fields. Persistent fields whose values are generated dynamically by the source dataset (such as calculated fields) can be included, but appear to client datasets on the receiving end as static read-only fields. For information about persistent fields, see "Persistent field components" on page 17-3.

If the client dataset will be editing the data and applying updates to the database server, you must include enough fields so that there are no duplicate records in the data packet. Otherwise, when the updates are applied, it is impossible to determine which record to update. If you do not want the client dataset to be able to see or use extra fields that are provided only to ensure uniqueness, set the *ProviderFlags* property for those fields to include *pfHidden*.

Note Including enough fields to avoid duplicate records is also a consideration when specifying a query on the source dataset. The query should include enough fields so that records are unique, even if your application does not use all the fields.

Setting options that influence the data packets

The *Options* property of the provider component lets you specify when BLOBs or nested detail tables are sent, whether field display properties are included, what type of updates are allowed, and so on. The following table lists the possible values that can be included in *Options*.

Table 21.2 Provider options

Value	Meaning
poReadOnly	The client dataset can't apply updates to the provider.
poDisableEdits	Client datasets can't modify existing data values. If the user tries to edit a field, the client dataset raises an exception. (This does not affect the client dataset's ability to insert or delete records.)
poDisableInserts	Client datasets can't insert new records. If the user tries to insert a new record, the client dataset raises an exception. (This does not affect the client dataset's ability to delete records or modify existing data.)

Table 21.2 Provider options (continued)

Value	Meaning
poDisableDeletes	Client datasets can't delete records. If the user tries to delete a record, the client dataset raises an exception. (This does not affect the client dataset's ability to insert or modify records.)
poFetchBlobsOnDemand	BLOB field values are not included in the data packet. Instead, client datasets must request these values on an as-needed basis. If the client dataset's *FetchOnDemand* property is *True*, it requests these values automatically. Otherwise, the application must call the client dataset's *FetchBlobs* method to retrieve BLOB data.
poFetchDetailsOnDemand	When the provider represents the master of a master/detail relationship, nested detail values are not included in the data packet. Instead, client applications request these on an as-needed basis. If the client dataset's *FetchOnDemand* property is *True*, it requests these values automatically. Otherwise, the application must call the client dataset's *FetchDetails* method to retrieve nested details.
poIncFieldProps	The data packet includes the following field properties (where applicable): *Alignment*, *DisplayLabel*, *DisplayWidth*, *Visible*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, *Currency*, *EditMask*, *DisplayValues.*
poCascadeDeletes	When the provider represents the master of a master/detail relationship, the server deletes detail records when master records are deleted. To use this option, the database server must be set up to perform cascaded deletes as part of its referential integrity.
poCascadeUpdates	When the provider represents the master of a master/detail relationship, key values on detail tables are updated automatically when the corresponding values are changed in master records. To use this option, the database server must be set up to perform cascaded updates as part of its referential integrity.
poAllowMultiRecordUpdates	A single update can cause more than one record of the underlying database table to change. This can be the result of triggers, referential integrity, SQL statements on the source dataset, and so on. Note that if an error occurs, the event handlers provide access to the record that was updated, not the other records that change in consequence.
poNoReset	Client datasets can't specify that the provider should reposition the cursor on the first record before providing data.
poAutoRefresh	The provider refreshes the client dataset with current record values whenever it applies updates.
poPropogateChanges	Changes made by the server to updated records as part of the update process are sent back to the client and merged into the client dataset.
poAllowCommandText	The client dataset can override the associated dataset's SQL text or the name of the table or stored procedure it represents.

Adding custom information to data packets

Providers can send application-defined information to the data packets using the *OnGetDataSetProperties* event. This information is encoded as an OleVariant, and stored under a name you specify. Client datasets can then retrieve the information using their *GetOptionalParam* method. You can also specify that the information be included in delta packets that the client dataset sends when updating records. In this case, the client may never be aware of the information, but the provider can send a round-trip message to itself.

When adding custom information in the *OnGetDataSetProperties* event, each individual attribute (sometimes called an "optional parameter") is specified using a Variant array that contains three elements: the name (a string), the value (a Variant), and a boolean flag indicating whether the information should be included in delta packets when the client applies updates. Multiple attributes can be added by creating a Variant array of Variant arrays. For example, the following *OnGetDataSetProperties* event handler sends two values, the time the data was provided and the total number of records in the source dataset. Only information about the time the data was provided is returned when client datasets apply updates:

```
procedure TMyDataModule1.Provider1GetDataSetProperties(Sender: TObject; DataSet: TDataSet;
out Properties: OleVariant);
begin
  Properties := VarArrayCreate([0,1], varVariant);
  Properties[0] := VarArrayOf(['TimeProvided', Now, True]);
  Properties[1] := VarArrayOf(['TableSize', DataSet.RecordCount, False]);
end;
```

When the client dataset applies updates, the time the original records were provided can be read in the provider's *OnUpdateData* event:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
var
  WhenProvided: TDateTime;
begin
  WhenProvided := DataSet.GetOptionalParam('TimeProvided');
  ...
end;
```

Responding to client data requests

In most applications, requests for data are handled automatically. A client dataset requests a data packet by calling *GetRecords* (indirectly, through the *IAppServer* interface). The provider responds automatically by fetching data from the associated dataset, creating a data packet, and returning the packet to the client dataset.

The provider has the option of editing data after it has been assembled into a data packet but before the packet is sent to the client dataset. For example, you might want to remove records from the packet based on some criterion (such as the user's level of access), or, in a multi-tiered application, you might want to encrypt sensitive data before it is sent on to the client.

To edit the data packet before sending it on to the client dataset, write an *OnGetData* event handler. The data packet is provided as a parameter in the form of a client dataset. Using the methods of this client dataset, data can be edited before it is sent to the client.

As with all method calls that are made through the *IAppServer* interface, the provider has an opportunity to communicate persistent state information with the client dataset before and after the call to *GetRecords*. This communication takes place using the *BeforeGetRecords* and *AfterGetRecords* event handlers.

Responding to client update requests

A provider applies updates to database records based on a *Delta* data packet received from a client dataset. The client dataset requests updates by calling the *ApplyUpdates* method (indirectly, through the *IAppServer* interface).

As with all method calls that are made through the *IAppServer* interface, the provider has an opportunity to communicate persistent state information with the client dataset before and after the call to *ApplyUpdates*. This communication takes place using the *BeforeApplyUpdates* and *AfterApplyUpdates* event handlers.

When a provider receives an update request, it generates an *OnUpdateData* event, where you can edit the Delta packet before it is written to the dataset or influence how updates are applied. After the *OnUpdateData* event, the provider writes the changes to the database.

The provider performs the update on a record-by-record basis. Before the provider applies each record, it generates a *BeforeUpdateRecord* event, which you can use to screen updates before they are applied. If an error occurs when updating a record, the provider receives an *OnUpdateError* event where it can resolve the error. Usually errors occur because the change violates a server constraint or the database record was changed by a different application subsequent to its retrieval by the provider, but prior to the client dataset's request to apply updates.

Update errors can be processed by either the provider or the client dataset. When the provider is part of a multi-tiered application, it should handle all update errors that do not require user interaction to resolve. When the provider can't resolve an error condition, it temporarily stores a copy of the offending record. When record processing is complete, the provider returns a count of the errors it encountered to the client dataset, and copies the unresolved records into a results data packet that it returns to the client dataset for further reconciliation.

The event handlers for all provider events are passed the set of updates as a client dataset. If your event handler is only dealing with certain types of updates, you can filter the dataset based on the update status of records. By filtering the records, your event handler does not need to sort through records it won't be using. To filter the client dataset on the update status of its records, set its *StatusFilter* property.

Note Applications must supply extra support when the updates are directed at a dataset that does not represent a single table. For details on how to do this, see "Applying updates to datasets that do not represent a single table" on page 21-10.

Editing delta packets before updating the database

Before the provider applies updates to the database, it generates an *OnUpdateData* event. The *OnUpdateData* event handler receives a copy of the *Delta* packet as a parameter. This is a client dataset.

In the *OnUpdateData* event handler, you can use any of the properties and methods of the client dataset to edit the *Delta* packet before it is written to the dataset. One particularly useful property is the *UpdateStatus* property. *UpdateStatus* indicates what type of modification the current record in the delta packet represents. It can have any of the values in Table 21.3.

Table 21.3 UpdateStatus values

Value	Description
usUnmodified	Record contents have not been changed
usModified	Record contents have been changed
usInserted	Record has been inserted
usDeleted	Record has been deleted

For example, the following *OnUpdateData* event handler inserts the current date into every new record that is inserted into the database:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
begin
  with DataSet do
  begin
    First;
    while not Eof do
    begin
      if UpdateStatus = usInserted then
      begin
        Edit;
        FieldByName('DateCreated').AsDateTime := Date;
        Post;
      end;
      Next;
    end;
end;
```

Influencing how updates are applied

The *OnUpdateData* event also gives your provider a chance to indicate how records in the delta packet are applied to the database.

By default, changes in the delta packet are written to the database using automatically generated SQL UPDATE, INSERT, or DELETE statements such as

```
UPDATE EMPLOYEES
  set EMPNO = 748, NAME = 'Smith', TITLE = 'Programmer 1', DEPT = 52
WHERE
  EMPNO = 748 and NAME = 'Smith' and TITLE = 'Programmer 1' and DEPT = 47
```

Unless you specify otherwise, all fields in the delta packet records are included in the UPDATE clause and in the WHERE clause. However, you may want to exclude some of these fields. One way to do this is to set the *UpdateMode* property of the provider. *UpdateMode* can be assigned any of the following values:

Table 21.4 UpdateMode values

Value	Meaning
upWhereAll	All fields are used to locate fields (the WHERE clause).
upWhereChanged	Only key fields and fields that are changed are used to locate records.
upWhereOnly	Only key fields are used to locate records.

You might, however, want even more control. For example, with the previous statement, you might want to prevent the EMPNO field from being modified by leaving it out of the UPDATE clause, and leave the TITLE and DEPT fields out of the WHERE clause to avoid update conflicts when other applications have modified the data. To specify the clauses where a specific field appears, use the field's *ProviderFlags* property. *ProviderFlags* is a set that can include any of the values in Table 21.5

Table 21.5 ProviderFlags values

Value	Description
pfInWhere	The field appears in the WHERE clause of generated INSERT, DELETE, and UPDATE statements when UpdateMode is upWhereAll or upWhereChanged.
pfInUpdate	The field can appear in the UPDATE clause of generated UPDATE statements.
pfInKey	The field is used in the WHERE clause of generated statements when UpdateMode is upWhereKeyOnly.
pfHidden	The field is included in records to ensure uniqueness, but can't be seen or used on the client side.

Thus, the following *OnUpdateData* event handler allows the TITLE field to be updated and uses the EMPNO and DEPT fields to locate the desired record. If an error occurs, and a second attempt is made to locate the record based only on the key, the generated SQL looks for the EMPNO field only:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TCustomClientDataSet);
begin
  with DataSet do
  begin
    FieldByName('TITLE').ProviderFlags := [pfInUpdate];
    FieldByName('EMPNO').ProviderFlags := [pfInWhere, pfInKey];
    FieldByName('DEPT').ProviderFlags := [pfInWhere];
  end;
end;
```

Note You can use the *ProviderFlags* property to influence how updates are applied even if you are updating to a dataset and not using dynamically generated SQL. These flags still determine which fields are used to locate records and which fields get updated.

Screening individual updates

Immediately before each update is applied, the provider receives a *BeforeUpdateRecord* event. You can use this event to edit records before they are applied, similar to the way you can use the *OnUpdateData* event to edit entire delta packets. For example, the provider does not compare BLOB fields (such as memos) when checking for update conflicts. If you want to check for update errors involving BLOB fields, you can use the *BeforeUpdateRecord* event.

In addition, you can use this event to apply updates yourself or to screen and reject updates. The *BeforeUpdateRecord* event handler lets you signal that an update has been handled already and should not be applied. The provider then skips that record, but does not count it as an update error. For example, this event provides a mechanism for applying updates to a stored procedure (which can't be updated automatically), allowing the provider to skip any automatic processing once the record is updated from within the event handler.

Resolving update errors on the provider

When an error condition arises as the provider tries to post a record in the delta packet, an *OnUpdateError* event occurs. If the provider can't resolve an update error, it temporarily stores a copy of the offending record. When record processing is complete, the provider returns a count of the errors it encountered, and copies the unresolved records into a results data packet that it passes back to the client dataset for further reconciliation.

In multi-tiered applications, this mechanism lets you handle any update errors you can resolve mechanically on the application server, while still allowing user interaction on the client application to correct error conditions.

The *OnUpdateError* handler gets a copy of the record that could not be changed, an error code from the database, and an indication of whether the resolver was trying to insert, delete, or update the record. The problem record is passed back in a client dataset. You should never use the data navigation methods on this dataset. However, for each field in the dataset, you can use the *NewValue*, *OldValue*, and *CurValue* properties to determine the cause of the problem and make any modifications to resolve the update error. If the *OnUpdateError* event handler can correct the problem, it sets the *Response* parameter so that the corrected record is applied.

Applying updates to datasets that do not represent a single table

When a provider generates SQL statements that apply updates directly to a database server, it needs the name of the database table that contains the records. Obtaining this table name can be a problem if the provider is applying updates to the data underlying a stored procedure or a multi-table query. There is no easy way to ascertain the name of the table to which updates should be applied.

You can supply the table name programmatically in an *OnGetTableName* event handler. Once an event handler supplies the table name, the provider can generate appropriate SQL statements to apply updates.

Note Supplying a table name only works if the target of the updates is a single database table (that is, only the records in one table need to be updated). If the update requires making changes to multiple underlying database tables, you must explicitly apply the updates in code using the *BeforeUpdateRecord* event of the provider. Once this event handler has applied an update, you can set its *Applied* parameter to *True* so that the provider does not generate an error.

Responding to client-generated events

Provider components implement a general-purpose event that lets you create your own calls from client datasets directly to the provider. This is the *OnDataRequest* event.

OnDataRequest is not part of the normal functioning of the provider. It is simply a hook to allow client datasets to communicate directly with providers. The event handler takes an OleVariant as an input parameter and returns an OleVariant. By using OleVariants, the interface is sufficiently general to accommodate almost any information you want to pass to or from the provider.

To generate an *OnDataRequest* event, an application calls the *DataRequest* method of the client dataset.

Part

III

Writing distributed applications

The chapters in “Writing distributed applications” present concepts and skills necessary for building applications that are distributed over a local area network or over the Internet. The components discussed may not be available in all editions of Kylix.

Chapter

22

Creating Internet server applications

Kylix allows you to create Web server applications as CGI or Apache applications. These Web server applications can contain any nonvisual component. Special components on the Internet palette page make it easy to create event handlers that are associated with a specific Uniform Resource Identifier (URI) and, when processing is complete, to programmatically construct HTML documents and transfer them to the client. These components, and the architecture that defines there relationships, are collectively referred to as the WebBroker technology.

Frequently, the content of Web pages is drawn from databases. Internet components can be used to automatically manage connections to databases, allowing a single server to handle numerous simultaneous, thread-safe database connections.

This chapter describes these Internet components, and discusses the creation of several types of Internet applications.

Terminology and standards

Many of the protocols that control activity on the Internet are defined in Request for Comment (RFC) documents that are created, updated, and maintained by the Internet Engineering Task Force (IETF), the protocol engineering and development arm of the Internet. Several important RFCs are useful when writing Internet applications:

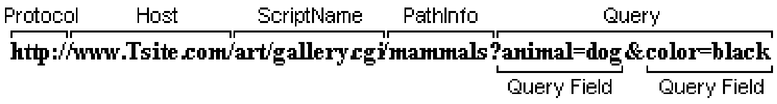
- RFC822, "Standard for the format of ARPA Internet text messages," describes the structure and content of message headers.
- RFC1521, "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies," describes the method used to encapsulate and transport multipart and multiformat messages.
- RFC1945, "Hypertext Transfer Protocol — HTTP/1.0," describes a transfer mechanism used to distribute collaborative hypermedia documents.

The IETF maintains a library of the RFCs on their Web site, `www.ietf.cnri.reston.va.us`

Parts of a Uniform Resource Locator

The Uniform Resource Locator (URL) is a complete description of the location of a resource that is available over the net. It is composed of several parts that may be accessed by an application. These parts are illustrated in Figure 22.1:

Figure 22.1 Parts of a Uniform Resource Locator


Protocol
Host
ScriptName
PathInfo
Query
http://www.Tsite.com/art/gallery.cgi/mammals?animal=dog&color=black
Query Field
Query Field

The first portion (not technically part of the URL) identifies the protocol (http). This portion can specify other protocols such as https (secure http), ftp, and so on.

The Host portion identifies the machine that runs the Web server and Web server application. Although it is not shown in the preceding picture, this portion can override the port that receives messages. Usually, there is no need to specify a port, because the port number is implied by the protocol.

The ScriptName portion specifies the name of the Web server application. This is the application to which the Web server passes messages.

Following the script name is the PathInfo. This identifies the destination of the message within the Web server application. PathInfo values may refer to directories on the host machine, the names of components that respond to specific messages, or any other mechanism the Web server application uses to divide the processing of incoming messages.

The Query portion contains a set a named values. These values and their names are defined by the Web server application.

URI vs. URL

The URL is a subset of the Uniform Resource Identifier (URI) defined in the HTTP standard, RFC1945. Web server applications frequently produce content from many sources where the final result does not reside in a particular location, but is created as necessary. URIs can describe resources that are not location-specific.

HTTP request header information

HTTP request messages contain many headers that describe information about the client, the target of the request, the way the request should be handled, and any content sent with the request. Each header is identified by a name, such as “Host” followed by a string value. For example, consider the following HTTP request:

```
GET /art/gallery.cgi/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (Linux; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

The first line identifies the request as a GET. A GET request message asks the Web server application to return the content associated with the URI that follows the word GET (in this case /art/gallery.cgi/animals?animal=doc&color=black). The last part of the first line indicates that the client is using the HTTP 1.0 standard.

The second line is the Connection header, and indicates that the connection should not be closed once the request is serviced. The third line is the User-Agent header, and provides information about the program generating the request. The next line is the Host header, and provides the Host name and port on the server that is contacted to form the connection. The final line is the Accept header, which lists the media types the client can accept as valid responses.

HTTP server activity

The client/server nature of Web browsers is deceptively simple. To most users, retrieving information on the World Wide Web is a simple procedure: click on a link, and the information appears on the screen. More knowledgeable users have some understanding of the nature of HTML syntax and the client/server nature of the protocols used. This is usually sufficient for the production of simple, page-oriented Web site content. Authors of more complex Web pages have a wide variety of options to automate the collection and presentation of information using HTML.

Before building a Web server application, it is useful to understand how the client issues a request and how the server responds to client requests.

Composing client requests

When an HTML hypertext link is selected (or the user otherwise specifies a URL), the browser collects information about the protocol, the specified domain, the path to the information, the date and time, the operating environment, the browser itself, and other content information. It then composes a request.

For example, to display a page of images based on criteria selected by clicking buttons on a form, the client might construct this URL:

```
http://www.TSite.com/art/gallery.cgi/animals?animal=dog&color=black
```

which specifies an HTTP server in the www.TSite.com domain. The client contacts www.TSite.com, connects to the HTTP server, and passes it a request. The request might look something like this:

```
GET /art/gallery.cgi/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (Apache; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

Serving CGI requests

The Web server receives a client request and can perform any number of actions, based on its configuration. If the server is configured to recognize the /gallery.cgi portion of the request as a program, it passes information contained in the request directly to the CGI program. The server waits while the program executes. When the CGI program exits, it passes the HTML content back to the server via the stdout pipe.

In all cases, the program acts on the request of and performs actions specified by the programmer: accessing databases, doing simple table lookups or calculations, constructing or selecting HTML documents, and so on.

Serving dynamic shared object requests

When a Web server application finishes with a client request, it constructs a page of HTML code or other MIME content, and passes it back (via the server) to the client for display. When a dynamic shared object (DSO) finishes, it passes the HTML page and any response information directly back to the server, which passes them back to the client.
Creating a Web server application as a shared object reduces system load and resource use by reducing the number of processes and disk accesses necessary to service an individual request. For more information on DSOs, see:

```
http://httpd.apache.org/docs/dso.html
```

Web server applications

Web server applications extend the functionality and capability of existing Web servers. The Web server application receives HTTP request messages from the Web server, performs any actions requested in those messages, and formulates responses that it passes back to the Web server. Any operation that you can perform with a Kylix application can be incorporated into a Web server application.

Types of Web server applications

Using the Internet components, you can create two types of Web server applications. Each type uses a type-specific descendant of *TWebApplication*, *TWebRequest*, and *TWebResponse*:

Table 22.1 Web server application components

Application Type	Application Object	Request Object	Response Object
Console CGI application	*TCGIApplication*	*TCGIRequest*	*TCGIResponse*
Apache DSO Module	*TApacheApplication*	*TApacheRequest*	*TApacheResponse*

CGI stand-alone

A CGI stand-alone Web server application is a console application that receives client request information on standard input and passes the results back to the server on standard output. This data is evaluated by *TCGIApplication*, which creates *TCGIRequest* and *TCGIResponse* objects. Each request message is handled by a separate instance of the application.

Apache DSO module

Note For DSO support, you must download the source code and rebuild Apache. See "Compiling an Apache application for DSO support" on page 22-23.

An Apache DSO module is a library application. The Apache program loads a shared object in memory, and then reserves a memory block which is used to pass data between the Apache program and module. This data is evaluated by *TApacheApplication*, which creates *TApacheRequest* and *TApacheResponse* objects.

Creating Web server applications

All new Web server applications are created by selecting File | New from the menu of the main window and selecting Web Server Application in the New Items dialog. A dialog box appears, where you can select one of the Web server application types:

- CGI stand-alone: Selecting this type of application sets up your project as a console application and adds the required entries to the uses clause of the project file.
- Apache DSO module: Selecting this type of application sets up your project as a library application and adds the required entries to the uses clause of the project file.

Choose the type of Web Server Application that communicates with the type of Web Server your application will use. This creates a new project configured to use Internet components and containing an empty Web Module.

The Web module

The Web module (*TWebModule*) is a descendant of *TDataModule* and may be used in the same way: to provide centralized control for business rules and non-visual components in the Web application.

Add any content producers that your application uses to generate response messages. These can be the built-in content producers such as *TPageProducer* or descendants of *TCustomContentProducer* that you have written yourself. If your application generates response messages that include material drawn from databases, you can add data access components.

In addition to storing non-visual components and business rules, the Web module also acts as a dispatcher, matching incoming HTTP request messages to action items that generate the responses to those requests.

You may have a data module already that is set up with many of the non-visual components and business rules that you want to use in your Web application. You can replace the Web module with your pre-existing data module. Simply delete the automatically generated Web module and replace it with your data module. Then, add a *TWebDispatcher* component to your data module, so that it can dispatch request messages to action items, the way a Web module can. If you want to change the way action items are chosen to respond to incoming HTTP request messages, derive a new dispatcher component from *TCustomWebDispatcher*, and add that to the data module instead.

Your project can contain only one dispatcher. This can either be the Web module that is automatically generated when you create the project, or the *TWebDispatcher* component that you add to a data module that replaces the Web module. If a second data module containing a dispatcher is created during execution, the Web server application generates a runtime error.

Note The Web module that you set up at design time is actually a template.

The Web Application object

The project that is set up for your Web application contains a global variable named *Application*. *Application* is a descendant of *TWebApplication* (either *TApacheApplication* or *TCGIApplication*) that is appropriate to the type of application you are creating. It runs in response to HTTP request messages received by the Web server.

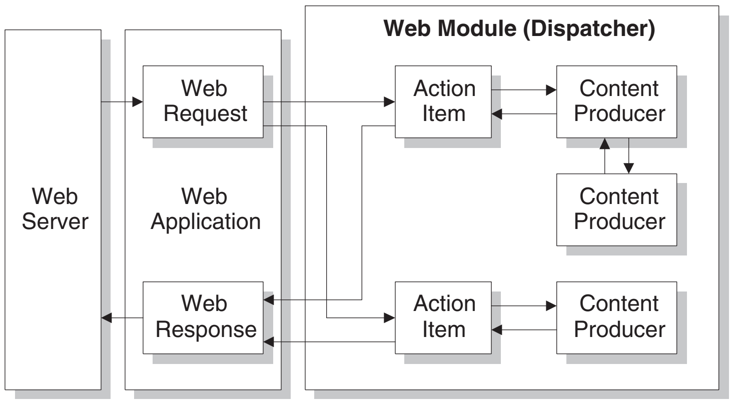
Warning Do not include the forms unit in the project **uses** clause after the CGIApp or ApacheApp unit. Forms also declares a global variable named *Application*, and if it appears after the CGIApp or ApacheApp unit, *Application* will be initialized to an object of the wrong type.

The structure of a Web server application

When the Web application receives an HTTP request message, it creates a *TWebRequest* object to represent the HTTP request message, and a *TWebResponse* object to represent the response that should be returned. The application then passes these objects to the Web dispatcher (either the Web module or a *TWebDispatcher* component).

The Web dispatcher controls the flow of the Web server application. The dispatcher maintains a collection of action items (*TWebActionItem*) that know how to handle certain types of HTTP request messages. The dispatcher identifies the appropriate action items or auto-dispatching components to handle the HTTP request message, and passes the request and response objects to the identified handler so that it can perform any requested actions or formulate a response message. It is described more fully in the section “The Web dispatcher” on page 22-7.

Figure 22.2 Structure of a Server Application



The action items are responsible for reading the request and assembling a response message. Specialized content producer components aid the action items in dynamically generating the content of response messages, which can include custom HTML code or other MIME content. The content producers can make use of other content producers or descendants of *THTMLTagAttributes*, to help them create the content of the response message. For more information on content producers, see "Generating the content of response messages" on page 22-16.

When all action items (or auto-dispatching components) have finished creating the response by filling out the *TWebResponse* object, the dispatcher passes the result back to the Web application. The application sends the response on to the client via the Web server.

The Web dispatcher

If you are using a Web module, it acts as a Web dispatcher. If you are using a pre-existing data module, you must add a single dispatcher component (*TWebDispatcher*) to that data module. The dispatcher maintains a collection of action items that know how to handle certain kinds of request messages. When the Web application passes a request object and a response object to the dispatcher, it chooses one or more action items to respond to the request.

Adding actions to the dispatcher

Open the action editor from the Object Inspector by clicking the ellipsis on the *Actions* property of the dispatcher. Action items can be added to the dispatcher by clicking the Add button in the action editor.

Add actions to the dispatcher to respond to different request methods or target URIs. You can set up your action items in a variety of ways. You can start with action items that preprocess requests, and end with a default action that checks whether the response is complete and either sends the response or returns an error code. Or, you can add a separate action item for every type of request, where each action item completely handles the request.

Action items are discussed in further detail in "Action items" on page 22-8.

Dispatching request messages

When the dispatcher receives the client request, it generates a *BeforeDispatch* event. This provides your application with a chance to preprocess the request message before it is seen by any of the action items.

Next, the dispatcher looks through its list of action items for one that matches the pathinfo portion of the request message's target URL and that can provide the service specified as the method of the request message. It does this by comparing the *PathInfo* and *MethodType* properties of the *TWebRequest* object with the properties of the same name on the action item.

When the dispatcher finds an appropriate action item, it causes that action item to fire. When the action item fires, it does one of the following:

- Fills in the response content and sends the response or signals that the request is completely handled.
- Adds to the response and then allows other action items to complete the job.
- Defers the request to other action items.

If, after checking all the action items, the request message has still not been fully handled, the dispatcher calls the default action item. The default action item does not need to match either the target URL or the method of the request.

If the dispatcher reaches the end of the action list (including the default action, if any) and no actions have been triggered, nothing is passed back to the server. The server simply drops the connection to the client.

If the request is handled by the action items, the dispatcher generates an *AfterDispatch* event. This provides a final opportunity for your application to check the response that was generated, and make any last minute changes.

Action items

Each action item (*TWebActionItem*) performs a specific task in response to a given type of request message.

Action items can completely respond to a request or perform part of the response and allow other action items to complete the job. Action items can send the HTTP response message for the request, or simply set up part of the response for other action items to complete. If a response is completed by the action items but not sent, the Web server application sends the response message.

Determining when action items fire

Most properties of the action item determine when the dispatcher selects it to handle an HTTP request message. To set the properties of an action item, you must first bring up the action editor: select the *Actions* property of the dispatcher in the Object Inspector and click on the ellipsis. When an action is selected in the action editor, its properties can be modified in the Object Inspector.

The target URL

The dispatcher compares the *PathInfo* property of an action item to the *PathInfo* of the request message. The value of this property should be the path information portion of the URL for all requests that the action item is prepared to handle. For example, given this URL,

```
http://www.TSite.com/art/gallery.cgi/mammals?animal=dog&color=black
```

and assuming that the `/gallery.cgi` part indicates the Web server application, the path information portion is

```
/mammals
```

Use path information to indicate where your Web application should look for information when servicing requests, or to divide you Web application into logical subservices.

The request method type

The *MethodType* property of an action item indicates what type of request messages it can process. The dispatcher compares the *MethodType* property of an action item to the *MethodType* of the request message. *MethodType* can take one of the following values:

Table 22.2 MethodType values

Value	Meaning
mtGet	The request is asking for the information associated with the target URI to be returned in a response message.
mtHead	The request is asking for the header properties of a response, as if servicing an *mtGet* request, but omitting the content of the response.
mtPost	The request is providing information to be posted to the Web application.
mtPut	The request asks that the resource associated with the target URI be replaced by the content of the request message.
mtAny	Matches any request method type, including *mtGet*, *mtHead*, *mtPut*, and *mtPost*.

Enabling and disabling action items

Each action item has an *Enabled* property that can be used to enable or disable that action item. By setting *Enabled* to *False*, you disable the action item so that it is not considered by the dispatcher when it looks for an action item to handle a request.

A *BeforeDispatch* event handler can control which action items should process a request by changing the *Enabled* property of the action items before the dispatcher begins matching them to the request message.

Caution Changing the *Enabled* property of an action during execution may cause unexpected results for subsequent requests. Use the *BeforeDispatch* event to ensure that all action items are correctly initialized to their appropriate starting states.

Choosing a default action item

Only one of the action items can be the default action item. The default action item is selected by setting its *Default* property to *True*. When the *Default* property of an action item is set to *True*, the *Default* property for the previous default action item (if any) is set to *False*.

When the dispatcher searches its list of action items to choose one to handle a request, it stores the name of the default action item. If the request has not been fully handled when the dispatcher reaches the end of its list of action items, it executes the default action item.

The dispatcher does not check the *PathInfo* or *MethodType* of the default action item. The dispatcher does not even check the *Enabled* property of the default action item. Thus, you can make sure the default action item is only called at the very end by setting its *Enabled* property to *False*.

The default action item should be prepared to handle any request that is encountered, even if it is only to return an error code indicating an invalid URI or *MethodType*. If the default action item does not handle the request, no response is sent to the Web client.

Caution Changing the *Default* property of an action during execution may cause unexpected results for the current request. If the *Default* property of an action that has already been triggered is set to *True*, that action will not be re-evaluated and the dispatcher will not trigger that action when it reaches the end of the action list.

Responding to request messages with action items

The real work of the Web server application is performed by action items when they execute. When the Web dispatcher fires an action item, that action item can respond to the current request message in two ways:

- If the action item has an associated producer component as the value of its *Producer* property, that producer automatically assigns the *Content* of the response message using its *Content* method. The Internet page of the component palette includes a number of content producer components that can help construct an HTML page for the content of the response message.
- After the producer has assigned any response content (if there is an associated producer), the action item receives an *OnAction* event. The *OnAction* event handler is passed the *TWebRequest* object that represents the HTTP request message and a *TWebResponse* object to fill with any response information.

If the action item's content can be generated by a single content producer, it is simplest to assign the content producer as the action item's *Producer* property. However, you can always access any content producer from the *OnAction* event handler as well. The *OnAction* event handler allows more flexibility, so that you can use multiple content producers, assign response message properties, and so on.

Both the content-producer component and the *OnAction* event handler can use any objects or runtime library methods to respond to request messages. They can access databases, perform calculations, construct or select HTML documents, and so on. For more information about generating response content using content-producer components, see "Generating the content of response messages" on page 22-16.

Sending the response

An *OnAction* event handler can send the response back to the Web client by using the methods of the *TWebResponse* object. However, if no action item sends the response to the client, it will still get sent by the Web server application as long as the last action item to look at the request indicates that the request was handled.

Using multiple action items

You can respond to a request from a single action item, or divide the work up among several action items. If the action item does not completely finish setting up the response message, it must signal this state in the *OnAction* event handler by setting the *Handled* parameter to *False*.

If many action items divide up the work of responding to request messages, each setting *Handled* to *False* so that others can continue, make sure the default action item leaves the *Handled* parameter set to *True*. Otherwise, no response will be sent to the Web client.

When dividing the work among several action items, either the *OnAction* event handler of the default action item or the *AfterDispatch* event handler of the dispatcher should check whether all the work was done and set an appropriate error code if it is not.

Accessing client request information

When an HTTP request message is received by the Web server application, the headers of the client request are loaded into the properties of a *TWebRequest* object. In Apache DSO applications, the request message is encapsulated by a *TApacheRequest* object. Console CGI applications use *TCGIRequest* object.

The properties of the request object are read-only. You can use them to gather all of the information available in the client request.

Properties that contain request header information

Most properties in a request object contain information about the request that comes from the HTTP request header. Not every request supplies a value for every one of these properties. Also, some requests may include header fields that are not surfaced in a property of the request object, especially as the HTTP standard continues to evolve. To obtain the value of a request header field that is not surfaced as one of the properties of the request object, use the *GetFieldByName* method.

Properties that identify the target

The full target of the request message is given by the *URL* property. Usually, this is a URL that can be broken down into the protocol (HTTP), *Host* (server system), *ScriptName* (server application), *PathInfo* (location on the host), and *Query*.

Each of these pieces is surfaced in its own property. The protocol is always HTTP, and the *Host* and *ScriptName* identify the Web server application. The dispatcher uses the *PathInfo* portion when matching action items to request messages. The *Query* is used by some requests to specify the details of the requested information. Its value is also parsed for you as the *QueryFields* property.

Properties that describe the Web client

The request also includes several properties that provide information about where the request originated. These include everything from the e-mail address of the sender (the *From* property), to the URI where the message originated (the *Referer* or *RemoteHost* property). If the request contains any content, and that content does not arise from the same URI as the request, the source of the content is given by the *DerivedFrom* property. You can also determine the IP address of the client (the *RemoteAddr* property), and the name and version of the application that sent the request (the *UserAgent* property).

Properties that identify the purpose of the request

The *Method* property is a string describing what the request message is asking the server application to do. The HTTP 1.1 standard defines the following methods:

Value	What the message requests
OPTIONS	Information about available communication options.
GET	Information identified by the *URL* property.
HEAD	Header information from an equivalent GET message, without the content of the response.
POST	The server application to post the data included in the *Content* property, as appropriate.
PUT	The server application to replace the resource indicated by the *URL* property with the data included in the *Content* property.
DELETE	The server application to delete or hide the resource identified by the *URL* property.
TRACE	The server application to send a loop-back to confirm receipt of the request.

The *Method* property may indicate any other method that the Web client requests of the server.

The Web server application does not need to provide a response for every possible value of *Method*. The HTTP standard does require that it service both GET and HEAD requests, however.

The *MethodType* property indicates whether the value of *Method* is GET (mtGet), HEAD (mtHead), POST (mtPost), PUT (mtPut) or some other string (mtAny). The dispatcher matches the value of the *MethodType* property with the *MethodType* of each action item.

Properties that describe the expected response

The *Accept* property indicates the media types the Web client will accept as the content of the response message. The *IfModifiedSince* property specifies whether the client only wants information that has changed recently. The *Cookie* property includes state information (usually added previously by your application) that can modify the response.

Properties that describe the content

Most requests do not include any content, as they are requests for information. However, some requests, such as POST requests, provide content that the Web server application is expected to use. The media type of the content is given in the *ContentType* property, and its length in the *ContentLength* property. If the content of the message was encoded (for example, for data compression), this information is in the *ContentEncoding* property. The name and version number of the application that produced the content is specified by the *ContentVersion* property. The *Title* property may also provide information about the content.

The content of HTTP request messages

In addition to the header fields, some request messages include a content portion that the Web server application should process in some way. For example, a POST request might include information that should be added to a database maintained by the Web server application.

The unprocessed value of the content is given by the *Content* property. If the content can be parsed into fields separated by ampersands (&), a parsed version is available in the *ContentFields* property.

Creating HTTP response messages

When the Web server application creates a *TWebRequest* object for an incoming HTTP request message, it also creates a corresponding *TWebResponse* object to represent the response message that will be sent in return. In Apache DSO applications, the response message is encapsulated by a *TApacheResponse* object. Console CGI applications use *TCGIResponse* objects.

The action items that generate the response to a Web client request fill in the properties of the response object. In some cases, this may be as simple as returning an error code or redirecting the request to another URI. In other cases, this may involve complicated calculations that require the action item to fetch information from other sources and assemble it into a finished form. Most request messages require some response, even if it is only the acknowledgment that a requested action was carried out.

Filling in the response header

Most of the properties of the *TWebResponse* object represent the header information of the HTTP response message that is sent back to the Web client. An action item sets these properties from its *OnAction* event handler.

Not every response message needs to specify a value for every one of the header properties. The properties that should be set depend on the nature of the request and the status of the response.

Indicating the response status

Every response message must include a status code that indicates the status of the response. You can specify the status code by setting the *StatusCode* property. The HTTP standard defines a number of standard status codes with predefined meanings. In addition, you can define your own status codes using any of the unused possible values.

Each status code is a three-digit number where the most significant digit indicates the class of the response, as follows:

- 1xx: Informational (The request was received but has not been fully processed).
- 2xx: Success (The request was received, understood, and accepted).
- 3xx: Redirection (Further action by the client is needed to complete the request).
- 4xx: Client Error (The request cannot be understood or cannot be serviced).
- 5xx: Server Error (The request was valid but the server could not handle it).

Associated with each status code is a string that explains the meaning of the status code. This is given by the *ReasonString* property. For predefined status codes, you do not need to set the *ReasonString* property. If you define your own status codes, you should also set the *ReasonString* property.

Indicating the need for client action

When the status code is in the 300-399 range, the client must perform further action before the Web server application can complete its request. If you need to redirect the client to another URI, or indicate that a new URI was created to handle the request, set the *Location* property. If the client must provide a password before you can proceed, set the *WWWAuthenticate* property.

Describing the server application

Some of the response header properties describe the capabilities of the Web server application. The *Allow* property indicates the methods to which the application can respond. The *Server* property gives the name and version number of the application used to generate the response. The *Cookies* property can hold state information about the client's use of the server application which is included in subsequent request messages.

Describing the content

Several properties describe the content of the response. *ContentType* gives the media type of the response, and *ContentVersion* is the version number for that media type. *ContentLength* gives the length of the response. If the content is encoded (such as for data compression), indicate this with the *ContentEncoding* property. If the content came from another URI, this should be indicated in the *DerivedFrom* property. If the value of the content is time-sensitive, the *LastModified* property and the *Expires* property indicate whether the value is still valid. The *Title* property can provide descriptive information about the content.

Setting the response content

For some requests, the response to the request message is entirely contained in the header properties of the response. In most cases, however, action item assigns some content to the response message. This content may be static information stored in a file, or information that was dynamically produced by the action item or its content producer.

You can set the content of the response message by using either the *Content* property or the *ContentStream* property.

The *Content* property is a string. Kylix strings are not limited to text values, so the value of the *Content* property can be a string of HTML commands, graphics content such as a bit-stream, or any other MIME content type.

Use the *ContentStream* property if the content for the response message can be read from a stream. For example, if the response message should send the contents of a file, use a *TFileStream* object for the *ContentStream* property. As with the *Content* property, *ContentStream* can provide a string of HTML commands or other MIME content type. If you use the *ContentStream* property, do not free the stream yourself. The Web response object automatically frees it for you.

Note If the value of the *ContentStream* property is not **nil**, the *Content* property is ignored.

Sending the response

If you are sure there is no more work to be done in response to a request message, you can send a response directly from an *OnAction* event handler. The response object provides two methods for sending a response: *SendResponse* and *SendRedirect*. Call *SendResponse* to send the response using the specified content and all the header properties of the *TWebResponse* object. If you only need to redirect the Web client to another URI, the *SendRedirect* method is more efficient.

If none of the event handlers send the response, the Web application object sends it after the dispatcher finishes. However, if none of the action items indicate that they have handled the response, the application will close the connection to the Web client without sending any response.

Generating the content of response messages

Kylix provides a number of objects to assist your action items in producing content for HTTP response messages. You can use these objects to generate strings of HTML commands that are saved in a file or sent directly back to the Web client. You can write your own content producers, deriving them from *TCustomContentProducer* or one of its descendants.

TCustomContentProducer provides a generic interface for creating any MIME type as the content of an HTTP response message. Its descendants include page producers and table producers:

- Page producers scan HTML documents for special tags that they replace with customized HTML code. They are described in the following section.
- Table producers create HTML commands based on the information in a dataset. They are described in “Using database information in responses” on page 22-20.

Using page producer components

Page producers (*TPageProducer* and its descendants) take an HTML template and convert it by replacing special HTML-transparent tags with customized HTML code. You can store a set of standard response templates that are filled in by page producers when you need to generate the response to an HTTP request message. You can chain page producers together to iteratively build up an HTML document by successive refinement of the HTML-transparent tags.

HTML templates

An HTML template is a sequence of HTML commands and HTML-transparent tags. An HTML-transparent tag has the form

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

The angle brackets (< and >) define the entire scope of the tag. A pound sign (#) immediately follows the opening angle bracket (<) with no spaces separating it from the angle bracket. The pound sign identifies the string to the page producer as an HTML-transparent tag. The tag name immediately follows the pound sign with no spaces separating it from the pound sign. The tag name can be any valid identifier and identifies the type of conversion the tag represents.

Following the tag name, the HTML-transparent tag can optionally include parameters that specify details of the conversion to be performed. Each parameter is of the form *ParamName=Value*, where there is no space between the parameter name, the equals symbol (=) and the value. The parameters are separated by whitespace.

The angle brackets (< and >) make the tag transparent to HTML browsers that do not recognize the #TagName construct.

While you can create your own HTML-transparent tags to represent any kind of information processed by your page producer, there are several predefined tag names associated with values of the *TTag* data type. These predefined tag names correspond to HTML commands that are likely to vary over response messages. They are listed in the following table:

Tag Name	TTag value	What the tag should be converted to
Link	*tgLink*	A hypertext link. The result is an HTML sequence beginning with an <A> tag and ending with an </A> tag.
Image	*tgImage*	A graphic image. The result is an HTML <IMG> tag.
Table	*tgTable*	An HTML table. The result is an HTML sequence beginning with a <TABLE> tag and ending with a </TABLE> tag.
ImageMap	*tgImageMap*	A graphic image with associated hot zones. The result is an HTML sequence beginning with a <MAP> tag and ending with a </MAP> tag.

Any other tag name is associated with *tgCustom*. The page producer supplies no built-in processing of the predefined tag names. They are simply provided to help applications organize the conversion process into many of the more common tasks.

Note The predefined tag names are case insensitive.

Specifying the HTML template

Page producers provide you with many choices in how to specify the HTML template. You can set the *HTMLFile* property to the name of a file that contains the HTML template. You can set the *HTMLDoc* property to a *TStrings* object that contains the HTML template. If you use either the *HTMLFile* property or the *HTMLDoc* property to specify the template, you can generate the converted HTML commands by calling the *Content* method.

In addition, you can call the *ContentFromString* method to directly convert an HTML template that is a single string which is passed in as a parameter. You can also call the *ContentFromStream* method to read the HTML template from a stream. Thus, for example, you could store all your HTML templates in a memo field in a database, and use the *ContentFromStream* method to obtain the converted HTML commands, reading the template directly from a *TBlobStream* object.

Converting HTML-transparent tags

The page producer converts the HTML template when you call one of its *Content* methods. When the *Content* method encounters an HTML-transparent tag, it triggers the *OnHTMLTag* event. You must write an event handler to determine the type of tag encountered, and to replace it with customized content.

If you do not create an *OnHTMLTag* event handler for the page producer, HTML-transparent tags are replaced with an empty string.

Using page producers from an action item

A typical use of a page producer component uses the *HTMLFile* property to specify a file containing an HTML template. The *OnAction* event handler calls the *Content* method to convert the template into a final HTML sequence:

```
procedure WebModule1.MyActionEventHandler(Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
  PageProducer1.HTMLFile := 'Greeting.html';
  Response.Content := PageProducer1.Content;
end;
```

Greeting.html is a file that contains this HTML template:

```
<HTML>
<HEAD><TITLE>Our brand new web site</TITLE></HEAD>
<BODY>
Hello <#UserName>!  Welcome to our web site.
</BODY>
</HTML>
```

The *OnHTMLTag* event handler replaces the custom tag (`<#UserName>`) in the HTML during execution:

```
procedure WebModule1.PageProducer1HTMLTag(Sender : TObject;Tag: TTag;
   const TagString: string; TagParams: TStrings; var ReplaceText: string);
begin
  if CompareText(TagString,'UserName') = 0 then
    ReplaceText := TPageProducer(Sender).Dispatcher.Request.Content;
end;
```

If the content of the request message was the string *Mr. Ed*, the value of *Response.Content* would be

```
<HTML>
<HEAD><TITLE>Our brand new web site</TITLE></HEAD>
<BODY>
Hello Mr. Ed!  Welcome to our web site.
</BODY>
</HTML>
```

Note This example uses an *OnAction* event handler to call the content producer and assign the content of the response message. You do not need to write an *OnAction* event handler if you assign the page producer's *HTMLFile* property at design time. In that case, you can simply assign *PageProducer1* as the value of the action item's *Producer* property to accomplish the same effect as the *OnAction* event handler above.

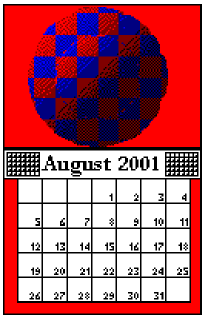
Chaining page producers together

The replacement text from an *OnHTMLTag* event handler need not be the final HTML sequence you want to use in the HTTP response message. You may want to use several page producers, where the output from one page producer is the input for the next.

The simplest way is to chain the page producers together is to associate each page producer with a separate action item, where all action items have the same *PathInfo* and *MethodType*. The first action item sets the content of the Web response message from its content producer, but its *OnAction* event handler makes sure the message is not considered handled. The next action item uses the *ContentFromString* method of its associated producer to manipulate the content of the Web response message, and so on. Action items after the first one use an *OnAction* event handler such as the following:

```
procedure WebModule1.Action2Action(Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageProducer2.ContentFromString(Response.Content);
end;
```

For example, consider an application that returns calendar pages in response to request messages that specify the month and year of the desired page. Each calendar page contains a picture, followed by the name and year of the month between small images of the previous month and next months, followed by the actual calendar. The resulting image looks something like this:



The general form of the calendar is stored in a template file. It looks like this:

```
<HTML>
<Head></HEAD>
<BODY>
<#MonthlyImage> <#TitleLine><#MainBody>
</BODY>
</HTML>
```

The *OnHTMLTag* event handler of the first page producer looks up the month and year from the request message. Using that information and the template file, it does the following:

- Replaces <#MonthlyImage> with <#Image Month=August Year=2001>.
- Replaces <#TitleLine> with <#Calendar Month=July Year=2001 Size=Small> August 2001 <#Calendar Month=September Year=2000 Size=Small>.
- Replaces <#MainBody> with <#Calendar Month=August Year=2001 Size=Large>.

The *OnHTMLTag* event handler of the next page producer uses the content produced by the first page producer, and replaces the <#Image Month=August Year=2001> tag with the appropriate HTML <IMG> tag. Yet another page producer resolves the #Calendar tags with appropriate HTML tables.

Using database information in responses

The response to an HTTP request message may include information taken from a database. Specialized content producers on the Internet palette page can generate the HTML to represent the records from a database in an HTML table.

Adding a session to the Web module

Console CGI applications and Apache DSO applications are launched in response to HTTP request messages. When working with databases in these types of applications, you can use the default session to manage your database connections, because each request message has its own instance of the application.

Representing database information in HTML

Specialized Content producer components on the Internet palette page supply HTML commands based on the records of a dataset. There are two types of data-aware content producers:

- The dataset page producer, which formats the fields of a dataset into the text of an HTML document.
- Table producers, which format the records of a dataset as an HTML table.

Using dataset page producers

Dataset page producers work like other page producer components: they convert a template that includes HTML-transparent tags into a final HTML representation. They include the special ability, however, of converting tags that have a tagname which matches the name of a field in a dataset into the current value of that field. For more information about using page producers in general, see "Using page producer components" on page 22-16.

To use a dataset page producer, add a *TDataSetPageProducer* component to your web module and set its *DataSet* property to the dataset whose field values should be displayed in the HTML content. Create an HTML template that describes the output of your dataset page producer. For every field value you want to display, include a tag of the form

```
<#FieldName>
```

in the HTML template, where *FieldName* specifies the name of the field in the dataset whose value should be displayed.

When your application calls the *Content*, *ContentFromString*, or *ContentFromStream* method, the dataset page producer substitutes the current field values for the tags that represent fields.

Using table producers

The Internet palette page includes two components that create an HTML table to represent the records of a dataset:

- Dataset table producers, which format the fields of a dataset into the text of an HTML document.
- Query table producers, which runs a query after setting parameters supplied by the request message and formats the resulting dataset as an HTML table.

Using either of the two table producers, you can customize the appearance of a resulting HTML table by specifying properties for the table's color, border, separator thickness, and so on. To set the properties of a table producer at design time, double-click the table producer component to display the Response Editor dialog.

Specifying the table attributes

Table producers use the *THTMLTableAttributes* object to describe the visual appearance of the HTML table that displays the records from the dataset. The *THTMLTableAttributes* object includes properties for the table's width and spacing within the HTML document, and for its background color, border thickness, cell padding, and cell spacing. These properties are all turned into options on the HTML <TABLE> tag created by the table producer.

At design time, specify these properties using the Object Inspector. Select the table producer object in the Object Inspector and expand the *TableAttributes* property to access the display properties of the *THTMLTableAttributes* object.

You can also specify these properties programmatically at runtime.

Specifying the row attributes

Similar to the table attributes, you can specify the alignment and background color of cells in the rows of the table that display data. The *RowAttributes* property is a *THTMLTableRowAttributes* object.

At design time, specify these properties using the Object Inspector by expanding the *RowAttributes* property. You can also specify these properties programmatically at runtime.

You can also adjust the number of rows shown in the HTML table by setting the *MaxRows* property.

Specifying the columns

If you know the dataset for the table at design time, you can use the Columns editor to customize the columns' field bindings and display attributes. Select the table producer component, and right-click. From the context menu, choose the Columns editor. This lets you add, delete, or rearrange the columns in the table. You can set the field bindings and display properties of individual columns in the Object Inspector after selecting them in the Columns editor.

If you are getting the name of the dataset from the HTTP request message, you can't bind the fields in the Columns editor at design time. However, you can still customize the columns programmatically at runtime, by setting up the appropriate *THTMLTableColumn* objects and using the methods of the *Columns* property to add them to the table. If you do not set up the *Columns* property, the table producer creates a default set of columns that match the fields of the dataset and specify no special display characteristics.

Embedding tables in HTML documents

You can embed the HTML table that represents your dataset in a larger document by using the *Header* and *Footer* properties of the table producer. Use *Header* to specify everything that comes before the table, and *Footer* to specify everything that comes after the table.

You may want to use another content producer (such as a page producer) to create the values for the *Header* and *Footer* properties.

If you embed your table in a larger document, you may want to add a caption to the table. Use the *Caption* and *CaptionAlignment* properties to give your table a caption.

Setting up a dataset table producer

TDataSetTableProducer is a table producer that creates an HTML table for a dataset. Set the *DataSet* property of *TDataSetTableProducer* to specify the dataset that contains the records you want to display. You do not set the *DataSource* property, as you would for most data-aware objects in a conventional database application. This is because *TDataSetTableProducer* generates its own data source internally.

You can set the value of *DataSet* at design time if your Web application always displays records from the same dataset. You must set the *DataSet* property at runtime if you are basing the dataset on the information in the HTTP request message.

Setting up a query table producer

You can produce an HTML table to display the results of a query, where the parameters of the query come from the HTTP request message. Specify the *TSQLQuery* object that uses those parameters as the *Query* property of a *TSQLQueryTableProducer* component.

If the request message is a GET request, the parameters of the query come from the *Query* fields of the URL that was given as the target of the HTTP request message. If the request message is a POST request, the parameters of the query come from the content of the request message.

When you call the *Content* method of *TSQLQueryTableProducer*, it runs the query, using the parameters it finds in the request object. It then formats an HTML table to display the records in the resulting dataset.

As with any table producer, you can customize the display properties or column bindings of the HTML table, or embed the table in a larger HTML document.

Debugging server applications

Debugging Web server applications presents some unique problems, because they run in response to messages from a Web server. You can not simply launch your application from the IDE, because that leaves the Web server out of the loop, and your application will not find the request message it is expecting. How you debug your Web server application depends on its type.

Debugging CGI applications

It is more difficult to debug CGI applications, because the application itself must be launched by the Web server.

Debugging as a shared object

Another approach you can take with CGI applications is first to create and debug your application as an Apache DSO application. Once your Apache DSO application is working smoothly, convert it to a CGI application. To convert your application, use the following steps:

1. Right-click the Web module and choose Add To Repository.
2. In the Add To Repository dialog box, give your Web module a title, text description, Repository page (probably Data Modules), author name, and icon.
3. Choose OK to save your Web module as a template.
4. From the main menu, choose File|New and select Web Server Application. In the New Web Server Application dialog box, choose CGI, as appropriate.
5. Delete the automatically generated Web module.
6. From the main menu, choose File|New and select the template you saved in step 3. This will be on the page you specified in step 2.

Debugging Apache DSO applications

Before you debug an Apache DSO application, you must first compile it.

Compiling an Apache application for DSO support

For the most current information, see the online document from the Apache Web site:

```
http://httpd.apache.org/docs/dso.html
```

To compile your application:

1. Download the Apache source code.

   Currently, you can use:

   ```
   http://httpd.apache.org/dist/apache_1.3.14.tar.gz
   ```

2. Extract the downloaded source tar.

3 Go to the root of the extracted directory, such as: /usr/apache_1.3.14

4 If the config.status file does not exist, create it and add the following lines:

```
#!/bin/sh
##
##  config.status -- APACI auto-generated configuration restore script
##
##  Use this shell script to re-run the APACI configure script for
##  restoring your configuration. Additional parameters can be supplied.
##


CFLAGS="-g" \
CFLAGS_SHLIB="-g" \
LIBS="/usr/lib/libpthread.so" \
./configure \
"--with-layout=Apache" \
"--enable-module=so" \
"--enable-rule=SHARED_CORE" \
"$@"
```

Note You must add the `LIBS="/usr/lib/libpthread.so" \` line to your config.status file.

5 At the command prompt ($=prompt), type the following lines:

```
$ chmod 755 config.status
$ ./config.status
$ make
$ make install
```

This configures Apache for DSO support installing it into the target directory.

Debugging Apache DSO applications

You can debug Apache applications by launching the host application from within the IDE by setting the run parameters.

You should be running Kylix as the root user to debug your Apache DSO application. Make sure that Apache is first compiled for DSO support with `libpthread.so`.

1 Set up the run parameters to launch the host application:
- Choose Run | Parameters.
- Set host application to /usr/local/apache/bin/httpd.
- Set parameters to -X.

2 Set the output directory.
- Choose Project | Options | Directories
- Set the output directory to ..../apache/libexec.

3 Set your breakpoint.

4 Build the project.

5 Run the application.

6 Open a Web browser and request a URL that will invoke your module.

Chapter

23

Working with sockets

This chapter describes the socket components that let you create an application that can communicate with other systems using TCP/IP and related protocols. Using sockets, you can read and write over connections to other machines without worrying about the details of the underlying networking software. Sockets provide connections based on the TCP/IP protocol, but are sufficiently general to work with related protocols such as Xerox Network System (XNS), Digital's DECnet, or Novell's IPX/SPX family.

Using sockets, you can write network servers or client applications that read from and write to other systems. A server or client application is usually dedicated to a single service such as Hypertext Transfer Protocol (HTTP) or File Transfer Protocol (FTP). Using server sockets, an application that provides one of these services can link to client applications that want to use that service. Client sockets allow an application that uses one of these services to link to server applications that provide the service.

Implementing services

Sockets provide one of the pieces you need to write network servers or client applications. For many services, such as HTTP or FTP, third party servers are readily available. Some are even bundled with the operating system, so that there is no need to write one yourself. However, when you want more control over the way the service is implemented, a tighter integration between your application and the network communication, or when no server is available for the particular service you need, then you may want to create your own server or client application. For example, when working with distributed data sets, you may want to write a layer to communicate with databases on other systems.

Understanding service protocols

Before you can write a network server or client, you must understand the service that your application is providing or using. Many services have standard protocols that your network application must support. If you are writing a network application for a standard service such as HTTP, FTP, or even finger or time, you must first understand the protocols used to communicate with other systems. See the documentation on the particular service you are providing or using.

If you are providing a new service for an application that communicates with other systems, the first step is designing the communication protocol for the servers and clients of this service. What messages are sent? How are these messages coordinated? How is the information encoded?

Communicating with applications

Often, your network server or client application provides a layer between the networking software and an application that uses the service. For example, an HTTP server sits between the Internet and a Web server application that provides content and responds to HTTP request messages.

Sockets provide the interface between your network server or client application and the networking software. You must provide the interface between your application and the applications that use it.

Services and ports

Most standard services are associated, by convention, with specific port numbers. We will discuss port numbers in greater detail later. For now, consider the port number a numeric code for the service.

If you are implementing a standard service, the *TTcpClient*) socket component provides methods for you to look up the port number for the service. If you are providing a new service, you can specify the associated port number in the /etc/services file. The services file is an ASCII file that lists Internet network services, including a name, port number, and protocol type. See the services man page for more information on setting up a services file.

Types of socket connections

Socket connections can be divided into three basic types, which reflect how the connection was initiated and what the local socket is connected to. These are

- Client connections.
- Listening connections.
- Server connections.

Once the connection to a client socket is completed, the server connection is indistinguishable from a client connection. Both end points have the same capabilities and receive the same types of events. Only the listening connection is fundamentally different, as it has only a single endpoint.

Client connections

Client connections connect a client socket on the local system to a server socket on a remote system. Client connections are initiated by the client socket. First, the client socket must describe the server socket it wishes to connect to. The client socket then looks up the server socket and, when it locates the server, requests a connection. The server socket may not complete the connection right away. Server sockets maintain a queue of client requests, and complete connections as they find time. When the server socket accepts the client connection, it sends the client socket a full description of the server socket to which it is connecting, and the connection is completed by the client.

Listening connections

Server sockets do not locate clients. Instead, they form passive "half connections" that listen for client requests. Server sockets associate a queue with their listening connections; the queue records client connection requests as they come in. When the server socket accepts a client connection request, it forms a new socket to connect to the client, so that the listening connection can remain open to accept other client requests.

Server connections

Server connections are formed by server sockets when a listening socket accepts a client request. A description of the server socket that completes the connection to the client is sent to the client when the server accepts the connection. The connection is established when the client socket receives this description and completes the connection.

Describing sockets

Sockets let your network application communicate with other systems over the network. Each socket can be viewed as an endpoint in a network connection. It has an address that specifies

- The system on which it is running.
- The types of interfaces it understands.
- The port it is using for the connection.

A full description of a socket connection includes the addresses of the sockets on both ends of the connection. You can describe the address of each socket endpoint by supplying both the IP address or host and the port number.

Before you can make a socket connection, you must fully describe the sockets that form its end points. Some of the information is available from the system your application is running on. For instance, you do not need to describe the local IP address of a client socket—this information is available from the operating system.

The information you must provide depends on the type of socket you are working with. Client sockets must describe the server they want to connect to. Listening server sockets must describe the port that represents the service they provide.

Describing the host

The host is the system that is running the application that contains the socket. You can describe the host for a socket by giving its IP address, which is a string of four numeric (byte) values in the standard Internet dot notation, such as

```
123.197.1.2
```

A single system may support more than one IP address.

IP addresses are often difficult to remember and easy to mistype. An alternative is to use the host name. Host names are aliases for the IP address that you often see in Uniform Resource Locators (URLs). They are strings containing a domain name and service, such as

```
http://www.wSite.Com
```

Most Intranets provide host names for the IP addresses of systems on the Internet. On Linux machines, if a host name is not available, you can create one for your local IP address by entering the name into the /etc/hosts file. See your Linux documentation on for more information on the /etc/hosts file.

Server sockets do not need to specify a host. The local IP address can be read from the system. If the local system supports more than one IP address, server sockets will listen for client requests on all IP addresses simultaneously. When a server socket accepts a connection, the client socket provides the remote IP address.

Client sockets must specify the remote host by providing either its host name or IP address.

Choosing between a host name and an IP address

Most applications use the host name to specify a system. Host names are easier to remember, and easier to check for typographical errors. Further, servers can change the system or IP address that is associated with a particular host name. Using a host name allows the client socket to find the abstract site represented by the host name, even when it has moved to a new IP address.

If the host name is unknown, the client socket must specify the server system using its IP address. Specifying the server system by giving the IP address is faster. When you provide the host name, the socket must search for the IP address associated with the host name, before it can locate the server system.

Using ports

While the IP address provides enough information to find the system on the other end of a socket connection, you also need a port number on that system. Without port numbers, a system could only form a single connection at a time. Port numbers are unique identifiers that enable a single system to host multiple connections simultaneously, by giving each connection a separate port number.

Earlier, we described port numbers as numeric codes for the services implemented by network applications. This is a convention that allows listening server connections to make themselves available on a fixed port number so that they can be found by client sockets. Server sockets listen on the port number associated with the service they provide. When they accept a connection to a client socket, they create a separate socket connection that uses a different, arbitrary, port number. This way, the listening connection can continue to listen on the port number associated with the service.

Client sockets use an arbitrary local port number, as there is no need for them to be found by other sockets. They specify the port number of the server socket to which they want to connect so that they can find the server application. Often, this port number is specified indirectly, by naming the desired service.

Using socket components

The Internet palette page includes socket components (client sockets and server sockets) that allow your network application to form connections to other machines, and that allow you to read and write information over that connection. Associated with each of these socket components are socket objects, which represent the endpoint of an actual socket connection. The socket components use the socket objects to encapsulate the operating system API calls, so that your application does not need to be concerned with the details of establishing the connection or managing the socket messages.

If you want to use the operating system API calls, or customize the details of the connections that the socket components make on your behalf, you can use the properties, events, and methods of the socket objects.

Using client sockets

Add a client socket component (*TTcpClient*) to your form or data module to turn your application into a TCP/IP client. Client sockets allow you to specify the server socket you want to connect to, and the service you want that server to provide. Once you have described the desired connection, you can use the client socket component to complete the connection to the server.

Specifying the target server

Client socket components have a number of properties that allow you to specify the server system and port to which you want to connect. Use the *RemoteHost* property to specify the remote host server by either its host name or IP address.

In addition to the host, you must specify the port on the server system that your client socket will connect to. You can use the *RemotePort* property to specify the server port number directly or indirectly by naming the target service.

Forming the connection

Once you have set the properties of your client socket component to describe the server you want to connect to, you can form the connection at runtime by calling the *Open* method. Or, if you want your application to form the connection automatically when it starts up, set the *Active* property to *True* at design time, using the Object Inspector.

Getting information about the connection

After completing the connection to a server socket, you can use the client socket object associated with your client socket component to obtain information about the connection. Use the *LocalHost* and *LocalPort* properties to determine the address and port number used by the client and server sockets to form the end points of the connection. You can use the *Handle* property to obtain a handle to the socket connection to use when making socket calls.

Closing the connection

When you have finished communicating with a server application over the socket connection, you can shut down the connection by calling the *Close* method. The connection may also be closed from the server end. If that is the case, you will receive notification in an *OnDisconnect* event.

Using server sockets

Add a server socket component (*TTcpServer*) to your form or data module to turn your application into a TCP/IP server. Server sockets allow you to specify the service you are providing or the port you want to use to listen for client requests. You can use the server socket component to listen for and accept client connection requests.

When a server socket component receives a connection request from the client, it creates a *TTcpClient* component. The *TTcpClient* component serves as an endpoint for the connection.

Specifying the port

Before your server socket can listen to client requests, you must specify the port that your server will listen on. You can specify this port using the *LocalPort* property. If your server application is providing a standard service that is associated by convention with a specific port number, you can also specify the service name using the *LocalPort* property. It is a good idea to use the service name instead of a port number, because it is easy to introduce typographical errors when specifying the port number.

Listening for client requests

Once you have set the port number of your server socket component, you can form a listening connection at runtime by calling the *Open* method. Or, if you want your application to form the listening connection automatically when it starts up, set the *Active* property to *True* at design time, using the Object Inspector.

Connecting to clients

A listening server socket component automatically accepts client connection requests when they are received if you set the *AutoAccept* property to *True*. You receive notification every time this occurs in an *OnAccept* event.

Closing server connections

When you want to shut down the listening connection, call the *Close* method or set the *Active* property to *False*. This shuts down all open connections to client applications, cancels any pending connections that have not been accepted, and then shuts down the listening connection so that your server socket component does not accept any new connections.

When clients shut down their individual connections to your server socket, you are informed by an *OnDisconnect* event.

Responding to socket events

When writing applications that use sockets, you can write or read to the socket anywhere in the program. You can write to the socket using the *SendBuf* method in your program after the socket has been opened. The *OnSend* and *RecvBuf* events are triggered every time something is written or read from the socket. They can be used for filtering. Every time you read or write, a read or write event is triggered.

Both client sockets and server sockets generate error events when they receive error messages from the connection.

Socket components also receive two events in the course of opening and completing a connection. If your application needs to influence how the opening of the socket proceeds. You must use the *SendBuf* and *RecvBuf* methods to respond to these client events or server events.

Error events

Both client and server sockets generate an *OnError* event when they receive error messages from the connection. You can write an *OnError* event handler to respond to these error messages. The event handler is passed information about

- What socket object received the error notification.

- What the socket was trying to do when the error occurred.
- The error code that was provided by the error message.

You can respond to the error in the event handler, and change the error code to 0 to prevent the socket from raising an exception.

Client events

When a client socket opens a connection, the following events occur:

- The socket is set up and initialized for event notification.
- An *OnCreateHandle* event occurs after the server and server socket is created. At this point, the socket object available through the *Handle* property can provide information about the server or client socket that will form the other end of the connection. This is the first chance to obtain the actual port used for the connection, which may differ from the port of the listening sockets that accepted the connection.
- The connection request is accepted by the server and completed by the client socket.
- When the connection is established, the *OnConnect* notification event occurs.

Server events

Server socket components form two types of connections: listening connections and connections to client applications. The server socket receives events during the formation of each of these connections.

Events when listening

Just before the listening connection is formed, the *OnCreateHandle* event occurs. You can use the *OnConnect* property to make changes to the socket before it is opened for listening. For example, if you want to restrict the IP addresses the server uses for listening, you would do that in an *OnCreateHandle* event handler.

Events with client connections

When a server socket accepts a client connection request, the following events occur:

- An *OnAccept* event occurs, passing in the new *TTcpClient* object to the event handler. This is the first point when you can use the properties of *TTcpClient* to obtain information about the server endpoint of the connection to a client.
- If *BlockMode* is *bmThreadBlocking* an *OnCreateServerSocketThread* event occurs. If you want to provide your own customized descendant of *TServerSocketThread*, you can create one in an *OnCreateServerSocketThread* event handler, and that will be used instead of *TServerSocketThread*.
- If *BlockMode* is *bmThreadBlocking*, an *OnCreateServerSocketThread* event occurs as the thread begins execution. If you want to perform any initialization of the

thread, or make any socket API calls before the thread starts reading or writing over the connection, use the *OnCreateServerSocketThread* event handler.

- The client completes the connection and an *OnAccept* event occurs. With a non-blocking server, you may want to start reading or writing over the socket connection at this point.

Reading and writing over socket connections

The reason you form socket connections to other machines is so that you can read or write information over those connections. What information you read or write, or when you read it or write it, depends on the service associated with the socket connection.

Reading and writing over sockets can occur asynchronously, so that it does not block the execution of other code in your network application. This is called a non-blocking connection. You can also form blocking connections, where your application waits for the reading or writing to be completed before executing the next line of code.

Non-blocking connections

Non-blocking connections read and write asynchronously, so that the transfer of data does not block the execution of other code in you network application. To create a non-blocking connection:

- On client sockets, set the *BlockMode* property to *bmNonBlocking.*
- On server sockets, set the *BlockMode* property to *bmNonBlocking*.

When the connection is non-blocking, reading and writing events inform your socket when the socket on the other end of the connection tries to read or write information.

Reading and writing events

Non-blocking sockets generate reading and writing events when it needs to read or write over the connection. With client sockets, you can respond to these notifications in an *OnRead* or *OnWrite* event handler. With server sockets, you can respond to these events in an *OnClientRead* or *OnClientWrite* event handler.

The socket object associated with the socket connection is provided as a parameter to the read or write event handlers. This socket object provides a number of methods to allow you to read or write over the connection.

To read from the socket connection, use the *ReceiveBuf* or *ReceiveText* method. Before using the *ReceiveBuf* method, you can use the *ReceiveLength* method to determine the number of bytes currently waiting in the local buffer. Note, however, that *ReceiveLength* returns a value which is valid at the time it is called, but additional data may arrive after the call to *ReceiveLength* and before the call to *ReceiveBuf*. If additional data does arrive during that time, your code may not receive a corresponding *OnRead* event. Therefore, you may want to call *ReceiveLength* repeatedly to ensure that no additional data is waiting.

To write to the socket connection, use the *SendBuf*, *SendStream*, or *SendText* method. If you have no more need of the socket connection after you have written your information over the socket, you can use the *SendStreamThenDrop* method. *SendStreamThenDrop* closes the socket connection after writing all information that can be read from the stream. If you use the *SendStream* or *SendStreamThenDrop* method, do not free the stream object. The socket frees the stream automatically when the connection is closed.

Note *SendStreamThenDrop* will close down a server connection to an individual client, not a listening connection.

Blocking connections

When the connection is blocking your socket must initiate reading or writing over the connection rather than waiting passively for a notification from the socket connection. Use a blocking socket when your end of the connection is in charge of when reading and writing takes place.

For client sockets, set the *BlockMode* property to *bmBlocking* to form a blocking connection. Depending on what else your client application does, you may want to create a new execution thread for reading or writing, so that your application can continue executing code on other threads while it waits for the reading or writing over the connection to be completed.

For server sockets, set the *BlockMode* property to *bmBlocking* or *bmThreadBlocking* to form a blocking connection. Because blocking connections hold up the execution of all other code while the socket waits for information to be written or read over the connection, server socket components always spawn a new execution thread for every client connection when the *BlockMode* is *bmThreadBlocking*. When the *BlockMode* is *bmBlocking*, program execution is blocked until a new connection is established.

Part

IV

Creating custom components

The chapters in "Creating custom components" present concepts necessary for designing and implementing custom components in Kylix.

Chapter

24

Overview of component creation

This chapter provides an overview of component design and the process of writing components for Kylix applications. The material here assumes that you are familiar with Kylix and its standard components.

For information on installing new components, see “Installing component packages” on page 11-5.

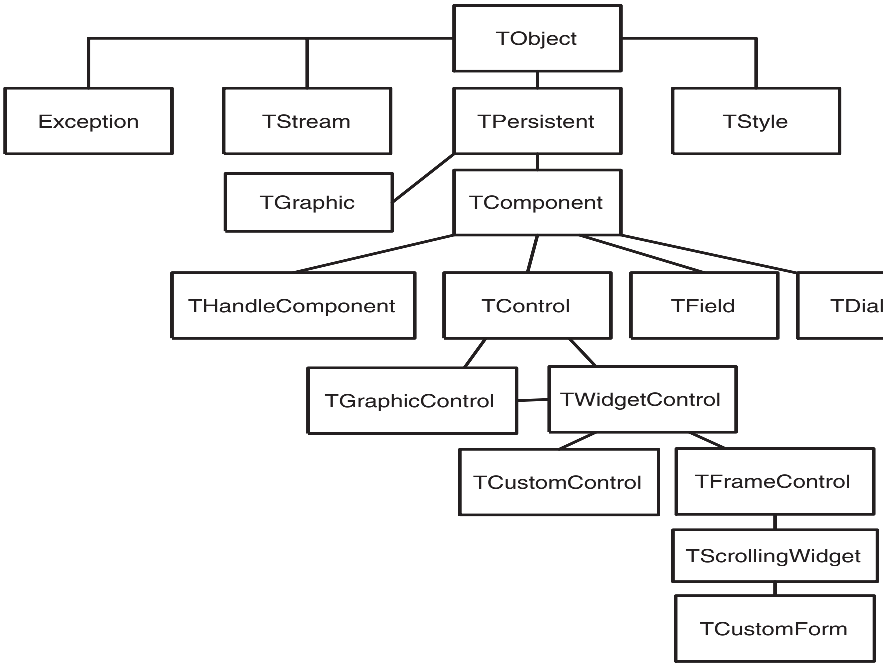
Component Library for Cross Platform (CLX)

Kylix’s components are all part of a class hierarchy called the Component Library for Cross Platform (CLX). Refer to Chapter 3 for more information about CLX basics.

Figure 24.1 shows the relationship of selected classes that make up CLX. For a more detailed discussion of class hierarchies and the inheritance relationships among classes, see Chapter 25, “Object-oriented programming for component writers.”

The *TComponent* class is the shared ancestor of every component in CLX. *TComponent* provides the minimal properties and events necessary for a component to work in Kylix. The various branches of the library provide other, more specialized capabilities.

Figure 24.1 CLX class hierarchy



When you create a component, you add to CLX by deriving a new class from one of the existing class types in the hierarchy.

Components and classes

Because components are classes, component writers work with objects at a different level from application developers. Creating new components requires that you derive new classes.

Briefly, there are two main differences between creating components and using them in applications. When creating components,

- You access parts of the class that are inaccessible to application programmers.
- You add new parts (such as properties) to your components.

Because of these differences, you need to be aware of more conventions and think about how application developers will use the components you write.

How to create components?

A component can be almost any program element that you want to manipulate at design time. Creating a component means deriving a new class from an existing one.

You can derive a new component from any existing component, but the following are the most common ways to create components:

- Modifying existing controls
- Creating controls
- Creating graphic controls
- Subclassing controls
- Creating nonvisual components

Table 24.1 summarizes the different kinds of components and the classes you use as starting points for each.

Table 24.1 Component creation starting points

To do this	Start with this type
Modify an existing component	Any existing component, such as *TButton* or *TPanel*, or an abstract component type, such as *TCustomListBox*
Create a widget-based control	*TWidgetControl*
Create a graphic control	*TGraphicControl*
Subclassing a control	Any widget-based control
Create a nonvisual component	*TComponent*

You can also derive classes that are not components and cannot be manipulated on a form. Kylix includes many such classes, for example, *TFont*.

Modifying existing controls

The simplest way to create a component is to customize an existing one. You can derive a new component from any of the components provided with Kylix. Some controls, such as list boxes and grids, come in several variations on a basic theme. In these cases, CLX includes an abstract class (with the word "custom" in its name, such as *TCustomGrid*) from which to derive customized versions.

For example, you might want to create a special list box that does not have some of the properties of the standard *TListBox* class. You cannot remove (hide) a property inherited from an ancestor class, so you need to derive your component from something above *TListBox* in the hierarchy. Rather than force you to start from the abstract *TWidgetControl* class and reinvent all the list box functions, CLX provides *TCustomListBox*, which implements the properties of a list box but does not publish all of them. When you derive a component from an abstract class like *TCustomListBox*, you only publish the properties you want to make available in your component and leave the rest protected.

Chapter 26, "Creating properties," explains publishing inherited properties. Chapter 31, "Modifying an existing component," and Chapter 33, "Customizing a grid," show examples of modifying existing controls.

Creating controls

Controls are objects that appear at runtime and that the user can interact with. Each widget-based control has a handle, accessed through its *Handle* property, that identifies the underlying widget.

The *TWidgetControl* class is the base class for all of the user interface widgets. All widget-based controls descend from *TWidgetControl*. These include most standard controls, such as pushbuttons, list boxes, and edit boxes. While you could derive an original control (one that's not related to any existing control) directly from *TWidgetControl*, Kylix provides the *TCustomControl* component for this purpose. *TCustomControl* is a specialized control that makes it easier to draw complex visual images.

You can either create a custom control based on *TCustomControl* or use a widget (such as a widget that is not already encapsulated by CLX) and encapsulate it yourself by descending from a related object (or *TWidgetControl* itself).

Chapter 33, "Customizing a grid," presents an example of creating a control.

Creating graphic controls

If your control does not need to receive input focus, you can make it a graphic control. Components like *TProgressBar*, which never receive input focus, are graphic controls. Although these controls cannot receive focus, you can design them to react to system events.

Kylix supports the creation of custom controls through the *TGraphicControl* component. *TGraphicControl* is an abstract class derived from *TControl*. Although you can derive controls directly from *TControl*, it is better to start from *TGraphicControl*, which provides a canvas to paint on.

Chapter 32, "Creating a graphic component," presents an example of creating a graphic control.

Subclassing controls

You create custom controls by defining a new widget. The widget contains information shared among instances of the same sort of control; you can base a new widget on an existing class, which is called *subclassing*. You then put your control in a shared object file, much like the standard controls, and provide an interface to it.

Using Kylix, you can create a component "wrapper" around any existing widget class. So if you already have a library of custom controls that you want to use in Kylix applications, you can create Kylix components that behave like your controls, and derive new controls from them just as you would with any other component.

For examples of the techniques used in subclassing controls, see the components in the QStdCtrls unit that represent standard controls, such as *TEdit*.

Creating nonvisual components

Nonvisual components are used as interfaces for elements like databases (*TSQLConnection*) and system clocks (*TTimer*), and as placeholders for dialog boxes (*TDialog)* and its descendants. Most of the components you write are likely to be visual controls. Nonvisual components can be derived directly from *TComponent*, the abstract base class for all components. *THandleComponent* is the base class for components that require a handle to the underlying widget, such as menus.

What goes into a component?

To make your components reliable parts of the Kylix environment, you need to follow certain conventions in their design. This section discusses the following topics:

- Removing dependencies
- Properties, methods, and events
- Graphics encapsulation
- Registration

Removing dependencies

One quality that makes components usable is the absence of restrictions on what they can do at any point in their code. By their nature, components are incorporated into applications in varying combinations, orders, and contexts. You should design components that function in any situation, without preconditions.

An example of removing dependencies is the *Handle* property of widget controls. If you have written GUI applications before, you know that one of the most difficult and error-prone aspects of getting a program running is making sure that you do not try to access a window or control until you have created it. *Handle* is a unique identifier for the instance of the underlying widget. You can use *Handle* when making low-level function calls, for example, into the Qt shared libraries, where the function call requires a unique identifier for the QWidget object. If the widget does not exist, reading *Handle* causes the component to create the underlying widget. Thus, whenever an application's code accesses the *Handle* property, it is assured of getting a valid handle.

By removing background tasks like creating a widget, Kylix components allow developers to focus on what they really want to do. Before passing a handle to a widget, you do not need to verify that the handle exists or to create the window. The application developer can assume that things will work, instead of constantly checking for things that might go wrong.

Although it can take time to create components that are free of dependencies, it is generally time well spent. It not only spares application developers from repetition and drudgery, but it reduces your documentation and support burdens.

Properties, methods, and events

Aside from the visible image manipulated in the Form designer, the most obvious attributes of a component are its properties, events, and methods. Each of these has a chapter devoted to it in this book, but the discussion that follows explains some of the motivation for their use.

Properties

Properties give the application developer the illusion of setting or reading the value of a variable, while allowing the component writer to hide the underlying data structure or to implement special processing when the value is accessed.

There are several advantages to using properties:

- Properties are available at design time. The application developer can set or change initial values of properties without having to write code.
- Properties can check values or formats as the application developer assigns them. Validating input at design time prevents errors.
- The component can construct appropriate values on demand. Perhaps the most common type of error programmers make is to reference a variable that has not been initialized. By representing data with a property, you can ensure that a value is always available on demand.
- Properties allow you to hide data under a simple, consistent interface. You can alter the way information is structured in a property without making the change visible to application developers.

Chapter 26, "Creating properties," explains how to add properties to your components.

Events

An event is a special property that invokes code in response to input or other activity at runtime. Events give the application developer a way to attach specific blocks of code to specific runtime occurrences, such as mouse actions and keystrokes. The code that executes when an event occurs is called an *event handler*.

Events allow application developers to specify responses to different kinds of input without defining new components.

Chapter 27, "Creating events," explains how to implement standard events and how to define new ones.

Methods

Application developers use methods to direct a component to perform a specific action or return a value not contained by any property. There are two types of methods: class methods and component methods. Class methods are procedures and functions that operate on a class rather than on specific instances of the class. For example, every component's constructor (the *Create* method) is a class method.

Component methods are procedures and functions that operate on the component instances themselves.

Because they require execution of code, methods can be called only at runtime. Methods are useful for several reasons:

- Methods encapsulate the functionality of a component in the same object where the data resides.
- Methods can hide complicated procedures under a simple, consistent interface. An application developer can call a component's *AlignControls* method without knowing how the method works or how it differs from the *AlignControls* method in another component.
- Methods allow updating of several properties with a single call.

Chapter 28, "Creating methods," explains how to add methods to your components.

Graphics encapsulation

Kylix simplifies graphics by encapsulating various graphic tools into a canvas. The canvas represents the drawing surface of a window or control and contains other classes, such as a pen, a brush, and a font. A canvas is a painter that takes care of all the bookkeeping for you.

To draw on a form or other component, you access the component's *Canvas* property. *Canvas* is a property and it is also an object called *TCanvas*. *TCanvas* is a wrapper around a Qt painter that is accessible through the *Handle* property. You can use the handle to access low-level Qt graphics library functions.

If you want to customize a pen or brush, you set its color or style. When you finish, Kylix disposes of the resources. Kylix also caches resources to avoid recreating them if your application frequently uses the same kinds of resource.

You can use the canvas built into Kylix components by descending from them. How graphics images work in the component depends on the canvas of the object from which your component descends. Graphics features are detailed in Chapter 29, "Using graphics in components."

Registration

Before you can install your components in the Kylix IDE, you have to register them. Registration tells Kylix where to place the component on the component palette. You can also customize the way Kylix stores your components in the form file. For information on registering a component, see Chapter 30, "Making components available at design time."

Creating a new component

You can create a new component two ways:

- Using the Component wizard
- Creating a component manually

You can use either of these methods to create a minimally functional component ready to install on the component palette. After installing, you can add your new component to a form and test it at both design time and runtime. You can then add more features to the component, update the component palette, and continue testing.

There are several basic steps that you perform whenever you create a new component. These steps are described below; other examples in this document assume that you know how to perform them.

1. Create a unit for the new component.
2. Derive your component from an existing component type.
3. Add properties, methods, and events.
4. Register your component with Kylix.
5. Create a Help file for your component and its properties, methods, and events.
6. Create a package (a special shared object file) so that you can install your component in the Kylix IDE.

When you finish, the complete component includes the following files:

- Package (bpl<*packagename*>.so)
- Compiled package (.dcp file)
- Compiled unit (.dcu and .dpu files)
- Palette bitmap (.dcr file)
- Help file

Creating a help file to instruct component users on how to use the component is optional.

The chapters in the rest of Part IV explain all the aspects of building components and provide several complete examples of writing different kinds of components.

Using the Component wizard

The Component wizard simplifies the initial stages of creating a component. When you use the Component wizard, you need to specify the following:

- The class from which the new component is derived
- The class name for the new component
- The component palette page where you want it to appear
- The name of the unit in which the component is created
- The search path where the unit is found
- The name of the package in which you want to place the component

The Component wizard performs the same tasks you would when creating a component manually:

- Creating a unit
- Deriving the component
- Registering the component

The Component wizard cannot add components to an existing unit. You must add components to existing units manually.

To start the Component wizard, choose one of these two methods:

- Choose Component | New Component.

  or

- Choose File | New and double-click on Component

Fill in the fields in the Component wizard:

1. In the Ancestor Type field, specify the class from which you are deriving your new component.
2. In the Class Name field, specify the name of your new component class.
3. In the Palette Page field, specify the page on the component palette on which you want the new component to be installed.
4. In the Unit file name field, specify the name of the unit you want the component class declared in.
5. If the unit is not on the search path, edit the search path in the Search Path field as necessary.

To place the component in a new or existing package, click Component | Install and use the dialog box that appears to specify a package.

Warning If you derive a component from a CLX class whose name begins with “custom” (such as *TCustomControl*), do not try to place the new component on a form until you have overridden any abstract methods in the original component. Kylix cannot create instance objects of a class that has abstract methods.

Kylix creates a new unit containing the class declaration and the *Register* procedure, and adds a **uses** clause that includes all the standard Kylix units. To see the source code for your unit, click View Unit. (If the Component wizard is already closed, open the unit file in the Code editor by selecting File | Open.) The unit looks like this:

```
unit MyControl;

interface

uses
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;

type
  TMyControl = class(TCustomControl)
  private
  { Private declarations }
  protected
  { Protected declarations }
```

```
    public
    { Public declarations }
    published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TMyControl]);
end;

end.
```

Creating a component manually

The easiest way to create a new component is to use the Component wizard. You can, however, perform the same steps manually.

Creating a component manually involves the following tasks:

- Creating a unit file
- Deriving the component
- Registering the component

Creating a unit file

A unit is a separately compiled module of Object Pascal code. Kylix uses units for several purposes. Every form has its own unit, and most components (or groups of related components) have their own units as well.

When you create a component, you either create a new unit for the component or add the new component to an existing unit.

To create a unit, choose File|New to and double-click on Unit. Kylix creates a new unit file and opens it in the Code editor.

To open an existing unit, choose File|Open and select the source code unit to which you want to add your component.

Note When adding a component to an existing unit, make sure that the unit contains only component code. For example, adding component code to a unit that contains a form causes errors in the component palette.

Once you have either a new or existing unit for your component, you can derive the component class.

Deriving the component

Every component is a class derived from *TComponent*, from one of its more specialized descendants (such as *TControl* or *TGraphicControl*), or from an existing component class. "How to create components?" on page 24-2 provides guidelines concerning which class to derive different kinds of components from. However, you need to review the CLX object hierarchy to determine the best object to use.

To derive a component, add an object type declaration to the **interface** part of the unit that will contain the component. Deriving classes is explained in more detail in the section "Defining new classes" on page 25-1.

A simple component class is a nonvisual component descended directly from *TComponent*.

To create a simple component class, add the following class declaration to the **interface** part of your component unit:

```
type
  TNewComponent = class(TComponent)
  end;
```

So far the new component does nothing different from *TComponent*. You have created a framework on which to build your new component.

Registering the component

Registration is a simple process that tells Kylix which components to add to its component library, and on which pages of the component palette they should appear. For a more detailed discussion of the registration process, see Chapter 30, "Making components available at design time."

To register a component,

1 Add a procedure named *Register* to the **interface** part of the component's unit. *Register* takes no parameters, so the declaration is very simple:

   ```
   procedure Register;
   ```

   If you are adding a component to a unit that already contains components, it should already have a *Register* procedure declared, so you do not need to change the declaration.

2 Write the *Register* procedure in the **implementation** part of the unit, calling *RegisterComponents* for each component you want to register. *RegisterComponents* is a procedure that takes two parameters: the name of a component palette page and a set of component types. If you are adding a component to an existing registration, you can either add the new component to the set in the existing statement, or add a new statement that calls *RegisterComponents*.

To register a component named *TMyControl* and place it on the Samples page of the palette, you would add the following *Register* procedure to the unit that contains *TMyControl*'s declaration:

```
procedure Register;
begin
  RegisterComponents('Samples', [TNewControl]);
end;
```

This *Register* procedure places *TMyControl* on the Samples page of the component palette.

Once you register a component, you can compile it into a package (see Chapter 30) and install it on the component palette.

Testing uninstalled components

You can test the runtime behavior of a component before you install it on the component palette. This is particularly useful for debugging newly created components, but the same technique works with any component, whether or not it is on the component palette. For information on testing already installed components, see "Testing installed components" on page 24-13.

You test an uninstalled component by emulating the actions performed by Kylix when the component is selected from the palette and placed on a form.

To test an uninstalled component,

1 Add the name of component's unit to the form unit's **uses** clause.

2 Add an object field to the form to represent the component.

   This is one of the main differences between the way you add components and the way the IDE does it. You add the object field to the public part at the bottom of the form's type declaration. The IDE would add it above, in the part of the type declaration that it manages.

   Never add fields to the IDE-managed part of the form's type declaration. The items in that part of the type declaration correspond to the items stored in the form file. Adding the names of components that do not exist on the form can render your form file invalid.

3 Attach a handler to the form's *OnCreate* event.

4 Construct the component in the form's *OnCreate* handler.

   When you call the component's constructor, you must pass a parameter specifying the owner of the component (the component responsible for saving the component to a form file and for destroying the component when the time comes). You will nearly always pass *Self* as the owner. In a method, *Self* is a reference to the object that contains the method. In this case, in the form's *OnCreate* handler, *Self* refers to the form.

5 If your component is a control, assign the *Parent* property.

   Setting the *Parent* property is always the first thing to do after constructing a control. The parent is the control that contains your control visually; usually it is the form on which the control appears, but it might be a group box or panel. Normally, you'll set *Parent* to *Self*, that is, the form. Always set *Parent* before setting other properties of the control.

Warning If your component is not a control (that is, if *TControl* is not one of its ancestors), skip this step. If you accidentally set the form's *Parent* property (instead of the component's) to *Self*, you can cause an operating-system problem.

6 Set any other component properties as desired.

For example, to test a new component of type *TMyControl* in a unit named *MyControl*: create a new project, then follow the steps to create a form unit that looks like this:

```
unit Unit1;
interface

uses
  SysUtils, Types, Classes, QGraphics, QControls;
  QForms, QDialogs, MyControl;                        { 1. Add MyControl to uses clause }

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);            { 3. Attach a handler to OnCreate }
  private
    { Private declarations }
  public
    { Public Declarations }
    MyControl1: TMyControl1;                                   { 2. Add an object field }
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  MyControl1 := TMyControl.Create(Self);                  { 4. Construct the component }
  MyControl1.Parent := Self;        { 5. Set Parent property if component is a control }
  MyControl1.Left := 12;                                  { 6. Set other properties... )
  ⋮                                                                ...continue as needed }
end;
end.
```

Testing installed components

You can test the design-time behavior of a component after you install it on the component palette. This is particularly useful for debugging newly created components, but the same technique works with any component, whether or not it is on the component palette. For information on testing components that have not yet been installed, see “Testing uninstalled components” on page 24-12.

Testing your components after installing allows you to debug the component that only generates design-time exceptions when dropped on a form.

Test an installed component using a second running instance of Kylix:

1 From the Kylix IDE menu select Project|Options and on the Directories/Conditional page, set the Debug Source Path to the component’s source file.

2 Then select Tools|Debugger Options. On the Language Exceptions page, enable the exceptions you want to track.

3. Open the component source file and set breakpoints.
4. Select Run | Parameters and set the Host Application field to the name and location of the Kylix executable file.
5. In the Run Parameters dialog, click the Load button to start a second instance of Kylix.
6. Then drop the components to be tested on the form, which should break on your breakpoints in the source.

Chapter

25

Object-oriented programming for component writers

If you have written applications with Kylix, you know that a class contains both data and code, and that you can manipulate classes at design time and at runtime. In that sense, you've become a component user.

When you create new components, you deal with classes in ways that application developers never need to. You also try to hide the inner workings of the component from the developers who will use it. By choosing appropriate ancestors for your components, designing interfaces that expose only the properties and methods that developers need, and following the other guidelines in this chapter, you can create versatile, reusable components.

Before you start creating components, you should be familiar with these topics, which are related to object-oriented programming (OOP):

- Defining new classes
- Ancestors, descendants, and class hierarchies
- Controlling access
- Dispatching methods
- Abstract class members
- Classes and pointers

Defining new classes

The difference between component writers and application developers is that component writers create new classes while application developers manipulate instances of classes.

A class is essentially a type. As a programmer, you are always working with types and instances, even if you do not use that terminology. For example, you create variables of a type, such as *Integer*. Classes are usually more complex than simple data types, but they work the same way: By assigning different values to instances of the same type, you can perform different tasks.

For example, it is quite common to create a form containing two buttons, one labeled OK and one labeled Cancel. Each is an instance of the class *TButton*, but by assigning different values to their *Caption* properties and different handlers to their *OnClick* events, you make the two instances behave differently.

Deriving new classes

There are two reasons to derive a new class:

- To change class defaults to avoid repetition
- To add new capabilities to a class

In either case, the goal is to create reusable objects. If you design components with reuse in mind, you can save work later on. Give your classes usable default values, but allow them to be customized.

To change class defaults to avoid repetition

Most programmers try to avoid repetition. Thus, if you find yourself rewriting the same lines of code over and over, you place the code in a subroutine or function, or build a library of routines that you can use in many programs. The same reasoning holds for components. If you find yourself changing the same properties or making the same method calls, you can create a new component that does these things by default.

For example, suppose that each time you create an application, you add a dialog box to perform a particular operation. Although it is not difficult to recreate the dialog each time, it is also not necessary. You can design the dialog once, set its properties, and install a wrapper component associated with it onto the Component palette. By making the dialog into a reusable component, you not only eliminate a repetitive task, but you encourage standardization and reduce the likelihood of errors each time the dialog is recreated.

Chapter 31, "Modifying an existing component," shows an example of changing a component's default properties.

Note If you want to modify only the published properties of an existing component, or to save specific event handlers for a component or group of components, you may be able to accomplish this more easily by creating a *component template*.

To add new capabilities to a class

A common reason for creating new components is to add capabilities not found in existing components. When you do this, you derive the new component from either an existing component or an abstract base class, such as *TComponent* or *TControl*.

Derive your new component from the class that contains the closest subset of the features you want. You can add capabilities to a class, but you cannot take them away; so if an existing component class contains properties that you do *not* want to include in yours, you should derive from that component's ancestor.

For example, if you want to add features to a list box, you could derive your component from *TListBox*. However, if you want to add new features but exclude some capabilities of the standard list box, you need to derive your component from *TCustomListBox*, the ancestor of *TListBox*. Then you can recreate (or make visible) only the list-box capabilities you want, and add your new features.

Chapter 33, "Customizing a grid," shows an example of customizing an abstract component class.

Declaring a new component class

In addition to standard components, Kylix provides many abstract classes designed as bases for deriving new components. Table 24.1 on page 24-3 shows the classes you can start from when you create your own components.

To declare a new component class, add a class declaration to the component's unit file.

Here is the declaration of a simple graphical component:

```
type
  TSampleShape = class(TGraphicControl)
  end;
```

A finished component declaration usually includes property, event, and method declarations before the **end**. But a declaration like the one above is also valid, and provides a starting point for the addition of component features.

Ancestors, descendants, and class hierarchies

Application developers take for granted that every control has properties named *Top* and *Left* that determine its position on the form. To them, it may not matter that all controls inherit these properties from a common ancestor, *TControl*. When you create a component, however, you must know which class to derive it from so that it inherits the appropriate features. And you must know everything that your control inherits, so you can take advantage of inherited features without recreating them.

The class from which you derive a component is called its *immediate ancestor*. Each component inherits from its immediate ancestor, and from the immediate ancestor of its immediate ancestor, and so forth. All of the classes from which a component inherits are called its *ancestors*; the component is a *descendant* of its ancestors.

Together, all the ancestor-descendant relationships in an application constitute a hierarchy of classes. Each generation in the hierarchy contains more than its ancestors, since a class inherits everything from its ancestors, then adds new properties and methods or redefines existing ones.

If you do not specify an immediate ancestor, Kylix derives your class from the default ancestor, *TObject*. *TObject* is the ultimate ancestor of all classes in the object hierarchy.

The general rule for choosing which object to derive from is simple: Pick the object that contains as much as possible of what you want to include in your new object, but which does not include anything you do not want in the new object. You can always add things to your objects, but you cannot take things out.

Controlling access

There are several levels of *access control*—also called *visibility*—on properties, methods, and fields. Visibility determines which code can access which parts of the class. By specifying visibility, you define the *interface* to your components.

Table 25.1 shows the levels of visibility, from most restrictive to most accessible:

Table 25.1 Levels of visibility within an object

Visibility	Meaning	Used for
private	Accessible only to code in the unit where the class is defined.	Hiding implementation details.
protected	Accessible to code in the unit(s) where the class and its descendants are defined.	Defining the component writer's interface.
public	Accessible to all code.	Defining the runtime interface.
published	Accessible to all code and from the Object Inspector.	Defining the design-time interface.

Declare members as **private** if you want them to be available only within the class where they are defined; declare them as **protected** if you want them to be available only within that class and its descendants. Remember, though, that if a member is available anywhere within a unit file, it is available *everywhere* in that file. Thus, if you define two classes in the same unit, the classes will be able to access each other's private methods. And if you derive a class in a different unit from its ancestor, all the classes in the new unit will be able to access the ancestor's protected methods.

Hiding implementation details

Declaring part of a class as **private** makes that part invisible to code outside the class's unit file. Within the unit that contains the declaration, code can access the part as if it were public.

This example shows how declaring a field as **private** hides it from application developers. The listing shows two form units. Each form has a handler for its *OnCreate* event which assigns a value to a private field. The compiler allows assignment to the field only in the form where it is declared.

```
unit HideInfo;
interface

uses SysUtils, Classes, QGraphics, QControls, QForms, QDialogs;

type
```

```
  TSecretForm = class(TForm)                                   { declare new form }
    procedure FormCreate(Sender: TObject);
  private                                                   { declare private part }
    FSecretCode: Integer;                                 { declare a private field }
  end;

var
  SecretForm: TSecretForm;

implementation
procedure TSecretForm.FormCreate(Sender: TObject);
begin
  FSecretCode := 42;                                      { this compiles correctly }
end;
end.                                                                  { end of unit }

unit TestHide;                                          { this is the main form file }

interface
uses SysUtils, Classes, QGraphics, QControls, QForms, QDialogs,
  HideInfo;                                        { use the unit with TSecretForm }

type
  TTestForm = class(TForm)
    procedure FormCreate(Sender: TObject);
  end;
var
  TestForm: TTestForm;

implementation
procedure TTestForm.FormCreate(Sender: TObject);
begin
  SecretForm.FSecretCode := 13;        { compiler stops with "Field identifier expected" }
end;
end.                                                                  { end of unit }
```

Although a program using the *HideInfo* unit can use objects of type *TSecretForm*, it cannot access the *FSecretCode* field in any of those objects.

Defining the component writer's interface

Declaring part of a class as **protected** makes that part visible only to the class itself and its descendants (and to other classes that share their unit files).

You can use **protected** declarations to define a *component writer's interface* to the class. Application units do not have access to the protected parts, but derived classes do. This means that component writers can change the way a class works without making the details visible to application developers.

Note A common mistake is trying to access protected methods from an event handler. Event handlers are typically methods of the form, not the component that receives the event. As a result, they do not have access to the component's protected methods (unless the component is declared in the same unit as the form).

Defining the runtime interface

Declaring part of a class as **public** makes that part visible to any code that has access to the class as a whole.

Public parts are available at runtime to all code, so the public parts of a class define its *runtime interface*. The runtime interface is useful for items that are not meaningful or appropriate at design time, such as properties that depend on runtime input or which are read-only. Methods that you intend for application developers to call must also be public.

Here is an example that shows two read-only properties declared as part of a component's runtime interface:

```
type
  TSampleComponent = class(TComponent)
  private
    FTempCelsius: Integer;                        { implementation details are private }
    function GetTempFahrenheit: Integer;
  public
    property TempCelsius: Integer read FTempCelsius;             { properties are public }
    property TempFahrenheit: Integer read GetTempFahrenheit;
  end;
  ⋮
function TSampleComponent.GetTempFahrenheit: Integer;
begin
  Result := FTempCelsius * 9 div 5 + 32;
end;
```

Defining the design-time interface

Declaring part of a class as **published** makes that part public and also generates runtime type information. Among other things, runtime type information allows the Object Inspector to access properties and events.

Because they show up in the Object Inspector, the published parts of a class define that class's *design-time interface*. The design-time interface should include any aspects of the class that an application developer might want to customize at design time, but must exclude any properties that depend on specific information about the runtime environment.

Here is an example of a published property called *Temperature*. Because it is published, it appears in the Object Inspector at design time.

```
type
  TSampleComponent = class(TComponent)
  private
    FTemperature: Integer;                        { implementation details are private }
  published
    property Temperature: Integer read FTemperature write FTemperature;      { writable! }
  end;
```

Dispatching methods

Dispatch refers to the way a program determines where a method should be invoked when it encounters a method call. The code that calls a method looks like any other procedure or function call. But classes have different ways of dispatching methods.

The three types of method dispatch are

- Static
- Virtual
- Dynamic

Static methods

All methods are static unless you specify otherwise when you declare them. Static methods work like regular procedures or functions. The compiler determines the exact address of the method and links the method at compile time.

The primary advantage of static methods is that dispatching them is very quick. Because the compiler can determine the exact address of the method, it links the method directly. Virtual and dynamic methods, by contrast, use indirect means to look up the address of their methods at runtime, which takes somewhat longer.

A static method does not change when inherited by a descendant class. If you declare a class that includes a static method, then derive a new class from it, the derived class shares exactly the same method at the same address. This means that you cannot override static methods; a static method always does exactly the same thing no matter what class it is called in. If you declare a method in a derived class with the same name as a static method in the ancestor class, the new method simply replaces the inherited one in the derived class.

An example of static methods

In the following code, the first component declares two static methods. The second declares two static methods with the same names that replace the methods inherited from the first component.

```
type
  TFirstComponent = class(TComponent)
    procedure Move;
    procedure Flash;
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move;       { different from the inherited method, despite same declaration }
    function Flash(HowOften: Integer): Integer;                 { this is also different }
  end;
```

Virtual methods

Virtual methods employ a more complicated, and more flexible, dispatch mechanism than static methods. A virtual method can be redefined in descendant classes, but still be called in the ancestor class. The address of a virtual method isn't determined at compile time; instead, the object where the method is defined looks up the address at runtime.

To make a method virtual, add the directive **virtual** after the method declaration. The **virtual** directive creates an entry in the object's *virtual method table*, or VMT, which holds the addresses of all the virtual methods in an object type.

When you derive a new class from an existing one, the new class gets its own VMT, which includes all the entries from the ancestor's VMT plus any additional virtual methods declared in the new class.

Overriding methods

Overriding a method means extending or refining it, rather than replacing it. A descendant class can override any of its inherited virtual methods.

To override a method in a descendant class, add the directive **override** to the end of the method declaration.

Overriding a method causes a compilation error if

- The method does not exist in the ancestor class.
- The ancestor's method of that name is static.
- The declarations are not otherwise identical (number and type of arguments parameters differ).

The following code shows the declaration of two simple components. The first declares three methods, each with a different kind of dispatching. The other, derived from the first, replaces the static method and overrides the virtual methods.

```
type
  TFirstComponent = class(TCustomControl)
    procedure Move;              { static method }
    procedure Flash; virtual;    { virtual method }
    procedure Beep; dynamic;     { dynamic virtual method }
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move;              { declares new method }
    procedure Flash; override;   { overrides inherited method }
    procedure Beep; override;    { overrides inherited method }
  end;
```

Dynamic methods

Dynamic methods are virtual methods with a slightly different dispatch mechanism. Because dynamic methods don't have entries in the object's virtual method table, they can reduce the amount of memory that objects consume. However, dispatching dynamic methods is somewhat slower than dispatching regular virtual methods. If a method is called frequently, or if its execution is time-critical, you should probably declare it as virtual rather than dynamic.

Objects must store the addresses of their dynamic methods. But instead of receiving entries in the virtual method table, dynamic methods are listed separately. The dynamic method list contains entries only for methods introduced or overridden by a particular class. (The virtual method table, in contrast, includes all of the object's virtual methods, both inherited and introduced.) Inherited dynamic methods are dispatched by searching each ancestor's dynamic method list, working backwards through the inheritance tree.

To make a method dynamic, add the directive **dynamic** after the method declaration.

Abstract class members

When a method is declared as **abstract** in an ancestor class, you must surface it (by redeclaring and implementing it) in any descendant component before you can use the new component in applications. Kylix cannot create instances of a class that contains abstract members. For more information about surfacing inherited parts of classes, see Chapter 26, "Creating properties," and Chapter 28, "Creating methods."

Classes and pointers

Every class (and therefore every component) is really a pointer. The compiler automatically dereferences class pointers for you, so most of the time you do not need to think about this. The status of classes as pointers becomes important when you pass a class as a parameter. In general, you should pass classes by value rather than by reference. The reason is that classes are already pointers, which are references; passing a class by reference amounts to passing a reference to a reference.

Chapter

26

Creating properties

Properties are the most visible parts of components. The application developer can see and manipulate them at design time and get immediate feedback as the components react in the Form designer. Well-designed properties make your components easier for others to use and easier for you to maintain.

To make the best use of properties in your components, you should understand the following:

- Why create properties?
- Types of properties
- Publishing inherited properties
- Defining properties
- Creating array properties
- Storing and loading properties

Why create properties?

From the application developer's standpoint, properties look like variables. Developers can set or read the values of properties as if they were fields. (About the only thing you can do with a variable that you cannot do with a property is pass it as a **var** parameter.)

Properties provide more power than simple fields because

- Application developers can set properties at design time. Unlike fields and methods, which are available only at runtime, properties let the developer customize components before running an application. Properties can appear in the Object Inspector, which simplifies the programmer's job; instead of handling several parameters to construct an object, you let Kylix read the values from the Object Inspector. The Object Inspector also validates property assignments as soon as they are made.

- Properties can hide implementation details. For example, data stored internally in an encrypted form can appear unencrypted as the value of a property; although the value is a simple number, the component may look up the value in a database or perform complex calculations to arrive at it. Properties let you attach complex effects to outwardly simple assignments; what looks like an assignment to a field can be a call to a method which implements elaborate processing.
- Properties can be virtual. Hence, what looks like a single property to an application developer may be implemented differently in different components.

A simple example is the *Top* property of all controls. Assigning a new value to *Top* does not just change a stored value; it repositions and repaints the control. And the effects of setting a property need not be limited to an individual component; for example, setting the *Down* property of a speed button to *True* sets *Down* property of all other speed buttons in its group to *False*.

Types of properties

A property can be of any type. Different types are displayed differently in the Object Inspector, which validates property assignments as they are made at design time.

Table 26.1 How properties appear in the Object Inspector

Property type	Object Inspector treatment
Simple	Numeric, character, and string properties appear as numbers, characters, and strings. The application developer can edit the value of the property directly.
Enumerated	Properties of enumerated types (including Boolean) appear as editable strings. The developer can also cycle through the possible values by double-clicking the value column, and there is a drop-down list that shows all possible values.
Set	Properties of set types appear as sets. By double-clicking on the property, the developer can expand the set and treat each element as a Boolean value (**true** if it is included in the set).
Object	Properties that are themselves classes often have their own property editors, specified in the component's registration procedure. If the class held by a property has its own published properties, the Object Inspector lets the developer to expand the list (by double-clicking) to include these properties and edit them individually. Object properties must descend from *TPersistent*.
Interface	Properties that are interfaces can appear in the Object Inspector as long as the value is an interface that is implemented by a component (a descendant of *TComponent*). Interface properties often have their own property editors.
Array	Array properties must have their own property editors; the Object Inspector has no built-in support for editing them. You can specify a property editor when you register your components.

Publishing inherited properties

All components inherit properties from their ancestor classes. When you derive a new component from an existing one, your new component inherits all the properties of its immediate ancestor. If you derive from one of the abstract classes, many of the inherited properties are either protected or public, but not published.

To make a protected or public property available at design time in the Object Inspector, you must redeclare the property as published. Redeclaring means adding a declaration for the inherited property to the declaration of the descendant class.

If you derive a component from *TWidgetControl*, it inherits the protected *Bitmap* property. By redeclaring *Bitmap* in your new component, you can change the level of protection to either public or published.

The following code shows a redeclaration of *Bitmap* as published, making it available at design time.

```
type
  TSampleComponent = class(TWidgetControl)
  published
    property Bitmap;
  end;
```

When you redeclare a property, you specify only the property name, not the type and other information described below in “Defining properties”. You can also declare new default values and specify whether to store the property.

Redeclarations can make a property less restricted, but not more restricted. Thus you can make a protected property public, but you cannot hide a public property by redeclaring it as protected.

Defining properties

This section shows how to declare new properties and explains some of the conventions followed in the standard components. Topics include

- The property declaration
- Internal data storage
- Direct access
- Access methods
- Default property values

The property declaration

A property is declared in the declaration of its component class. To declare a property, you specify three things:

- The name of the property.
- The type of the property.

- The methods used to read and write the value of the property. If no write method is declared, the property is read-only.

Properties declared in a **published** section of the component's class declaration are editable in the Object Inspector at design time. The value of a published property is saved with the component in the form file. Properties declared in a **public** section are available at runtime and can be read or set in program code.

Here is a typical declaration for a property called *Count*.

```
type
  TYourComponent = class(TComponent)
  private
    FCount: Integer;                        { used for internal storage }
    procedure SetCount (Value: Integer);    { write method }
  public
    property Count: Integer read FCount write SetCount;
  end;
```

Internal data storage

There are no restrictions on how you store the data for a property. In general, however, components follow these conventions:

- Property data is stored in class fields.
- The fields used to store property data are private and should be accessed only from within the component itself. Derived components should use the inherited property; they do not need direct access to the property's internal data storage.
- Identifiers for these fields consist of the letter *F* followed by the name of the property. For example, the raw data for the *Width* property defined in *TControl* is stored in a field called *FWidth*.

The principle that underlies these conventions is that only the implementation methods for a property should access the data behind it. If a method or another property needs to change that data, it should do so through the property, not by direct access to the stored data. This ensures that the implementation of an inherited property can change without invalidating derived components.

Direct access

The simplest way to make property data available is *direct access*. That is, the **read** and **write** parts of the property declaration specify that assigning or reading the property value goes directly to the internal-storage field without calling an access method. Direct access is useful when you want to make a property available in the Object Inspector but changes to its value trigger no immediate processing.

It is common to have direct access for the **read** part of a property declaration but use an access method for the **write** part. This allows the status of the component to be updated when the property value changes.

The following component-type declaration shows a property that uses direct access for both the **read** and the **write** parts.

```
type
  TSampleComponent = class(TComponent)
  private                                               { internal storage is private}
    FMyProperty: Boolean;                         { declare field to hold property value }
  published                                     { make property available at design time }
    property MyProperty: Boolean read FMyProperty write FMyProperty;
  end;
```

Access methods

You can specify an access method instead of a field in the **read** and **write** parts of a property declaration. Access methods should be protected, and are usually declared as **virtual**; this allows descendant components to override the property's implementation.

Avoid making access methods public. Keeping them protected ensures that application developers do not inadvertently modify a property by calling one of these methods.

Here is a class that declares three properties using the index specifier, which allows all three properties to have the same read and write access methods:

```
type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  private
    function GetDateElement(Index: Integer): Integer; { note the Index parameter }
    procedure SetDateElement(Index: Integer; Value: Integer);
  ⋮
```

Because each element of the date (day, month, and year) is an integer, and because setting each requires encoding the date when set, the code avoids duplication by sharing the read and write methods for all three properties. You need only one method to read a date element, and another to write the date element.

Here is the read method that obtains the date element:

```
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);            { break encoded date into elements }
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;
```

This is the write method that sets the appropriate date element:

```
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
  AYear, AMonth, ADay: Word;
begin
  if Value > 0 then                              { all elements must be positive }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);      { get current date elements }
    case Index of                                { set new element depending on Index }
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);    { encode the modified date }
    Refresh;                                     { update the visible calendar }
  end;
end;
```

The read method

The read method for a property is a function that takes no parameters (except as noted below) and returns a value of the same type as the property. By convention, the function's name is *Get* followed by the name of the property. For example, the read method for a property called *Count* would be *GetCount*. The read method manipulates the internal storage data as needed to produce the value of the property in the appropriate type.

The only exceptions to the no-parameters rule are for array properties and properties that use index specifiers (see "Creating array properties" on page 26-8), both of which pass their index values as parameters. (Use index specifiers to create a single read method that is shared by several properties. For more information about index specifiers, see the *Object Pascal Language Guide*.)

If you do not declare a read method, the property is write-only. Write-only properties are seldom used.

The write method

The write method for a property is a procedure that takes a single parameter (except as noted below) of the same type as the property. The parameter can be passed by reference or by value, and can have any name you choose. By convention, the write method's name is *Set* followed by the name of the property. For example, the write method for a property called *Count* would be *SetCount*. The value passed in the parameter becomes the new value of the property; the write method must perform any manipulation needed to put the appropriate data in the property's internal storage.

The only exceptions to the single-parameter rule are for array properties and properties that use index specifiers, both of which pass their index values as a second parameter. (Use index specifiers to create a single write method that is shared by several properties. For more information about index specifiers, see the *Object Pascal Language Guide*.)

If you do not declare a write method, the property is read-only.

Write methods commonly test whether a new value differs from the current value before changing the property. For example, here is a simple write method for an integer property called *Count* that stores its current value in a field called *FCount*.

```
procedure TMyComponent.SetCount(Value: Integer);
begin
  if Value <> FCount then
  begin
    FCount := Value;
    Update;
  end;
end;
```

Default property values

When you declare a property, you can specify a *default value* for it. Kylix uses the default value to determine whether to store the property in a form file. If you do not specify a default value for a property, Kylix always stores the property.

To specify a default value for a property, append the **default** directive to the property's declaration (or redeclaration), followed by the default value. For example,

```
property Cool Boolean read GetCool write SetCool default True;
```

Note Declaring a default value does not set the property to that value. The component's constructor should initialize property values when appropriate. However, since objects always initialize their fields to 0, it is not strictly necessary for the constructor to set integer properties to 0, string properties to null, or Boolean properties to *False*.

Specifying no default value

When redeclaring a property, you can specify that the property has no default value, even if the inherited property specified one.

To designate a property as having no default value, append the **nodefault** directive to the property's declaration. For example,

```
property FavoriteFlavor string nodefault;
```

When you declare a property for the first time, there is no need to include **nodefault**. The absence of a declared default value means that there is no default.

Here is the declaration of a component that includes a single Boolean property called *IsTrue* with a default value of *True*. Below the declaration (in the **implementation** section of the unit) is the constructor that initializes the property.

```
type
  TSampleComponent = class(TComponent)
  private
    FIsTrue: Boolean;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property IsTrue: Boolean read FIsTrue write FIsTrue default True;
  end;
⋮
constructor TSampleComponent.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);               { call the inherited constructor }
  FIsTrue := True;                        { set the default value }
end;
```

Creating array properties

Some properties lend themselves to being indexed like arrays. For example, the *Items* property of *TCustomTreeView* is an indexed list of the nodes in the tree; you can treat it as an array of tree nodes. *Items* provides natural access to a particular element (a node) in a larger set of data (the set of all nodes in the tree).

Array properties are declared like other properties, except that

- The declaration includes one or more indexes with specified types. The indexes can be of any type.
- The **read** and **write** parts of the property declaration, if specified, must be methods. They cannot be fields.

The read and write methods for an array property take additional parameters that correspond to the indexes. The parameters must be in the same order and of the same type as the indexes specified in the declaration.

There are a few important differences between array properties and arrays. Unlike the index of an array, the index of an array property does not have to be an integer type. You can index a property on a string, for example. In addition, you can reference only individual elements of an array property, not the entire range of the property.

The following example shows the declaration of a property that returns a string based on an integer index.

```
type
  TDemoComponent = class(TComponent)
  private
    function GetNumberName(Index: Integer): string;
  public
    property NumberName[Index: Integer]: string read GetNumberName;
  end;
⋮
function TDemoComponent.GetNumberName(Index: Integer): string;
begin
  Result := 'Unknown';
  case Index of
    -MaxInt..-1: Result := 'Negative';
    0: Result := 'Zero';
    1..100: Result := 'Small';
    101..MaxInt: Result := 'Large';
  end;
end;
```

Creating properties for subcomponents

By default, when a property's value is another component, you assign a value to that property by adding an instance of the other component to the form or data module and then assigning that component as the value of the property. However, it is also possible for your component to create its own instance of the object that implements the property value. Such a dedicated component is called a subcomponent.

Subcomponents can be any persistent object (any descendant of *TPersistent*). Unlike separate components that happen to be assigned as the value of a property, the published properties of subcomponents are saved with the component that creates them. In order for this to work, however, the following conditions must be met:

- The *Owner* of the subcomponent must be the component that creates it and uses it as the value of a published property. For subcomponents that are descendants of *TComponent*, you can accomplish this by setting the *Owner* property of the subcomponent. For other subcomponents, you must override the *GetOwner* method of the persistent object so that it returns the creating component.
- If the subcomponent is a descendant of *TComponent*, it must indicate that it is a subcomponent by calling the *SetSubComponent* method. Typically, this call is made either by the owner when it creates the subcomponent or by the constructor of the subcomponent.

Typically, properties whose values are subcomponents are read-only. If you allow a property whose value is a subcomponent to be changed, the property setter must free the subcomponent when another component is assigned as the property value. In addition, the component often re-instantiates its subcomponent when the property is set to nil. Otherwise, once the property is changed to another component, the subcomponent can never be restored at design time. The following example illustrates such a property setter for a property whose value is a *TTimer*:

```
procedure TDemoComponent.SetTimerProp(Value: TTimer);
begin
  if Value <> FTimer then
  begin
    if Value <> nil then
    begin
      if (FTimer <> nil and FTimer.Owner = self then
        FTimer.Free;
      FTimer := Value;
      FTimer,FreeNotification(self);
    end
    else { nil value }
    begin
      if FTimer.Owner <> self then
      {
        FTimer := TTimer.Create(self);
        FTimer.SetSubComponent(True);
        FTimer.FreeNotification(self);
      }
    end;
  end;
end;
```

Note that the property setter above called the *FreeNotification* method of the component that is set as the property value. This call ensures that the component that is the value of the property sends a notification if it is about to be destroyed. It sends this notification by calling the *Notification* method. You handle this call by overriding the *Notification* method, as follows:

```
procedure TDemoComponent.Notification(AComponent: TComponent; Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and (AComponent = FTimer) then
    FTimer := nil;
end;
```

Creating properties for interfaces

You can use an interface as the value of a published property, much as you can use an object. However, the mechanism by which your component receives notifications from the implementation of that interface differs. In the previous topic, the property setter called the *FreeNotification* method of the component that was assigned as the property value. This allowed the component to update itself when the component that was the value of the property was freed. When the value of the property is an interface, however, you don’t have access to the component that implements that interface. As a result, you can’t call its *FreeNotification* method.

To handle this situation, you can call your component’s *ReferenceInterface* method:

```
procedure TDemoComponent.SetMyIntfProp(const Value: IMyInterface);
begin
  ReferenceInterface(FIntfField, opRemove);
  FIntfField := Value;
  ReferenceInterface(FIntfField, opInsert);
end;
```

Calling *ReferenceInterface* with a specified interface does the same thing as calling another component’s *FreeNotification* method. Thus, after calling *ReferenceInterface* from the property setter, you can override the *Notification* method to handle the notifications from the implementor of the interface:

```
procedure TDemoComponent.Notification(AComponent: TComponent; Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Assigned(MyIntfProp)) and (AComponent.IsImplementorOf(MyInftProp)) then
    MyIntfProp := nil;
end;
```

Note that the *Notification* code assigns **nil** to the *MyIntfProp* property, not to the private field (*FIntfField*). This ensures that *Notification* calls the property setter, which calls *ReferenceInterface* to remove the notification request that was established when the property value was set previously. All assignments to the interface property must be made through the property setter.

Storing and loading properties

Kylix stores forms and their components in form (.xfm) files. A form file contains the properties of a form and its components. When Kylix developers add components you write to their forms, your components must have the ability to write their properties to the form file when saved. Similarly, when loaded into Kylix or executed as part of an application, the components must restore themselves from the form file.

Most of the time you will not need to do anything to make your components work with form files because the ability to store a representation and load from it are part of the inherited behavior of components. Sometimes, however, you might want to alter the way a component stores itself or the way it initializes when loaded; so you should understand the underlying mechanism.

These are the aspects of property storage you need to understand:

- Using the store-and-load mechanism
- Specifying default values
- Determining what to store
- Initializing after loading
- Storing and loading unpublished properties

Using the store-and-load mechanism

The description of a form consists of a list of the form's properties, along with similar descriptions of each component on the form. Each component, including the form itself, is responsible for storing and loading its own description.

By default, when storing itself, a component writes the values of all its public and published properties that differ from their default values, in the order of their declaration. When loading itself, a component first constructs itself, setting all properties to their default values, then reads the stored, non-default property values.

This default mechanism serves the needs of most components, and requires no action at all on the part of the component writer. There are several ways you can customize the storing and loading process to suit the needs of your particular components, however.

Specifying default values

Kylix components save their property values only if those values differ from the defaults. If you do not specify otherwise, Kylix assumes a property has no default value, meaning the component always stores the property, whatever its value.

To specify a default value for a property, add the **default** directive and the new default value to the end of the property declaration.

You can also specify a default value when redeclaring a property. In fact, one reason to redeclare a property is to designate a different default value.

Note Specifying the default value does not automatically assign that value to the property on creation of the object. You must make sure that the component's constructor assigns the necessary value. A property whose value is not set by a component's constructor assumes a zero value—that is, whatever value the property assumes when its storage memory is set to 0. Thus numeric values default to 0, Boolean values to *False*, pointers to **nil**, and so on. If there is any doubt, assign a value in the constructor.

The following code shows a component declaration that specifies a default value for the *Align* property and the implementation of the component's constructor that sets the default value. In this case, the new component is a special case of the standard panel component that will be used for status bars in a window, so its default alignment should be to the bottom of its owner.

```
type
  TStatusBar = class(TPanel)
  public
    constructor Create(AOwner: TComponent); override;       { override to set new default }
  published
    property Align default alBottom;                   { redeclare with new default value }
  end;
⋮
constructor TStatusBar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                            { perform inherited initialization }
  Align := alBottom;                                 { assign new default value for Align }
end;
```

Determining what to store

You can control whether Kylix stores each of your components' properties. By default, all properties in the published part of the class declaration are stored. You can choose not to store a given property at all, or you can designate a function that determines dynamically whether to store the property.

To control whether Kylix stores a property, add the **stored** directive to the property declaration, followed by *True*, *False*, or the name of a Boolean function.

The following code shows a component that declares three new properties. One is always stored, one is never stored, and the third is stored depending on the value of a Boolean function:

```
type
  TSampleComponent = class(TComponent)
  protected
    function StoreIt: Boolean;
  public
  ⋮
  published
    property Important: Integer stored True;          { always stored }
    property Unimportant: Integer stored False;       { never stored }
    property Sometimes: Integer stored StoreIt;       { storage depends on function value }
  end;
```

Initializing after loading

After a component reads all its property values from its stored description, it calls a virtual method named *Loaded*, which performs any required initializations. The call to *Loaded* occurs before the form and its controls are shown, so you do not need to worry about initialization causing flicker on the screen.

To initialize a component after it loads its property values, override the *Loaded* method.

Note The first thing to do in any *Loaded* method is call the inherited *Loaded* method. This ensures that any inherited properties are correctly initialized before you initialize your own component.

Storing and loading unpublished properties

By default, only published properties are loaded and saved with a component. However, it is possible to load and save unpublished properties. This allows you to have persistent properties that do not appear in the Object Inspector. It also allows components to store and load property values that Kylix does not know how to read or write because the value of the property is too complex. For example, the *TStrings* object can't rely on Kylix's automatic behavior to store and load the strings it represents and must use the following mechanism.

You can save unpublished properties by adding code that tells Kylix how to load and save your property's value.

To write your own code to load and save properties, use the following steps:

1. Create methods to store and load the property value.
2. Override the *DefineProperties* method, passing those methods to a filer object.

Creating methods to store and load property values

To store and load unpublished properties, you must first create a method to store your property value and another to load your property value. You have two choices:

- Create a method of type *TWriterProc* to store your property value and a method of type *TReaderProc* to load your property value. This approach lets you take advantage of Kylix's built-in capabilities for saving and loading simple types. If your property value is built out of types that Kylix knows how to save and load, use this approach.
- Create two methods of type *TStreamProc*, one to store and one to load your property's value. *TStreamProc* takes a stream as an argument, and you can use the stream's methods to write and read your property values.

For example, consider a property that represents a component that is created at runtime. Kylix knows how to write this value, but does not do so automatically because the component is not created in the form designer. Because the streaming system can already load and save components, you can use the first approach. The following methods load and store the dynamically created component that is the value of a property named *MyCompProperty*:

```
procedure TSampleComponent.LoadCompProperty(Reader: TReader);
begin
  if Reader.ReadBoolean then
    MyCompProperty := Reader.ReadComponent(nil);
end;
procedure TSampleComponent.StoreCompProperty(Writer: TWriter);
begin
  Writer.WriteBoolean(MyCompProperty <> nil);
  if MyCompProperty <> nil then
    Writer.WriteComponent(MyCompProperty);
end;
```

Overriding the DefineProperties method

Once you have created methods to store and load your property value, you can override the component's *DefineProperties* method. Kylix calls this method when it loads or stores the component. In the *DefineProperties* method, you must call the *DefineProperty* method or the *DefineBinaryProperty* method of the current filer, passing it the method to use for loading or saving your property value. If your load and store methods are of type *TWriterProc* and type *TReaderProc*, then you call the filer's *DefineProperty* method. If you created methods of type *TStreamProc*, call *DefineBinaryProperty* instead.

No matter which method you use to define the property, you pass it the methods that store and load your property value as well as a boolean value indicating whether the property value needs to be written. If the value can be inherited or has a default value, you do not need to write it.

For example, given the *LoadCompProperty* method of type *TReaderProc* and the *StoreCompProperty* method of type *TWriterProc*, you would override *DefineProperties* as follows:

```
procedure TSampleComponent.DefineProperties(Filer: TFiler);
  function DoWrite: Boolean;
  begin
    if Filer.Ancestor <> nil then { check Ancestor for an inherited value }
    begin
      if TSampleComponent(Filer.Ancestor).MyCompProperty = nil then
        Result := MyCompProperty <> nil
      else if MyCompProperty = nil or
         TSampleComponent(Filer.Ancestor).MyCompProperty.Name <> MyCompProperty.Name then
        Result := True
      else Result := False;
    end
    else { no inherited value -- check for default (nil) value }
      Result := MyCompProperty <> nil;
  end;
begin
  inherited; { allow base classes to define properties }
  Filer.DefineProperty('MyCompProperty', LoadCompProperty, StoreCompProperty, DoWrite);
end;
```

Chapter

27

Creating events

An event is a link between an occurrence in the system (such as a user action or a change in focus) and a piece of code that responds to that occurrence. The responding code is an *event handler*, and is nearly always written by the application developer. Events let application developers customize the behavior of components without having to change the classes themselves. This is known as *delegation*.

Events for the most common user actions (such as mouse actions) are built into all the standard components, but you can also define new events. To create events in a component, you need to understand the following:

- What are events?
- Implementing the standard events
- Defining your own events

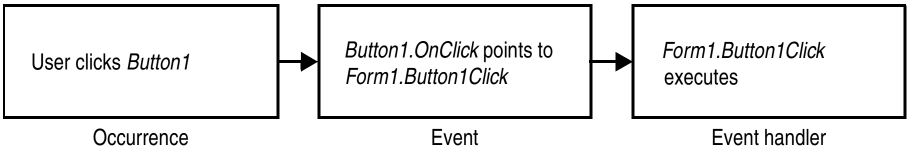
Events are implemented as properties, so you should already be familiar with the material in Chapter 26, “Creating properties,” before you attempt to create or change a component’s events.

What are events?

An event is a mechanism that links an occurrence to some code. More specifically, an event is a method pointer that points to a method in a specific class instance.

From the application developer’s perspective, an event is just a name related to a system occurrence, such as *OnClick*, to which specific code can be attached. For example, a push button called *Button1* has an *OnClick* event. By default, when you assign a value to the *OnClick* event, Kylix generates an event handler called *Button1Click* in the form that contains the button and assigns it to *OnClick*. When a click event occurs in the button, the button calls the method assigned to *OnClick*, in this case, *Button1Click*.

To write an event, you need to understand the following:


User clicks *Button1*
Button1.OnClick points to *Form1.Button1Click*
Form1.Button1Click executes
Occurrence
Event
Event handler

- Events are method pointers.
- Events are properties.
- Event types are method-pointer types
- Event-handler types are procedures
- Event handlers are optional.

Events are method pointers

Kylix uses method pointers to implement events. A method pointer is a special pointer type that points to a specific method in an instance object. As a component writer, you can treat the method pointer as a placeholder: When your code detects that an event occurs, you call the method (if any) specified by the user for that event.

Method pointers work just like any other procedural type, but they maintain a hidden pointer to an object. When the application developer assigns a handler to a component's event, the assignment is not just to a method with a particular name, but rather to a method in a specific instance object. That object is usually the form that contains the component, but it need not be.

All controls, for example, inherit a dynamic method called *Click* for handling click events:

```
procedure Click; dynamic;
```

The implementation of *Click* calls the user's click-event handler, if one exists. If the user has assigned a handler to a control's *OnClick* event, clicking the control results in that method being called. If no handler is assigned, nothing happens.

Events are properties

Components use properties to implement their events. Unlike most other properties, events typically do not use methods to implement their read and write parts. Instead, event properties use a private class field of the same type as the property.

By convention, the field's name is the name of the property preceded by the letter *F*. For example, the *OnClick* method's pointer is stored in a field called *FOnClick* of type *TNotifyEvent*, and the declaration of the *OnClick* event property looks like this:

```
type
  TControl = class(TComponent)
  private
    FOnClick: TNotifyEvent;                 { declare a field to hold the method pointer }
    ⋮
  protected
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
  end;
```

To learn about *TNotifyEvent* and other event types, see the next section, "Event types are method-pointer types".

As with any other property, you can set or change the value of an event at runtime. The main advantage to having events be properties, however, is that component users can assign handlers to events at design time, using the Object Inspector.

Event types are method-pointer types

Because an event is a pointer to an event handler, the type of the event property must be a method-pointer type. Similarly, any code to be used as an event handler must be an appropriately typed method of an object.

All event-handler methods are procedures. To be compatible with an event of a given type, an event-handler method must have the same number and type of parameters, in the same order, passed in the same way.

CLX defines method types for all its standard events. When you create your own events, you can use an existing type if that is appropriate, or define one of your own.

Event-handler types are procedures

Although the compiler allows you to declare method-pointer types that are functions, you should never do so for handling events. Because an empty function returns an undefined result, an empty event handler that was a function might not always be valid. For this reason, all your events and their associated event handlers should be procedures.

Although an event handler cannot be a function, you can still get information from the application developer's code using **var** parameters. When doing this, make sure you assign a valid value to the parameter before calling the handler so you don't require the user's code to change the value.

An example of passing **var** parameters to an event handler is the *OnKeyPress* event, of type *TKeyPressEvent*. *TKeyPressEvent* defines two parameters, one to indicate which object generated the event, and one to indicate which key was pressed:

```
type
  TKeyPressEvent = procedure(Sender: TObject; var Key: Char) of object;
```

Normally, the *Key* parameter contains the character pressed by the user. Under certain circumstances, however, the user of the component may want to change the character. One example might be to force all characters to uppercase in an editor. In that case, the user could define the following handler for keystrokes:

```
procedure TForm1.Edit1KeyPressed(Sender: TObject; var Key: Char);
begin
  Key := UpCase(Key);
end;
```

You can also use **var** parameters to let the user override the default handling.

Event handlers are optional

When creating events, remember that developers using your components may not attach handlers to them. This means that your component should not fail or generate errors simply because there is no handler attached to a particular event. (The mechanics of calling handlers and dealing with events that have no attached handler are explained in “Calling the event” on page 27-8.)

Events happen almost constantly in a GUI application. Just moving the mouse pointer across a visual component causes numerous mouse-move events, which the component surfaces as *OnMouseMove* events. In most cases, developers do not want to handle the mouse-move events, and this should not cause a problem. The components you create should not require handlers for their events.

CLX components have events that are written in such a way as to minimize the chance of an event handler generating errors. Obviously, you cannot protect against logic errors in application code, but you can ensure that data structures are initialized before calling events so that application developers do not try to access invalid data.

Implementing the standard events

The controls that come with Kylix inherit events for the most common occurrences. These are called the *standard events*. Although all these events are built into the controls, they are often **protected**, meaning developers cannot attach handlers to them. When you create a control, you can choose to make events visible to users of your control.

There are three things you need to consider when incorporating the standard events into your controls:

- Identifying standard events
- Making events visible
- Changing the standard event handling

Identifying standard events

There are two categories of standard events: those defined for all controls and those defined only for the widget controls.

Standard events for all controls

The most basic events are defined in the class *TControl*. All controls, whether widget-based, graphical, or custom, inherit these events. The following events are available in all controls:

OnClick	*OnDragDrop*	*OnMouseMove*	*OnMouseWheelUp*
OnConstrainedResize	*OnDragOver*	*OnMouseUp*	*OnResize*
OnContextPopup	*OnEndDrag*	*OnMouseWheel*	*OnStartDrag*
OnDblClick	*OnMouseDown*	*OnMouseWheelDown*	

The standard events have corresponding protected virtual methods declared in *TControl*, with names that correspond to the event names. For example, *OnClick* events call a method named *Click*, and *OnEndDrag* events call a method named *DoEndDrag*.

Standard events for widget-based controls

In addition to the events common to all controls, standard widget-based controls (those that descend from *TWidgetControl*) have the following events:

OnEnter	*OnKeyDown*	*OnKeyPress*
OnKeyUp	*OnExit*	

Like the standard events in *TControl*, the widget-based events have corresponding methods.

Making events visible

The declarations of the standard events in *TControl* and *TWidgetControl* are protected, as are the methods that correspond to them. If you are inheriting from one of these abstract classes and want to make their events accessible at runtime or design time, you need to redeclare the events as either public or published.

Redeclaring a property without specifying its implementation keeps the same implementation methods, but changes the protection level. You can, therefore, take an event that is defined in *TControl* but not made visible, and surface it by declaring it as public or published.

For example, to create a component that surfaces the *OnClick* event at design time, add the following to the component's class declaration.

```
type
  TMyControl = class(TCustomControl)
  ⋮
  published
    property OnClick;
  end;
```

Changing the standard event handling

If you want to change the way your component responds to a certain kind of event, you might be tempted to write some code and assign it to the event. As an application developer, that is exactly what you would do. But when you are creating a component, you must keep the event available for developers who use the component.

This is the reason for the protected implementation methods associated with each of the standard events. By overriding the implementation method, you can modify the internal event handling; and by calling the inherited method you can maintain the standard handling, including the event for the application developer's code.

The order in which you call the methods is significant. As a rule, call the inherited method first, allowing the application developer's event-handler to execute before your customizations (and in some cases, to keep the customizations from executing). There may be times when you want to execute your code before calling the inherited method, however. For example, if the inherited code is somehow dependent on the status of the component and your code changes that status, you should make the changes and then allow the user's code to respond to them.

Suppose you are writing a component and you want to modify the way it responds to mouse clicks. Instead of assigning a handler to the *OnClick* event as a application developer would, you override the protected method *Click*:

```
procedure click override       { forward declaration }
⋮
procedure TMyControl.Click;
begin
  inherited Click;             { perform standard handling, including calling handler }
...                           { your customizations go here }
end;
```

Defining your own events

Defining entirely new events is relatively unusual. There are times, however, when a component introduces behavior that is entirely different from that of any other component, so you will need to define an event for it.

There are the issues you will need to consider when defining an event:

- Triggering the event
- Defining the handler type
- Declaring the event
- Calling the event

Triggering the event

You need to know what triggers the event. For some events, the answer is obvious. For example, a mouse-down event occurs when the user presses the left button on the mouse. When notified of an event by the system, the component calls its *MouseDown* method, which in turn calls any code the user has attached to the *OnMouseDown* event.

Some events, however, are less clearly tied to specific external occurrences. For example, a scroll bar has an *OnChange* event, which is triggered by several kinds of occurrence, including keystrokes, mouse clicks, and changes in other controls. When defining your events, you must ensure that all the appropriate occurrences call the method that triggers the event.

Two kinds of events

Events can be widget events, such as highlighting a menu item, or system events, such as working with timers or key presses.

Widget events are actions that are generated by user interaction with a widget. Widget events generate a signal that is passed onto the CLX component for processing. The CLX component has an associated event handler installed for the signal being passed to it. Examples of widget events are *OnChange* (the user changed text in an edit control), *OnHightlighted* (the user highlighted a menu item on a menu), and *OnReturnPressed* (the user pressed Enter in a memo control). These events are always tied to specific widgets and are defined within those widgets.

System events are events that the operating system generates. For example, the *OnTimer* event (the *TTimer* component issues one of these events whenever a predefined interval has elapsed), the *OnCreate* event (the component is being created), the *OnPaint* event (a component or widget needs to be redrawn), *OnKeyPress* event (a key was pressed on the keyboard), and so on. These are events that the application programmer must respond to if they are not handled as you want them to be.

You have total control over the triggering of the events you define. Define the events with care so that developers are able to understand and use them.

Defining the handler type

Once you determine when the event occurs, you must define how you want the event handled. This means determining the type of the event handler. In most cases, handlers for events you define yourself are either simple notifications or event-specific types. It is also possible to get information back from the handler.

Simple notifications

A notification event is one that only tells you that the particular event happened, with no specific information about when or where. Notifications use the type *TNotifyEvent*, which carries only one parameter, the sender of the event. All a handler for a notification "knows" about the event is what kind of event it was, and what component the event happened to. For example, click events are notifications. When you write a handler for a click event, all you know is that a click occurred and which component was clicked.

Notification is a one-way process. There is no mechanism to provide feedback or prevent further handling of a notification.

Event-specific handlers

In some cases, it is not enough to know which event happened and what component it happened to. For example, if the event is a key-press event, it is likely that the handler will want to know which key the user pressed. In these cases, you need handler types that include parameters for additional information.

If your event was generated in response to a message, it is likely that the parameters you pass to the event handler come directly from the message parameters.

Returning information from the handler

Because all event handlers are procedures, the only way to pass information back from a handler is through a **var** parameter. Your components can use such information to determine how or whether to process an event after the user's handler executes.

For example, all the key events (*OnKeyDown*, *OnKeyUp*, and *OnKeyPress*) pass by reference the value of the key pressed in a parameter named *Key*. The event handler can change *Key* so that the application sees a different key as being involved in the event. This is a way to force typed characters to uppercase, for example.

Declaring the event

Once you have determined the type of your event handler, you are ready to declare the method pointer and the property for the event. Be sure to give the event a meaningful and descriptive name so that users can understand what the event does. Try to be consistent with names of similar properties in other components.

Event names start with "On"

The names of most events in Kylix begin with "On." This is just a convention; the compiler does not enforce it. The Object Inspector determines that a property is an event by looking at the type of the property: all method-pointer properties are assumed to be events and appear on the Events page.

Developers expect to find events in the alphabetical list of names starting with "On." Using other kinds of names is likely to confuse them.

Note The main exception to this rule is that many events that occur before and after some occurrence begin with "Before" and "After".

Calling the event

You should centralize calls to an event. That is, create a virtual method in your component that calls the application's event handler (if it assigns one) and provides any default handling.

Putting all the event calls in one place ensures that someone deriving a new component from yours can customize event handling by overriding a single method, rather than searching through your code for places where you call the event.

There are two other considerations when calling the event:

- Empty handlers must be valid.
- Users can override default handling.

Empty handlers must be valid

You should never create a situation in which an empty event handler causes an error, nor should the proper functioning of your component depend on a particular response from the application's event-handling code.

An empty handler should produce the same result as no handler at all. So the code for calling an application's event handler should look like this:

```
if Assigned(OnClick) then OnClick(Self);
...  { perform default handling }
```

You should *never* have something like this:

```
if Assigned(OnClick) then OnClick(Self)
else { perform default handling };
```

Users can override default handling

For some kinds of events, developers may want to replace the default handling or even suppress all responses. To allow this, you need to pass an argument by reference to the handler and check for a certain value when the handler returns.

This is in keeping with the rule that an empty handler should have the same effect as no handler at all. Because an empty handler will not change the values of arguments passed by reference, the default handling always takes place after calling the empty handler.

When handling key-press events, for example, application developers can suppress the component's default handling of the keystroke by setting the **var** parameter *Key* to a null character (#0). The logic for supporting this looks like

```
if Assigned(OnKeyPress) then OnKeyPress(Self, Key);
if Key <> #0 then ...  { perform default handling }
```

The actual code is a little different from this because it deals with system events, but the logic is the same. By default, the component calls any user-assigned handler, then performs its standard handling. If the user's handler sets *Key* to a null character, the component skips the default handling.

Chapter

28

Creating methods

Component methods are procedures and functions built into the structure of a class. Although there are essentially no restrictions on what you can do with the methods of a component, CLX does use some standards you should follow. These guidelines include

- Avoiding dependencies
- Naming methods
- Protecting methods
- Making methods virtual
- Declaring methods

In general, components should not contain many methods and you should minimize the number of methods that an application needs to call. The features you might be inclined to implement as methods are often better encapsulated into properties. Properties provide an interface that suits the Kylix environment and are accessible at design time.

Avoiding dependencies

At all times when writing components, minimize the preconditions imposed on the developer. To the greatest extent possible, developers should be able to do anything they want to a component, whenever they want to do it. There will be times when you cannot accommodate that, but your goal should be to come as close as possible.

This list gives you an idea of the kinds of dependencies to avoid:

- Methods that the user *must* call to use the component
- Methods that must execute in a particular order
- Methods that put the component into a state or mode where certain events or methods could be invalid

The best way to handle these situations is to ensure that you provide ways out of them. For example, if calling a method puts your component into a state where calling another method might be invalid, then write that second method so that if an application calls it when the component is in a bad state, the method corrects the state before executing its main code. At a minimum, you should raise an exception in cases when a user calls a method that is invalid.

In other words, if you create a situation where parts of your code depend on each other, the burden should be on *you* to be sure that using the code in incorrect ways does not cause problems. A warning message, for example, is preferable to a system failure if the user does not accommodate your dependencies.

Naming methods

Kylix imposes no restrictions on what you name methods or their parameters. There are a few conventions that make methods easier for application developers, however. Keep in mind that the nature of a component architecture dictates that many different kinds of people can use your components.

If you are accustomed to writing code that only you or a small group of programmers use, you might not think too much about how you name things. It is a good idea to make your method names clear because people unfamiliar with your code (and even unfamiliar with coding) might have to use your components.

Here are some suggestions for making clear method names:

- Make names descriptive. Use meaningful verbs.

  A name like *PasteFromClipboard* is much more informative than simply *Paste* or *PFC*.

- Function names should reflect the nature of what they return.

  Although it might be obvious to you as a programmer that a function named *X* returns the horizontal position of something, a name like *GetHorizontalPosition* is more universally understandable.

As a final consideration, make sure the method really needs to be a method. A good guideline is that method names have verbs in them. If you find that you create a lot of methods that do not have verbs in their names, consider whether those methods ought to be properties.

Protecting methods

All parts of classes, including fields, methods, and properties, have a level of protection or “visibility,” as explained in “Controlling access” on page 25-4. Choosing the appropriate visibility for a method is simple.

Most methods you write in your components are **public** or **protected**. You rarely need to make a method **private**, unless it is truly specific to that type of component, to the point that even derived components should not have access to it.

Methods that should be public

Any method that application developers need to call must be declared as **public**. Keep in mind that most method calls occur in event handlers, so methods should avoid tying up system resources or putting the operating system in a state where it cannot respond to the user.

Note Constructors and destructors should always be **public**.

Methods that should be protected

Any implementation methods for the component should be **protected** so that applications cannot call them at the wrong time. If you have methods that application code should not call, but that are called in derived classes, declare them as **protected**.

For example, suppose you have a method that relies on having certain data set up for it beforehand. If you make that method **public**, there is a chance that applications will call it before setting up the data. On the other hand, by making it **protected**, you ensure that applications cannot call it directly. You can then set up other, **public** methods that ensure that data setup occurs before calling the **protected** method.

Property-implementation methods should be declared as virtual **protected** methods. Methods that are so declared allow the application developers to override the property implementation, either augmenting its functionality or replacing it completely. Such properties are fully polymorphic. Keeping access methods **protected** ensures that developers do not accidentally call them, inadvertently modifying a property.

Abstract methods

Sometimes a method is declared as **abstract** in a component. In CLX, abstract methods usually occur in classes whose names begin with “custom”, such as *TCustomGrid*. Such classes are themselves abstract, in the sense that they are intended only for deriving descendant classes.

While you can create an instance object of a class that contains an abstract member, it is not recommended. Calling the abstract member leads to an *EAbstractError* exception.

The **abstract** directive is used to indicate parts of classes that should be surfaced and defined in descendant components; it forces Component writers to redeclare the abstract member in descendant classes before actual instances of the class can be created.

Making methods virtual

You make methods **virtual** when you want different types to be able to execute different code in response to the same method call.

If you create components intended to be used directly by application developers, you can probably make all your methods nonvirtual. On the other hand, if you create abstract components from which other components will be derived, consider making the added methods **virtual**. This way, derived components can override the inherited **virtual** methods.

Declaring methods

Declaring a method in a component is the same as declaring any class method.

To declare a new method in a component, you do two things:

- Add the declaration to the component's object-type declaration.
- Implement the method in the **implementation** part of the component's unit.

The following code shows a component that defines two new methods, one protected static method and one public virtual method.

```
type
  TSampleComponent = class(TControl)
  protected
    procedure MakeBigger;                              { declare protected static method }

  public
    function CalculateArea: Integer; virtual;            { declare public virtual method }
  end;
⋮
implementation
⋮
procedure TSampleComponent.MakeBigger;                          { implement first method }
begin
  Height := Height + 5;
  Width := Width + 5;
end;

function TSampleComponent.CalculateArea: Integer;              { implement second method }
begin
  Result := Width * Height;
end;
```

Chapter

29

Using graphics in components

Instead of forcing you to deal with graphics at a detailed level, CLX provides a simple yet complete interface: your component's *Canvas* property. The canvas has properties that represent the current pen, brush, and font.

The canvas manages resources for you, so you need not concern yourself with creating, selecting, and releasing things like pen handles. You just tell the canvas what kind of pen it should use, and it takes care of the rest.

One of the benefits of letting Kylix manage graphic resources is that it can cache resources for later use, which can speed up repetitive operations. For example, if you have a program that repeatedly creates, uses, and disposes of a particular kind of pen tool, you need to repeat those steps each time you use it. Because Kylix caches graphic resources, chances are good that a tool you use repeatedly is still in the cache, so instead of having to recreate a tool, Kylix uses an existing one.

An example of this is an application that has dozens of forms open, with hundreds of controls. Each of these controls might have one or more *TFont* properties. Though this could result in hundreds or thousands of instances of *TFont* objects, most applications wind up using only two or three font handles.

Using the canvas

The canvas class encapsulates graphics controls at several levels, including high-level functions for drawing individual lines, shapes, and text; and intermediate properties for manipulating the drawing capabilities of the canvas.

Table 29.1 summarizes the capabilities of the canvas.

Table 29.1 Canvas capability summary

Level	Operation	Tools
High	Drawing lines and shapes	Methods such as *MoveTo*, *LineTo*, *Rectangle*, and *Ellipse*
	Displaying and measuring text	*TextOut*, *TextHeight*, *TextWidth*, and *TextRect* methods
	Filling areas	*FillRect* and *FloodFill* methods
Intermediate	Customizing text and graphics	*Pen*, *Brush*, and *Font* properties
	Copying and merging images	*Draw*, *StretchDraw*, and *CopyRect* methods; *CopyMode* property
Low	Calling low-level graphics functions	*Handle* property

For detailed information on canvas classes and their methods and properties, see the online Help.

Working with pictures

Most of the graphics work you do in Kylix is limited to drawing directly on the canvases of components and forms. Kylix also provides for handling stand-alone graphic images, such as bitmaps, drawings (recorded drawing instructions), and icons.

There are three important aspects to working with pictures in Kylix:

- Using a picture, graphic, or canvas
- Loading and storing graphics

Using a picture, graphic, or canvas

There are three kinds of classes in Kylix that deal with graphics:

- A *canvas* represents a drawing surface on a form, graphic control, printer, or bitmap. A canvas is always a property of something else, never a stand-alone class.
- A *graphic* represents a graphic image of the sort usually found in a file or resource, such as a bitmap, icon, or drawing. Kylix defines classes *TBitmap*, *TDrawing*, and *TIcon*, all descended from a generic *TGraphic*. You can also define your own graphic classes. By defining a minimal standard interface for all graphics, *TGraphic* provides a simple mechanism for applications to use different kinds of graphics easily.
- A *picture* is a container for a graphic, meaning it could contain any of the graphic classes. That is, an item of type *TPicture* can contain a bitmap, an icon, a drawing, or a user-defined graphic type, and an application can access them all in the same way through the picture class. For example, the image control has a property called *Picture*, of type *TPicture*, that allows the control to display images from many kinds of graphics.

Keep in mind that a picture class always has a graphic, and a graphic might have a canvas. Normally, when dealing with a picture, you work only with the parts of the graphic class exposed through *TPicture*. If you need access to the specifics of the graphic class itself, you can refer to the picture's *Graphic* property.

Loading and storing graphics

All pictures and graphics in Kylix can load their images from files and store them back again (or into different files). You can load or store the image of a picture at any time.

To load an image into a picture from a file, call the picture's *LoadFromFile* method.

To save an image from a picture into a file, call the picture's *SaveToFile* method.

LoadFromFile and *SaveToFile* each take the name of a file as the only parameter. *LoadFromFile* uses the extension of the file name to determine what kind of graphic object it will create and load. *SaveToFile* saves whatever type of file is appropriate for the type of graphic object being saved.

Note You can also load images from and save them to a Qt MIME source, or a stream object.

To load a bitmap into an image control's picture, for example, pass the name of a bitmap file to the picture's *LoadFromFile* method:

```
procedure TForm1.LoadBitmapClick(Sender: TObject);
begin
  Image1.Picture.LoadFromFile('RANDOM.bmp');
end;
```

The picture recognizes bmp as a standard extension for bitmap files, so it creates its graphic as a *TBitmap*, then calls that graphic's *LoadFromFile* method. Because the graphic is a bitmap, it loads the image from the file as a bitmap.

Off-screen bitmaps

When drawing complex graphic images, a common technique in graphics programming is to create an off-screen bitmap, draw the image on the bitmap, and then copy the complete image from the bitmap to the final destination onscreen. Using an off-screen image reduces flicker caused by repeated drawing directly to the screen.

The bitmap class in Kylix, which represents bitmapped images in resources and files, can also work as an off-screen image.

There are two main aspects to working with off-screen bitmaps:

- Creating and managing off-screen bitmaps.
- Copying bitmapped images.

Creating and managing off-screen bitmaps

When creating complex graphic images, you should avoid drawing them directly on a canvas that appears onscreen. Instead of drawing on the canvas for a form or control, you can construct a bitmap object, draw on its canvas, and then copy its completed image to the onscreen canvas.

The most common use of an off-screen bitmap is in the *Paint* method of a graphic control. As with any temporary object, the bitmap should be protected with a **try**..**finally** block:

```
type
  TFancyControl = class(TGraphicControl)
  protected
    procedure Paint; override;                              { override the Paint method }
  end;

procedure TFancyControl.Paint;
var
  Bitmap: TBitmap;                        { temporary variable for the off-screen bitmap }
begin
  Bitmap := TBitmap.Create;                                 { construct the bitmap object }
  try
    { draw on the bitmap }
    { copy the result into the control's canvas }
  finally
    Bitmap.Free;                                              { destroy the bitmap object }
  end;
end;
```

Copying bitmapped images

Kylix provides four different ways to copy images from one canvas to another. Depending on the effect you want to create, you call different methods.

Table 29.2 summarizes the image-copying methods in canvas objects.

Table 29.2 Image-copying methods

To create this effect	Call this method
Copy an entire graphic.	Draw
Copy and resize a graphic.	StretchDraw
Copy part of a canvas.	CopyRect
Copy a graphic repeatedly to tile an area.	TiledDraw

Responding to changes

All graphic objects, including canvases and their owned objects (pens, brushes, and fonts) have events built into them for responding to changes in the object. By using these events, you can make your components (or the applications that use them) respond to changes by redrawing their images.

Responding to changes in graphic objects is particularly important if you publish them as part of the design-time interface of your components. The only way to ensure that the design-time appearance of the component matches the properties set in the Object Inspector is to respond to changes in the objects.

To respond to changes in a graphic object, assign a method to the class's *OnChange* event.

The shape component publishes properties representing the pen and brush it uses to draw its shape. The component's constructor assigns a method to the *OnChange* event of each, causing the component to refresh its image if either the pen or brush changes:

```
type
  TShape = class(TGraphicControl)
  public
    procedure StyleChanged(Sender: TObject);
  end;
⋮
implementation
⋮
constructor TShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                     { always call the inherited constructor! }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                                { construct the pen }
  FPen.OnChange := StyleChanged;                        { assign method to OnChange event }
  FBrush := TBrush.Create;                                          { construct the brush }
  FBrush.OnChange := StyleChanged;                      { assign method to OnChange event }
end;

procedure TShape.StyleChanged(Sender: TObject);
begin
  Invalidate();                                         { erase and repaint the component }
end;
```

Chapter

30

Making components available at design time

This chapter describes the steps for making the components you create available in the IDE. Making your components available at design time requires several steps:

- Registering components
- Adding palette bitmaps
- Adding property editors
- Adding component editors
- Compiling components into packages

Not all these steps apply to every component. For example, if you don't define any new properties or events, you don't need to provide Help for them. The only steps that are always necessary are registration and compilation.

Once your components have been registered and compiled into packages, they can be distributed to other developers and installed in the IDE. For information on installing packages in the IDE, see "Installing component packages" on page 11-5.

Registering components

Registration works on a compilation unit basis, so if you create several components in a single compilation unit, you can register them all at once.

To register a component, add a *Register* procedure to the unit. Within the *Register* procedure, you register the components and determine where to install them on the Component palette.

Note If you create your component by choosing Component | New Component in the IDE, the code required to register your component is added automatically.

The steps for manually registering a component are:

- Declaring the Register procedure
- Writing the Register procedure

Declaring the Register procedure

Registration involves writing a single procedure in the unit, which must have the name *Register*. The *Register* procedure must appear in the interface part of the unit, and (unlike the rest of Object Pascal) its name is case-sensitive.

The following code shows the outline of a simple unit that creates and registers new components:

```
unit MyBtns;
interface
type
  ...                                               { declare your component types here }
procedure Register;                          { this must appear in the interface section }
implementation
  ...                                              { component implementation goes here }
procedure Register;
begin
  ...                                                         { register the components }
end;
end.
```

Within the *Register* procedure, call *RegisterComponents* for each component you want to add to the Component palette. If the unit contains several components, you can register them all in one step.

Writing the Register procedure

Inside the *Register* procedure of a unit containing components, you must register each component you want to add to the Component palette. If the unit contains several components, you can register them at the same time.

To register a component, call the *RegisterComponents* procedure once for each page of the Component palette to which you want to add components. *RegisterComponents* involves three important things:

1. Specifying the components
2. Specifying the palette page
3. Using the RegisterComponents function

Specifying the components

Within the Register procedure, pass the component names in an open array, which you can construct inside the call to RegisterComponents.

```
RegisterComponents('Miscellaneous', [TMyComponent]);
```

You can register several components on the same page at once, or register components on different pages, as shown in the following code:

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TFirst, TSecond]);           { two on this page... }
  RegisterComponents('Assorted', [TThird]);                          { ...one on another... }
  RegisterComponents(LoadStr(srStandard), [TFourth]);    { ...and one on the Standard page }
end;
```

Specifying the palette page

The palette-page name is a string. If the name you give for the palette page does not already exist, the forms designer creates a new page with that name. Kylix stores the names of the standard pages in string-list resources so that international versions of the product can name the pages in their native languages. If you want to install a component on one of the standard pages, you should obtain the string for the page name by calling the *LoadStr* function, passing the constant representing the string resource for that page, such as *srSystem* for the System page.

Using the RegisterComponents function

Within the *Register* procedure, call *RegisterComponents* to register the components in the classes array. *RegisterComponents* is a function that takes two parameters: the name of a Component palette page and the array of component classes.

Set the Page parameter to the name of the page on the component palette where the components should appear. If the named page already exists, the components are added to that page. If the named page does not exist, the forms designer creates a new palette page with that name.

Call RegisterComponents from the implementation of the Register procedure in one of the units that defines the custom components. The units that define the components must then be compiled into a package and the package must be installed before the custom components are added to the component palette.

```
procedure Register;
begin
  RegisterComponents('System', [TSystem1, TSystem2]);                  {add to system page}
  RegisterComponents('MyCustomPage',[TCustom1, TCustom2]);                       { new page}
end;
```

Adding palette bitmaps

Every component needs a bitmap to represent it on the Component palette. If you don’t specify your own bitmap, Kylix uses a default bitmap.

Each bitmap should be 24 pixels square. There are a number of tools available on Linux for generating bitmap files. One approach is to use a tool such as Gimp to create the images as a .ppm file and then convert them to the .bmp format using ppmtobmp.

Because the palette bitmaps are needed only at design time, you don't compile them into the component's compilation unit. Instead, you supply them in a resource file with the same name as the unit, but with the extension .dcr (dynamic component resource).

For each unit that contains components you want to install, supply a palette bitmap resource file, and within each resource file, supply a bitmap for each component you register. The bitmap image has the same name as the component. For example, if you create a component named *TMyControl*, you need to create a .dcr or .res resource file that contains a bitmap called TMYCONTROL. The resource names are not case-sensitive, but by convention, they are usually in uppercase letters.

Keep the resource file in the same directory with the compiled files, so Kylix can find the bitmaps when it installs the components on the Component palette.

Adding property editors

The Object Inspector provides default editing for all types of properties. You can, however, provide an alternate editor for specific properties by writing and registering property editors. You can register property editors that apply only to the properties in the components you write, but you can also create editors that apply to all properties of a certain type.

At the simplest level, a property editor can operate in either or both of two ways: displaying and allowing the user to edit the current value as a text string, and displaying a dialog box that permits some other kind of editing.

Writing a property editor requires these five steps:

1. Deriving a property-editor class
2. Editing the property as text
3. Editing the property as a whole
4. Specifying editor attributes
5. Registering the property editor

Deriving a property-editor class

CLX defines several kinds of property editors, all of which descend from *TPropertyEditor*. When you create a property editor, your property-editor class can either descend directly from *TPropertyEditor* or indirectly through one of the property-editor classes described in Table 30.1.

Note All that is absolutely necessary for a property editor is that it descend from *TBasePropertyEditor* and that it support the *IProperty* interface. *TPropertyEditor*, however, provides a default implementation of the *IProperty* interface.

CLX also defines some very specialized property editors used by unique properties such as the component name. The listed property editors are the ones that are the most useful for user-defined properties.

Table 30.1 Predefined property-editor types

Type	Properties edited
TOrdinalProperty	All ordinal-property editors (those for integer, character, and enumerated properties) descend from *TOrdinalProperty*.
TIntegerProperty	All integer types, including predefined and user-defined subranges.
TCharProperty	*Char*-type and subranges of *Char*, such as 'A'..'Z'.
TEnumProperty	Any enumerated type.
TFloatProperty	All floating-point numbers.
TStringProperty	Strings.
TSetElementProperty	Individual elements in sets, shown as Boolean values
TSetProperty	All sets. Sets are not directly editable, but can expand into a list of set-element properties.
TClassProperty	Classes. Displays the name of the class and allows expansion of the class's properties.
TMethodProperty	Method pointers, most notably events.
TComponentProperty	Components in the same form. The user cannot edit the component's properties, but can point to a specific component of a compatible type.
TColorProperty	Component colors. Shows color constants if applicable, otherwise displays hexadecimal value. Drop-down list contains the color constants. Double-click opens the color-selection dialog box.
TFontNameProperty	Font names. The drop-down list displays all currently installed fonts.
TFontProperty	Fonts. Allows expansion of individual font properties as well as access to the font dialog box.

The following example shows the declaration of a simple property editor named *TMyPropertyEditor*:

```
type
  TFloatProperty = class(TPropertyEditor)
  public
    function AllEqual: Boolean; override;
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
  end;
```

Editing the property as text

All properties need to provide a string representation of their values for the Object Inspector to display. Most properties also allow the user to type in a new value for the property. *TPropertyEditor* and its descendants provide virtual methods you can override to convert between the text representation and the actual value.

The methods you override are called *GetValue* and *SetValue*. Your property editor also inherits methods for assigning and reading different sorts of values from *TPropertyEditor*. Some of these are listed in Table 30.2.

Table 30.2 Methods for reading and writing property values

Property type	Get method	Set method
Floating point	GetFloatValue	SetFloatValue
Method pointer (event)	GetMethodValue	SetMethodValue
Ordinal type	GetOrdValue	SetOrdValue
String	GetStrValue	SetStrValue

When you override a *GetValue* method, you call one of the Get methods, and when you override *SetValue*, you call one of the Set methods.

Displaying the property value

The property editor's *GetValue* method returns a string that represents the current value of the property. The Object Inspector uses this string in the value column for the property. By default, *GetValue* returns "unknown".

To provide a string representation of your property, override the property editor's *GetValue* method.

If the property is not a string value, *GetValue* must convert the value into a string representation.

Setting the property value

The property editor's *SetValue* method takes a string typed by the user in the Object Inspector, converts it into the appropriate type, and sets the value of the property. If the string does not represent a proper value for the property, *SetValue* should throw an exception and not use the improper value.

To read string values into properties, override the property editor's *SetValue* method.

SetValue should convert the string and validate the value before calling one of the Set methods.

Here are the *GetValue* and *SetValue* methods for *TIntegerProperty*. *Integer* is an ordinal type, so *GetValue* calls *GetOrdValue* and converts the result to a string. *SetValue* converts the string to an integer, performs some range checking, and calls *SetOrdValue*.

```
function TIntegerProperty.GetValue: string;
begin
  with GetTypeData(GetPropType)^ do
    if OrdType = otULong then // unsigned
      Result := IntToStr(Cardinal(GetOrdValue))
    else
      Result := IntToStr(GetOrdValue);
end;

procedure TIntegerProperty.SetValue(const Value: string);
  procedure Error(const Args: array of const);
```

```
    begin
      raise EPropertyError.CreateResFmt(@SOutOfRange, Args);
    end;
  var
    L: Int64;
  begin
    L := StrToInt64(Value);
    with GetTypeData(GetPropType)^ do
      if OrdType = otULong then
      begin   // unsigned compare and reporting needed
        if (L < Cardinal(MinValue)) or (L > Cardinal(MaxValue)) then
        // bump up to Int64 to get past the %d in the format string
          Error([Int64(Cardinal(MinValue)), Int64(Cardinal(MaxValue))]);
      end
      else if (L < MinValue) or (L > MaxValue) then
        Error([MinValue, MaxValue]);
    SetOrdValue(L);
  end;
```

The specifics of the particular examples here are less important than the principle: *GetValue* converts the value to a string; *SetValue* converts the string and validates the value before calling one of the "Set" methods.

Editing the property as a whole

You can optionally provide a dialog in which the user can visually edit a property. The most common use of property editors is for properties that are themselves classes. An example is the *Font* property, for which the user can open a font dialog box to choose all the attributes of the font at once.

To provide a whole-property editor dialog box, override the property-editor class's *Edit* method.

Edit methods use the same Get and Set methods used in writing *GetValue* and *SetValue* methods. In fact, an *Edit* method calls both a Get method and a Set method. Because the editor is type-specific, there is usually no need to convert the property values to strings. The editor generally deals with the value "as retrieved."

When the user clicks the '...' button next to the property or double-clicks the value column, the Object Inspector calls the property editor's *Edit* method.

Within your implementation of the *Edit* method, follow these steps:

1. Construct the editor you are using for the property.
2. Read the current value and assign it to the property using a Get method.
3. When the user selects a new value, assign that value to the property using a Set method.
4. Destroy the editor.

The *Color* properties found in most components use the standard color dialog box as a property editor. Here is the *Edit* method from *TColorProperty*, which invokes the dialog box and uses the result:

```
procedure TColorProperty.Edit;
var
  ColorDialog: TColorDialog;
begin
  ColorDialog := TColorDialog.Create(Application);              { construct the editor }
  try
    ColorDialog.Color := GetOrdValue;                          { use the existing value }
    if ColorDialog.Execute then                         { if the user OKs the dialog... }
      SetOrdValue(ColorDialog.Color);                  { ...use the result to set value }
  finally
    ColorDialog.Free;                                             { destroy the editor }
  end;
end;
```

Specifying editor attributes

The property editor must provide information that the Object Inspector can use to determine what tools to display. For example, the Object Inspector needs to know whether the property has subproperties or can display a list of possible values.

To specify editor attributes, override the property editor's *GetAttributes* method.

GetAttributes is a method that returns a set of values of type *TPropertyAttributes* that can include any or all of the following values:

Table 30.3 Property-editor attribute flags

Flag	Related method	Meaning if included
paValueList	GetValues	The editor can give a list of enumerated values.
paSubProperties	GetProperties	The property has subproperties that can display.
paDialog	Edit	The editor can display a dialog box for editing the entire property.
paMultiSelect	N/A	The property should display when the user selects more than one component.
paAutoUpdate	SetValue	Updates the component after every change instead of waiting for approval of the value.
paSortList	N/A	The Object Inspector should sort the value list.
paReadOnly	N/A	Users cannot modify the property value.
paRevertable	N/A	Enables the Revert to Inherited menu item on the Object Inspector's context menu. The menu item tells the property editor to discard the current property value and return to some previously established default or standard value.
paFullWidthName	N/A	The value does not need to be displayed. The Object Inspector uses its full width for the property name instead.

Table 30.3 Property-editor attribute flags (continued)

Flag	Related method	Meaning if included
paVolatileSubProperties	GetProperties	The Object Inspector refetches the values of all subproperties any time the property value changes.
paReference	GetComponent Value	The value is a reference to something else. When used in conjunction with paSubProperties the referenced object should be displayed as sub properties to this property.

Color properties are more versatile than most, in that they allow several ways for users to choose them in the Object Inspector: typing, selection from a list, and customized editor. *TColorProperty*'s *GetAttributes* method, therefore, includes several attributes in its return value:

```
function TColorProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paMultiSelect, paDialog, paValueList, paRevertable];
end;
```

Registering the property editor

Once you create a property editor, you need to register it with Kylix. Registering a property editor associates a type of property with a specific property editor. You can register the editor with all properties of a given type or just with a particular property of a particular type of component.

To register a property editor, call the *RegisterPropertyEditor* procedure.

RegisterPropertyEditor takes four parameters:

- A type-information pointer for the type of property to edit.

  This is always a call to the built-in function *TypeInfo*, such as `TypeInfo(TMyComponent)`.
- The type of the component to which this editor applies. If this parameter is **nil**, the editor applies to all properties of the given type.
- The name of the property. This parameter only has meaning if the previous parameter specifies a particular type of component. In that case, you can specify the name of a particular property in that component type to which this editor applies.
- The type of property editor to use for editing the specified property.

Here is an excerpt from the procedure that registers the editors for the standard components on the Component palette:

```
procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(TComponent), nil, '', TComponentProperty);
  RegisterPropertyEditor(TypeInfo(TComponentName), TComponent, 'Name',
    TComponentNameProperty);
  RegisterPropertyEditor(TypeInfo(TMenuItem), TMenu, '', TMenuItemProperty);
end;
```

The three statements in this procedure cover the different uses of *RegisterPropertyEditor*:

- The first statement is the most typical. It registers the property editor *TComponentProperty* for all properties of type *TComponent* (or descendants of *TComponent* that do not have their own editors registered). In general, when you register a property editor, you have created an editor for a particular type, and you want to use it for all properties of that type, so the second and third parameters are **nil** and an empty string, respectively.
- The second statement is the most specific kind of registration. It registers an editor for a specific property in a specific type of component. In this case, the editor is for the *Name* property (of type *TComponentName*) of all components.
- The third statement is more specific than the first, but not as limited as the second. It registers an editor for all properties of type *TMenuItem* in components of type *TMenu*.

Property categories

In the IDE, the Object Inspector lets you selectively hide and display properties based on property categories. The properties of new custom components can be fit into this scheme by registering properties in categories. Do this at the same time you register the component by calling *RegisterPropertyInCategory* or *RegisterPropertiesInCategory*. Use *RegisterPropertyInCategory* to register a single property. Use *RegisterPropertiesInCategory* to register multiple properties in a single function call. These functions are defined in the unit DesignIntf.

Note that it is not mandatory that you register properties or that you register all of the properties of a custom component when some are registered. Any property not explicitly associated with a category is included in the *TMiscellaneousCategory* category. Such properties are displayed or hidden in the Object Inspector based on that default categorization.

In addition to these two functions for registering properties, there is an *IsPropertyInCategory* function. This function is useful for creating localization utilities, in which you must determine whether a property is registered in a given property category.

Registering one property at a time

Register one property at a time and associate it with a property category using the *RegisterPropertyInCategory* function. *RegisterPropertyInCategory* comes in four overloaded variations, each providing a different set of criteria for identifying the property in the custom component to be associated with the property category.

The first variation lets you identify the property by the property’s name. The line below registers a property related to visual display of the component, identifying the property by its name, “AutoSize”.

```
RegisterPropertyInCategory('Visual', 'AutoSize');
```

The second variation is much like the first, except that it limits the category to only those properties of the given name that appear on components of a given type. The example below registers (into the ‘Help and Hints’ category) a property named “HelpContext” of a component of the custom class *TMyButton*.

```
RegisterPropertyInCategory('Help and Hints', TMyButton, 'HelpContext');
```

The third variation identifies the property using its type rather than its name. The example below registers a property based on its type, Integer.

```
RegisterPropertyInCategory('Visual', TypeInfo(Integer));
```

The final variation uses both the property’s type and its name to identify the property. The example below registers a property based on a combination of its type, *TBitmap*, and its name, ”Pattern”.

```
RegisterPropertyInCategory('Visual', TypeInfo(TBitmap), 'Pattern');
```

See the section Specifying property categories for a list of the available property categories and a brief description of their uses.

Registering multiple properties at once

Register multiple properties at one time and associate them with a property category using the *RegisterPropertiesInCategory* function. *RegisterPropertiesInCategory* comes in three overloaded variations, each providing a different set of criteria for identifying the property in the custom component to be associated with property categories.

The first variation lets you identify properties based on property name or type. The list is passed as an array of constants. In the example below, any property that either has the name “Text” or belongs to a class of type *TEdit* is registered in the category ‘Localizable’.

```
RegisterPropertiesInCategory('Localizable', ['Text', TEdit]);
```

The second variation lets you limit the registered properties to those that belong to a specific component. The list of properties to register include only names, not types. For example, the following code registers a number of properties into the ‘Help and Hints’ category for all components:

```
RegisterPropertiesInCategory('Help and Hints', TComponent, ['HelpContext', 'Hint',
'ParentShowHint', 'ShowHint']);
```

The third variation lets you limit the registered properties to those that have a specific type. As with the second variation, the list of properties to register can include only names:

```
RegisterPropertiesInCategory('Localizable', TypeInfo(String), ['Text', 'Caption']);
```

See the section Specifying property categories for a list of the available property categories and a brief description of their uses.

Specifying property categories

When you register properties in a category, you can use any string you want as the name of the category. If you use a string that has not been used before, the Object Inspector generates a new property category class with that name. You can also, however, register properties into one of the categories that are built-in. The built-in property categories are described in Table 30.4.

Table 30.4 Property categories

Category	**Purpose**
Action	Properties related to runtime actions; the *Enabled* and *Hint* properties of *TEdit* are in this category.
Database	Properties related to database operations; the *DatabaseName* and *SQL* properties of *TQuery* are in this category.
Drag, Drop, and Docking	Properties related to drag-n-drop and docking operations; the *DragCursor* and *DragKind* properties of *TImage* are in this category.
Help and Hints	Properties related to using online help or hints; the *HelpContext* and *Hint* properties of *TMemo* are in this category.
Layout	Properties related to the visual display of a control at design-time; the *Top* and *Left* properties of *TDBEdit* are in this category.
Legacy	Properties related to obsolete operations; the *Ctl3D* and *ParentCtl3D* properties of *TComboBox* are in this category.
Linkage	Properties related to associating or linking one component to another; the *DataSet* property of *TDataSource* is in this category.
Locale	Properties related to international locales; the *BiDiMode* and *ParentBiDiMode* properties of *TMainMenu* are in this category.
Localizable	Properties that may require modification in localized versions of an application. Many string properties (such as *Caption*) are in this category, as are properties that determine the size and position of controls.
Visual	Properties related to the visual display of a control at runtime; the *Align* and *Visible* properties of *TScrollBox* are in this category.
Input	Properties related to the input of data (need not be related to database operations); the *Enabled* and *ReadOnly* properties of *TEdit* are in this category.
Miscellaneous	Properties that do not fit a category or do not need to be categorized (and properties not explicitly registered to a specific category); the AllowAllUp and Name properties of TSpeedButton are in this category.

Using the IsPropertyInCategory function

An application can query the existing registered properties to determine whether a given property is already registered in a specified category. This can be especially useful in situations like a localization utility that checks the categorization of properties preparatory to performing its localization operations. Two overloaded variations of the *IsPropertyInCategory* function are available, allowing for different criteria in determining whether a property is in a category.

The first variation lets you base the comparison criteria on a combination of the class type of the owning component and the property's name. In the command line below, for *IsPropertyInCategory* to return *True*, the property must belong to a *TCustomEdit* descendant, have the name "Text", and be in the property category `'Localizable'`.

```
IsItThere := IsPropertyInCategory('Localizable', TCustomEdit, 'Text');
```

The second variation lets you base the comparison criteria on a combination of the class name of the owning component and the property's name. In the command line below, for *IsPropertyInCategory* to return *True*, the property must be a *TCustomEdit* descendant, have the name "Text", and be in the property category `'Localizable'`.

```
IsItThere := IsPropertyInCategory('Localizable', 'TCustomEdit', 'Text');
```

Adding component editors

Component editors determine what happens when the component is double-clicked in the designer and add commands to the context menu that appears when the component is right-clicked. They can also copy your component to the clipboard in custom formats.

If you do not give your components a component editor, Kylix uses the default component editor. The default component editor is implemented by the class *TDefaultEditor*. *TDefaultEditor* does not add any new items to a component's context menu. When the component is double-clicked, *TDefaultEditor* searches the properties of the component and generates (or navigates to) the first event handler it finds.

To add items to the context menu, change the behavior when the component is double-clicked, or add new clipboard formats, derive a new class from *TComponentEditor* and register its use with your component. In your overridden methods, you can use the *Component* property of *TComponentEditor* to access the component that is being edited.

Adding a custom component editor consists of the steps:

- Adding items to the context menu
- Changing the double-click behavior
- Adding clipboard formats
- Registering the component editor

Adding items to the context menu

When the user right-clicks the component, the *GetVerbCount* and *GetVerb* methods of the component editor are called to build context menu. You can override these methods to add commands (verbs) to the context menu.

Adding items to the context menu requires the steps:

- Specifying menu items
- Implementing commands

Specifying menu items

Override the *GetVerbCount* method to return the number of commands you are adding to the context menu. Override the *GetVerb* method to return the strings that should be added for each of these commands. When overriding *GetVerb*, add an ampersand (&) to a string to cause the following character to appear underlined in the context menu and act as a shortcut key for selecting the menu item. Be sure to add an ellipsis (...) to the end of a command if it brings up a dialog. *GetVerb* has a single parameter that indicates the index of the command.

The following code overrides the *GetVerbCount* and *GetVerb* methods to add two commands to the context menu.

```
function TMyEditor.GetVerbCount: Integer;
begin
  Result := 2;
end;

function TMyEditor.GetVerb(Index: Integer): String;
begin
  case Index of
    0: Result := ”&DoThis ...”;
    1: Result := ”Do&That”;
  end;
end;
```

Note Be sure that your *GetVerb* method returns a value for every possible index indicated by *GetVerbCount*.

Implementing commands

When the command provided by *GetVerb* is selected in the designer, the *ExecuteVerb* method is called. For every command you provide in the *GetVerb* method, implement an action in the *ExecuteVerb* method. You can access the component that is being edited using the *Component* property of the editor.

For example, the following *ExecuteVerb* method implements the commands for the *GetVerb* method in the previous example.

```
procedure TMyEditor.ExecuteVerb(Index: Integer);
var
  MySpecialDialog: TMyDialog;
begin
  case Index of
    0: begin
         MyDialog := TMySpecialDialog.Create(Application);       { instantiate the editor }
         if MySpecialDialog.Execute then;                 { if the user OKs the dialog... }
           MyComponent.FThisProperty := MySpecialDialog.ReturnValue;   { ...use the value }
         MySpecialDialog.Free;                                         { destroy the editor }
       end;
    1: That;                                                          { call the That method }
  end;
end;
```

Changing the double-click behavior

When the component is double-clicked, the *Edit* method of the component editor is called. By default, the *Edit* method executes the first command added to the context menu. Thus, in the previous example, double-clicking the component executes the *DoThis* command.

While executing the first command is usually a good idea, you may want to change this default behavior. For example, you can provide an alternate behavior if

- you are not adding any commands to the context menu.
- you want to display a dialog that combines several commands when the component is double-clicked.

Override the *Edit* method to specify a new behavior when the component is double-clicked. For example, the following *Edit* method brings up a font dialog when the user double-clicks the component:

```
procedure TMyEditor.Edit;
var
  FontDlg: TFontDialog;
begin
  FontDlg := TFontDialog.Create(Application);
  try
    if FontDlg.Execute then
      MyComponent.FFont.Assign(FontDlg.Font);
  finally
    FontDlg.Free
  end;
end;
```

Note If you want a double-click on the component to display the Code editor for an event handler, use *TDefaultEditor* as a base class for your component editor instead of *TComponentEditor*. Then, instead of overriding the *Edit* method, override the protected *TDefaultEditor.EditProperty* method instead. *EditProperty* scans through the event handlers of the component, and brings up the first one it finds. You can change this to look a particular event instead. For example:

```
procedure TMyEditor.EditProperty(PropertyEditor: TPropertyEditor;
  Continue, FreeEditor: Boolean)
begin
  if (PropertyEditor.ClassName = ‘TMethodProperty’) and
    (PropertyEditor.GetName = ‘OnSpecialEvent’) then
    // DefaultEditor.EditProperty(PropertyEditor, Continue, FreeEditor);
end;
```

Adding clipboard formats

By default, when a user chooses Copy while a component is selected in the IDE, the component is copied in Kylix’s internal format. It can then be pasted into another form or data module. Your component can copy additional formats to the Clipboard by overriding the *Copy* method.

For example, the following *Copy* method allows a *TImage* component to copy its picture to the Clipboard. This picture is ignored by the Kylix IDE, but can be pasted into other applications.

```
procedure TMyComponent.Copy;
var
  Data: TMemoryStream;
begin
  Data := TMemoryStream.Create;
  try
    TImage(Component).Picture.SaveToStream(Data);
    Data.Position := 0;
    Clipboard.SetFormat('image/delphi.bitmap', Data);
  finally
    Data.Free;
end;
```

Registering the component editor

Once the component editor is defined, it can be registered to work with a particular component class. A registered component editor is created for each component of that class when it is selected in the form designer.

To create the association between a component editor and a component class, call *RegisterComponentEditor*. *RegisterComponentEditor* takes the name of the component class that uses the editor, and the name of the component editor class that you have defined. For example, the following statement registers a component editor class named *TMyEditor* to work with all components of type *TMyComponent*:

```
RegisterComponentEditor(TMyComponent, TMyEditor);
```

Place the call to *RegisterComponentEditor* in the *Register* procedure where you register your component. For example, if a new component named *TMyComponent* and its component editor *TMyEditor* are both implemented in the same unit, the following code registers the component and its association with the component editor.

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TMyComponent);
  RegisterComponentEditor(classes[0], TMyEditor);
end;
```

Compiling components into packages

Once your components are registered, you must compile them as packages before they can be installed in the IDE. A package can contain one or several components as well as custom property editors. For more information about packages, see Chapter 11, “Working with packages and components”.

To create and compile a package, see “Creating and editing packages” on page 11-6. Put the source-code units for your custom components in the package’s Contains list. If your components depend on other packages, include those packages in the Requires list.

To install your components in the IDE, see “Installing component packages” on page 11-5.

Chapter

31

Modifying an existing component

The easiest way to create a component is to derive it from a component that does nearly everything you want, then make whatever changes you need. What follows is a simple example that modifies the standard memo component to create a memo that does not wrap words by default.

The value of the memo component's *WordWrap* property is initialized to *True*. If you frequently use non-wrapping memos, you can create a new memo component that does not wrap words by default.

Note To modify published properties or save specific event handlers for an existing component, it is often easier to use a *component template* rather than create a new class.

Modifying an existing component takes only two steps:

- Creating and registering the component
- Modifying the component class

Creating and registering the component

Creation of every component begins the same way: you create a unit, derive a component class, register it, and install it on the Component palette. This process is outlined in "Creating a new component" on page 24-8.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *Memos*.
- Derive a new component type called *TWrapMemo*, descended from *TMemo*.
- Register *TWrapMemo* on the Samples page of the Component palette.

- The resulting unit should look like this:

```
unit Memos;
interface
uses
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs, QStdCtrls;
type
  TWrapMemo = class(TMemo)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TWrapMemo]);
end;
end.
```

If you compile and install the new component now, it behaves exactly like its ancestor, *TMemo*. In the next section, you will make a simple change to your component.

Modifying the component class

Once you have created a new component class, you can modify it in almost any way. In this case, you will change only the initial value of one property in the memo component. This involves two small changes to the component class:

- Overriding the constructor
- Specifying the new default property value

The constructor actually sets the value of the property. The default tells Kylix what values to store in the form file. Kylix stores only values that differ from the default, so it is important to perform both steps.

Overriding the constructor

When a component is placed on a form at design time, or when an application constructs a component at runtime, the component's constructor sets the property values. When a component is loaded from a form file, the application sets any properties changed at design time.

Note When you override a constructor, the new constructor must call the inherited constructor before doing anything else. For more information, see "Overriding methods" on page 25-8.

For this example, your new component needs to override the constructor inherited from *TMemo* to set the *WordWrap* property to *False*. To achieve this, add the constructor override to the forward declaration, then write the new constructor in the **implementation** part of the unit:

```
type
  TWrapMemo = class(TMemo)
  public                                          { constructors are always public }
    constructor Create(AOwner: TComponent); override; { this syntax is always the same }
  end;
⋮
constructor TWrapMemo.Create(AOwner: TComponent);     { this goes after implementation }
begin
  inherited Create(AOwner);                       { ALWAYS do this first! }
  WordWrap := False;                              { set the new desired value }
end;
```

Now you can install the new component on the Component palette and add it to a form. Note that the *WordWrap* property is now initialized to *False*.

If you change an initial property value, you should also designate that value as the default. If you fail to match the value set by the constructor to the specified default value, Kylix cannot store and restore the proper value.

Specifying the new default property value

When Kylix stores a description of a form in a form file, it stores the values only of properties that differ from their defaults. Storing only the differing values keeps the form files small and makes loading the form faster. If you create a property or change the default value, it is a good idea to update the property declaration to include the new default. Form files, loading, and default values are explained in more detail in Chapter 30, "Making components available at design time."

To change the default value of a property, redeclare the property name, followed by the directive **default** and the new default value. You don't need to redeclare the entire property, just the name and the default value.

For the word-wrapping memo component, you redeclare the *WordWrap* property in the **published** part of the object declaration, with a default value of *False*:

```
type
  TWrapMemo = class(TMemo)
  ⋮
  published
    property WordWrap default False;
  end;
```

Specifying the default property value does not affect the workings of the component. You must still initialize the value in the component's constructor. Redeclaring the default ensures that Kylix knows when to write *WordWrap* to the form file.

Chapter

32

Creating a graphic component

A graphic control is a simple kind of component. Because a purely graphic control never receives focus, it does not have or need its own window. Users can still manipulate the control with the mouse, but there is no keyboard interface.

The graphic component presented in this chapter is *TShape*, the shape component on the Additional page of the Component palette. Although the component created is identical to the standard shape component, you need to call it something different to avoid duplicate identifiers. This chapter calls its shape component *TSampleShape* and shows you all the steps involved in creating the shape component:

- Creating and registering the component
- Publishing inherited properties
- Adding graphic capabilities

Creating and registering the component

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in "Creating a new component" on page 24-8.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *Shapes*.
- Derive a new component type called *TSampleShape*, descended from *TGraphicControl*.

- Register *TSampleShape* on the Samples page of the Component palette.

  The resulting unit should look like this:

```
unit Shapes;
interface
uses SysUtils, Types, Classes, QGraphics, QControls, QForms;
type
  TSampleShape = class(TGraphicControl)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponent('Samples', [TSampleShape]);
end;
end.
```

Publishing inherited properties

Once you derive a component type, you can decide which of the properties and events declared in the protected parts of the ancestor class you want to surface in the new component. *TGraphicControl* already publishes all the properties that enable the component to function as a control, so all you need to publish is the ability to respond to mouse events and handle drag-and-drop.

Publishing inherited properties and events is explained in “Publishing inherited properties” on page 26-3 and “Making events visible” on page 27-5. Both processes involve redeclaring just the name of the properties in the published part of the class declaration.

For the shape control, you can publish the three mouse events, the three drag-and-drop events, and the two drag-and-drop properties:

```
type
  TSampleShape = class(TGraphicControl)
  published
    property DragCursor;        { drag-and-drop properties }
    property DragMode;
    property OnDragDrop;        { drag-and-drop events }
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;       { mouse events }
    property OnMouseMove;
    property OnMouseUp;
  end;
```

The sample shape control now makes mouse and drag-and-drop interactions available to its users.

Adding graphic capabilities

Once you have declared your graphic component and published any inherited properties you want to make available, you can add the graphic capabilities that distinguish your component. You have two tasks to perform when creating a graphic control:

1 Determining what to draw.
2 Drawing the component image.

In addition, for the shape control example, you will add some properties that enable application developers to customize the appearance of the shape at design time.

Determining what to draw

A graphic control can change its appearance to reflect a dynamic condition, including user input. A graphic control that always looks the same should probably not be a component at all. If you want a static image, you can import the image instead of using a control.

In general, the appearance of a graphic control depends on some combination of its properties. The gauge control, for example, has properties that determine its shape and orientation and whether it shows its progress numerically as well as graphically. Similarly, the shape control has a property that determines what kind of shape it should draw.

To give your control a property that determines the shape it draws, add a property called *Shape*. This requires

1 Declaring the property type.
2 Declaring the property.
3 Writing the implementation method.

Creating properties is explained in more detail in Chapter 26, “Creating properties.”

Declaring the property type

When you declare a property of a user-defined type, you must declare the type first, before the class that includes the property. The most common sort of user-defined type for properties is enumerated.

For the shape control, you need an enumerated type with an element for each kind of shape the control can draw.

Add the following type definition above the shape control class’s declaration.

```
type
  TSampleShapeType = (sstRectangle, sstSquare, sstRoundRect, sstRoundSquare,
    sstEllipse, sstCircle);
  TSampleShape = class(TGraphicControl) { this is already there }
```

You can now use this type to declare a new property in the class.

Declaring the property

When you declare a property, you usually need to declare a private field to store the data for the property, then specify methods for reading and writing the property value. Often, you don't need to use a method to read the value, but can just point to the stored data instead.

For the shape control, you will declare a field that holds the current shape, then declare a property that reads that field and writes to it through a method call.

Add the following declarations to *TSampleShape*:

```
type
  TSampleShape = class(TGraphicControl)
  private
    FShape: TSampleShapeType;  { field to hold property value }
    procedure SetShape(Value: TSampleShapeType);
  published
    property Shape: TSampleShapeType read FShape write SetShape;
  end;
```

Now all that remains is to add the implementation of *SetShape*.

Writing the implementation method

When the **read** or **write** part of a property definition uses a method instead of directly accessing the stored property data, you need to implement the method.

Add the implementation of the *SetShape* method to the **implementation** part of the unit:

```
procedure TSampleShape.SetShape(Value: TSampleShapeType);
begin
  if FShape <> Value then                        { ignore if this isn't a change }
  begin
    FShape := Value;                             { store the new value }
    Invalidate;                                  { force a repaint with the new shape }
  end;
end;
```

Overriding the constructor and destructor

To change default property values and initialize owned classes for your component, you must override the inherited constructor and destructor. In both cases, remember always to call the inherited method in your new constructor or destructor.

Changing default property values

The default size of a graphic control is fairly small, so you can change the width and height in the constructor. Changing default property values is explained in more detail in Chapter 31, "Modifying an existing component."

In this example, the shape control sets its size to a square 65 pixels on each side.

Add the overridden constructor to the declaration of the component class:

```
type
  TSampleShape = class(TGraphicControl)
  public                                             { constructors are always public }
    constructor Create(AOwner: TComponent); override       { remember override directive }
  end;
```

1 Redeclare the *Height* and *Width* properties with their new default values:

```
type
  TSampleShape = class(TGraphicControl)
  ⋮
  published
    property Height default 65;
    property Width default 65;
  end;
```

2 Write the new constructor in the **implementation** part of the unit:

```
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);  { always call the inherited constructor }
  Width := 65;
  Height := 65;
end;
```

Publishing the pen and brush

By default, a canvas has a thin black pen and a solid white brush. To let developers change the pen and brush, you must provide classes for them to manipulate at design time, then copy the classes into the canvas during painting. Classes such as an auxiliary pen or brush are called *owned classes* because the component owns them and is responsible for creating and destroying them.

Managing owned classes requires

1 Declaring the class fields.

2 Declaring the access properties.

3 Initializing owned classes.

4 Setting owned classes' properties.

Declaring the class fields

Each class a component owns must have a class field declared for it in the component. The class field ensures that the component always has a pointer to the owned object so that it can destroy the class before destroying itself. In general, a component initializes owned objects in its constructor and destroys them in its destructor.

Fields for owned objects are nearly always declared as private. If applications (or other components) need access to the owned objects, you can declare **published** or **public** properties for this purpose.

Add fields for a pen and brush to the shape control:

```
type
  TSampleShape = class(TGraphicControl)
  private           { fields are nearly always private }
    FPen: TPen;      { a field for the pen object }
    FBrush: TBrush;  { a field for the brush object }
    ⋮
  end;
```

Declaring the access properties

You can provide access to the owned objects of a component by declaring properties of the type of the objects. That gives developers a way to access the objects at design time or runtime. Usually, the read part of the property just references the class field, but the write part calls a method that enables the component to react to changes in the owned object.

To the shape control, add properties that provide access to the pen and brush fields. You will also declare methods for reacting to changes to the pen or brush.

```
type
  TSampleShape = class(TGraphicControl)
  ⋮
  private                                         { these methods should be private }
    procedure SetBrush(Value: TBrush);
    procedure SetPen(Value: TPen);
  published                                     { make these available at design time }
    property Brush: TBrush read FBrush write SetBrush;
    property Pen: TPen read FPen write SetPen;
  end;
```

Then, write the *SetBrush* and *SetPen* methods in the implementation part of the unit:

```
procedure TSampleShape.SetBrush(Value: TBrush);
begin
  FBrush.Assign(Value);                        { replace existing brush with parameter }
end;

procedure TSampleShape.SetPen(Value: TPen);
begin
  FPen.Assign(Value);                            { replace existing pen with parameter }
end;
```

To directly assign the contents of *Value* to *FBrush*...

```
  FBrush := Value;
```

...would overwrite the internal pointer for *FBrush*, lose memory, and create a number of ownership problems.

Initializing owned classes

If you add classes to your component, the component's constructor must initialize them so that the user can interact with the objects at runtime. Similarly, the component's destructor must also destroy the owned objects before destroying the component itself.

Because you have added a pen and a brush to the shape control, you need to initialize them in the shape control's constructor and destroy them in the control's destructor:

1 Construct the pen and brush in the shape control constructor:

```
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                     { always call the inherited constructor }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                             { construct the pen }
  FBrush := TBrush.Create;                                       { construct the brush }
end;
```

2 Add the overridden destructor to the declaration of the component class:

```
type
  TSampleShape = class(TGraphicControl)
  public                                                 { destructors are always public}
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;                         { remember override directive }
  end;
```

3 Write the new destructor in the **implementation** part of the unit:

```
destructor TSampleShape.Destroy;
begin
  FPen.Free;                                                  { destroy the pen object }
  FBrush.Free;                                              { destroy the brush object }
  inherited Destroy;                        { always call the inherited destructor, too }
end;
```

Setting owned classes' properties

As the final step in handling the pen and brush classes, you need to make sure that changes in the pen and brush cause the shape control to repaint itself. Both pen and brush classes have *OnChange* events, so you can create a method in the shape control and point both *OnChange* events to it.

Add the following method to the shape control, and update the component's constructor to set the pen and brush events to the new method:

```
type
  TSampleShape = class(TGraphicControl)
  published
    procedure StyleChanged(Sender: TObject);
  end;
⋮
implementation
⋮
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                     { always call the inherited constructor }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                             { construct the pen }
  FPen.OnChange := StyleChanged;                       { assign method to OnChange event }
```

```
    FBrush := TBrush.Create;                                  { construct the brush }
    FBrush.OnChange := StyleChanged;                { assign method to OnChange event }
  end;

  procedure TSampleShape.StyleChanged(Sender: TObject);
  begin
    Invalidate;                                   { erase and repaint the component }
  end;
```

With these changes, the component redraws to reflect changes to either the pen or the brush.

Drawing the component image

The essential element of a graphic control is the way it paints its image on the screen. The abstract type *TGraphicControl* defines a method called *Paint* that you override to paint the image you want on your control.

The *Paint* method for the shape control needs to do several things:

- Use the pen and brush selected by the user.
- Use the selected shape.
- Adjust coordinates so that squares and circles use the same width and height.

Overriding the *Paint* method requires two steps:

1. Add *Paint* to the component's declaration.
2. Write the *Paint* method in the **implementation** part of the unit.

For the shape control, add the following declaration to the class declaration:

```
type
  TSampleShape = class(TGraphicControl)
  ⋮
  protected
    procedure Paint; override;
  ⋮
  end;
```

Then write the method in the **implementation** part of the unit:

```
procedure TSampleShape.Paint;
begin
  with Canvas do
  begin
    Pen := FPen;                                        { copy the component's pen }
    Brush := FBrush;                                  { copy the component's brush }
    case FShape of
      sstRectangle, sstSquare:
        Rectangle(0, 0, Width, Height);              { draw rectangles and squares }
      sstRoundRect, sstRoundSquare:
        RoundRect(0, 0, Width, Height, Width div 4, Height div 4); { draw rounded shapes }
      sstCircle, sstEllipse:
        Ellipse(0, 0, Width, Height);                         { draw round shapes }
    end;
  end;
end;
```

Paint is called whenever the control needs to update its image. Controls are painted when they first appear or when a window in front of them goes away. In addition, you can force repainting by calling *Invalidate*, as the *StyleChanged* method does.

Refining the shape drawing

The standard shape control does one more thing that your sample shape control does not yet do: it handles squares and circles as well as rectangles and ellipses. To do that, you need to write code that finds the shortest side and centers the image.

Here is a refined *Paint* method that adjusts for squares and ellipses:

```
procedure TSampleShape.Paint;
var
  X, Y, W, H, S: Integer;
begin
  with Canvas do
  begin
    Pen := FPen;                                           { copy the component's pen }
    Brush := FBrush;                                     { copy the component's brush }
    W := Width;                                             { use the component width }
    H := Height;                                           { use the component height }
    if W < H then S := W else S := H;             { save smallest for circles/squares }

    case FShape of                                { adjust height, width and position }
      sstRectangle, sstRoundRect, sstEllipse:
        begin
          X := 0;                                { origin is top-left for these shapes }
          Y := 0;
        end;
      sstSquare, sstRoundSquare, sstCircle:
        begin
          X := (W - S) div 2;                           { center these horizontally... }
          Y := (H - S) div 2;                                        { ...and vertically }
          W := S;                                { use shortest dimension for width... }
          H := S;                                                     { ...and for height }
        end;
    end;

    case FShape of
      sstRectangle, sstSquare:
        Rectangle(X, Y, X + W, Y + H);                  { draw rectangles and squares }
      sstRoundRect, sstRoundSquare:
        RoundRect(X, Y, X + W, Y + H, S div 4, S div 4);          { draw rounded shapes }
      sstCircle, sstEllipse:
        Ellipse(X, Y, X + W, Y + H);                                 { draw round shapes }
    end;
  end;
end;
```

Chapter

33 Customizing a grid

Kylix provides abstract components you can use as the basis for customized components. The most important of these are grids and list boxes. In this chapter, you will see how to create a small one-month calendar from the basic grid component, *TCustomGrid*.

Creating the calendar involves these tasks:

- Creating and registering the component
- Publishing inherited properties
- Changing initial values
- Resizing the cells
- Filling in the cells
- Navigating months and years
- Navigating days

The resulting component is similar to the *TCalendar* component on the Samples page of the Component palette.

Creating and registering the component

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in “Creating a new component” on page 24-8.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component’s unit *CalSamp*.
- Derive a new component type called *TSampleCalendar*, descended from *TCustomGrid*.
- Register *TSampleCalendar* on the Samples page of the Component palette.

The resulting unit should look like this:

```
unit CalSamp;

interface

uses
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs, QGrids;

type
  TSampleCalendar = class(TCustomGrid)
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TSampleCalendar]);
end;

end.
```

If you install the calendar component now, you will find that it appears on the Samples page. The only properties available are the most basic control properties. The next step is to make some of the more specialized properties available to users of the calendar.

Note While you can install the sample calendar component you have just compiled, do not try to place it on a form yet. The *TCustomGrid* component has an abstract *DrawCell* method that must be redeclared before instance objects can be created. Overriding the *DrawCell* method is described in "Filling in the cells" below.

Publishing inherited properties

The abstract grid component, *TCustomGrid*, provides a large number of **protected** properties. You can choose which of those properties you want to make available to users of the calendar control.

To make inherited protected properties available to users of your components, redeclare the properties in the **published** part of your component's declaration.

For the calendar control, publish the following properties and events, as shown here:

```
type
  TSampleCalendar = class(TCustomGrid)
  published
    property Align;  { publish properties }
    property BorderStyle;
    property Color;
    property Font;
    property GridLineWidth;
    property ParentColor;
    property ParentFont;
    property OnClick;  { publish events }
    property OnDblClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
  end;
```

There are a number of other properties you could also publish, but which do not apply to a calendar, such as the *Options* property that would enable the user to choose which grid lines to draw.

If you install the modified calendar component to the Component palette and use it in an application, you will find many more properties and events available in the calendar, all fully functional. You can now start adding new capabilities of your own design.

Changing initial values

A calendar is essentially a grid with a fixed number of rows and columns, although not all the rows always contain dates. For this reason, you have not published the grid properties *ColCount* and *RowCount*, because it is highly unlikely that users of the calendar will want to display anything other than seven days per week. You still must set the initial values of those properties so that the week always has seven days, however.

To change the initial values of the component's properties, override the constructor to set the desired values. The constructor must be virtual.

Remember that you need to add the constructor to the **public** part of the component's object declaration, then write the new constructor in the **implementation** part of the component's unit. The first statement in the new constructor should always be a call to the inherited constructor.

```
type
  TSampleCalendar = class(TCustomGrid
  public
    constructor Create(AOwner: TComponent); override;
  ⋮
  end;
⋮
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                              { call inherited constructor }
  ColCount := 7;                                           { always seven days/week }
  RowCount := 7;                                   { always six weeks plus the headings }
  FixedCols := 0;                                                   { no row labels }
  FixedRows := 1;                                           { one row for day names }
  ScrollBars := ssNone;                                         { no need to scroll }
  Options := Options - [goRangeSelect] + [goDrawFocusSelected];  {disable range selection}
end;
```

The calendar now has seven columns and seven rows, with the top row fixed, or nonscrolling.

Resizing the cells

When a user or application changes the size of a window or control, it is automatically notified by a call to the protected *BoundsChanged* method . Your component can respond to this notification by altering the size of the cells so they all fit inside the boundaries of the control.

In this case, the calendar control needs to override *BoundsChanged* so that it calculates the proper cell size to allow all cells to be visible in the new size:

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure BoundsChanged; override;
  ⋮
  end;
⋮
procedure TSampleCalendar.BoundsChanged;
var
  GridLines: Integer;                                        { temporary local variable }
begin
  GridLines := 6 * GridLineWidth;                  { calculate combined size of all lines }
  DefaultColWidth := (Width - GridLines) div 7;     { set new default cell width }
  DefaultRowHeight := (Height - GridLines) div 7;             { and cell height }
  inherited; {now call the inherited method }
end;
```

Now when the calendar is resized, it displays all the cells in the largest size that will fit in the control.

Filling in the cells

A grid control fills in its contents cell-by-cell. In the case of the calendar, that means calculating which date, if any, belongs in each cell. The default drawing for grid cells takes place in a virtual method called *DrawCell*.

To fill in the contents of grid cells, override the *DrawCell* method.

The easiest part to fill in is the heading cells in the fixed row. The runtime library contains an array with short day names, so for the calendar, use the appropriate one for each column:

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
override;
  end;
⋮
procedure TSampleCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect;
  AState: TGridDrawState);
begin
  if ARow = 0 then
    Canvas.TextOut(ARect.Left, ARect.Top, ShortDayNames[ACol + 1]);    { use RTL strings }
end;
```

Tracking the date

For the calendar control to be useful, users and applications must have a mechanism for setting the day, month, and year. CLX stores dates and times in variables of type *TDateTime*. *TDateTime* is an encoded numeric representation of the date and time, which is useful for programmatic manipulation, but not convenient for human use.

You can therefore store the date in encoded form, providing runtime access to that value, but also provide *Day*, *Month*, and *Year* properties that users of the calendar component can set at design time.

Tracking the date in the calendar consists of the processes:

- Storing the internal date
- Accessing the day, month, and year
- Generating the day numbers
- Selecting the current day

Storing the internal date

To store the date for the calendar, you need a private field to hold the date and a runtime-only property that provides access to that date.

Adding the internal date to the calendar requires three steps:

1 Declare a private field to hold the date:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FDate: TDateTime;
  ⋮
```

2 Initialize the date field in the constructor:

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);         { this is already here }
  ⋮                                 { other initializations here }
  FDate := Date;                    { get current date from RTL }
end;
```

3 Declare a runtime property to allow access to the encoded date.

You'll need a method for setting the date, because setting the date requires updating the onscreen image of the control:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    procedure SetCalendarDate(Value: TDateTime);
  public
    property CalendarDate: TDateTime read FDate write SetCalendarDate;
  ⋮
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;                { set new date value }
  Refresh;                       { update the onscreen image }
end;
```

Accessing the day, month, and year

An encoded numeric date is fine for applications, but humans prefer to work with days, months, and years. You can provide alternate access to those elements of the stored, encoded date by creating properties.

Because each element of the date (day, month, and year) is an integer, and because setting each requires encoding the date when set, you can avoid duplicating the code each time by sharing the implementation methods for all three properties. That is, you can write two methods, one to read an element and one to write one, and use those methods to get and set all three properties.

To provide design-time access to the day, month, and year, you do the following:

1 Declare the three properties, assigning each a unique **index** number:

```
type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  ⋮
```

2 Declare and write the implementation methods, setting different elements for each index value:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    function GetDateElement(Index: Integer): Integer;         { note the Index parameter }
    procedure SetDateElement(Index: Integer; Value: Integer);
  ⋮
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);             { break encoded date into elements }
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
  AYear, AMonth, ADay: Word;
begin
  if Value > 0 then                                       { all elements must be positive }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);                    { get current date elements }
    case Index of                                    { set new element depending on Index }
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);              { encode the modified date }
    Refresh;                                             { update the visible calendar }
  end;
end;
```

Now you can set the calendar's day, month, and year at design time using the Object Inspector or at runtime using code. Of course, you have not yet added the code to paint the dates into the cells, but now you have the needed data.

Generating the day numbers

Putting numbers into the calendar involves several considerations. The number of days in the month depends on which month it is, and whether the given year is a leap year. In addition, months start on different days of the week, dependent on the month and year. Use the *IsLeapYear* function to determine whether the year is a leap year. Use the *MonthDays* array in the SysUtils unit to get the number of days in the month.

Once you have the information on leap years and days per month, you can calculate where in the grid the individual dates go. The calculation is based on the day of the week the month starts on.

Because you will need the month-offset number for each cell you fill in, the best practice is to calculate it once when you change the month or year, then refer to it each time. You can store the value in a class field, then update that field each time the date changes.

To fill in the days in the proper cells, you do the following:

1 Add a month-offset field to the object and a method that updates the field value:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FMonthOffset: Integer;                                 { storage for the offset }
  ⋮
  protected
    procedure UpdateCalendar; virtual;                   { property for offset access }
  end;
⋮
procedure TSampleCalendar.UpdateCalendar;
var
  AYear, AMonth, ADay: Word;
  FirstDate: TDateTime;                          { date of the first day of the month }
begin
  if FDate <> 0 then                         { only calculate offset if date is valid }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);                    { get elements of date }
    FirstDate := EncodeDate(AYear, AMonth, 1);                    { date of the first }
    FMonthOffset := 2 - DayOfWeek(FirstDate);        { generate the offset into the grid }
  end;
  Refresh;                                                { always repaint the control }
end;
```

2 Add statements to the constructor and the *SetCalendarDate* and *SetDateElement* methods that call the new update method whenever the date changes:

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                                { this is already here }
  ⋮                                                     { other initializations here }
  UpdateCalendar;                                             { set proper offset }
end;

procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;                                         { this was already here }
  UpdateCalendar;                                   { this previously called Refresh }
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
  ⋮
    FDate := EncodeDate(AYear, AMonth, ADay);              { encode the modified date }
    UpdateCalendar;                                 { this previously called Refresh }
  end;
end;
```

3 Add a method to the calendar that returns the day number when passed the row and column coordinates of a cell:

```
function TSampleCalendar.DayNum(ACol, ARow: Integer): Integer;
begin
  Result := FMonthOffset + ACol + (ARow - 1) * 7;         { calculate day for this cell }
  if (Result < 1) or (Result > MonthDays[IsLeapYear(Year), Month]) then
    Result := -1;                                            { return -1 if invalid }
end;
```

Remember to add the declaration of *DayNum* to the component's type declaration.

4 Now that you can calculate where the dates go, you can update *DrawCell* to fill in the dates:

```
procedure TCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
var
  TheText: string;
  TempDay: Integer;
begin
  if ARow = 0 then                                      { if this is the header row ...}
    TheText := ShortDayNames[ACol + 1]                         { just use the day name }
  else begin
    TheText := '';                                        { blank cell is the default }
    TempDay := DayNum(ACol, ARow);                         { get number for this cell }
    if TempDay <> -1 then TheText := IntToStr(TempDay);      { use the number if valid }
  end;
  with ARect, Canvas do
    TextRect(ARect, Left + (Right - Left - TextWidth(TheText)) div 2,
      Top + (Bottom - Top - TextHeight(TheText)) div 2, TheText);
end;
```

Now if you reinstall the calendar component and place one on a form, you will see the proper information for the current month.

Selecting the current day

Now that you have numbers in the calendar cells, it makes sense to move the selection highlighting to the cell containing the current day. By default, the selection starts on the top left cell, so you need to set the *Row* and *Column* properties both when constructing the calendar initially and when the date changes.

To set the selection on the current day, change the *UpdateCalendar* method to set *Row* and *Column* before calling *Refresh*:

```
procedure TSampleCalendar.UpdateCalendar;
begin
  if FDate <> 0 then
  begin
    ⋮ { existing statements to set FMonthOffset }
    Row := (ADay - FMonthOffset) div 7 + 1;
    Col := (ADay - FMonthOffset) mod 7;
  end;
  Refresh; { this is already here }
end;
```

Note that you are now reusing the *ADay* variable previously set by decoding the date.

Navigating months and years

Properties are useful for manipulating components, especially at design time. But sometimes there are types of manipulations that are so common or natural, often involving more than one property, that it makes sense to provide methods to handle them. One example of such a natural manipulation is a “next month” feature for a calendar. Handling the wrapping around of months and incrementing of years is simple, but very convenient for the developer using the component.

The only drawback to encapsulating common manipulations into methods is that methods are only available at runtime. However, such manipulations are generally only cumbersome when performed repeatedly, and that is fairly rare at design time.

For the calendar, add the following four methods for next and previous month and year. Each of these methods uses the *IncMonth* function in a slightly different manner to increment or decrement *CalendarDate*, by increments of a month or a year. After incrementing or decrementing *CalendarDate*, decode the date value to fill the Year, Month, and Day properties with corresponding new values.

```
procedure TCalendar.NextMonth;
begin
  DecodeDate(IncMonth(CalendarDate, 1), Year, Month, Day);
end;

procedure TCalendar.PrevMonth;
begin
  DecodeDate(IncMonth(CalendarDate, -1), Year, Month, Day);
end;

procedure TCalendar.NextYear;
```

```
begin
  DecodeDate(IncMonth(CalendarDate, 12), Year, Month, Day);
end;

procedure TCalendar.PrevYear;
begin
  DecodeDate(IncMonth(CalendarDate, -12), Year, Month, Day);
end;
```

Be sure to add the declarations of the new methods to the class declaration.

Now when you create an application that uses the calendar component, you can easily implement browsing through months or years.

Navigating days

Within a given month, there are two obvious ways to navigate among the days. The first is to use the arrow keys, and the other is to respond to clicks of the mouse. The standard grid component handles both as if they were clicks. That is, an arrow movement is treated like a click on an adjacent cell.

The process of navigating days consists of

- Moving the selection
- Providing an OnChange event
- Excluding blank cells

Moving the selection

The inherited behavior of a grid handles moving the selection in response to either arrow keys or clicks, but if you want to change the selected day, you need to modify that default behavior.

To handle movements within the calendar, override the *Click* method of the grid.

When you override a method such as *Click* that is tied in with user interactions, you will nearly always include a call to the inherited method, so as not to lose the standard behavior.

The following is an overridden *Click* method for the calendar grid. Be sure to add the declaration of *Click* to *TSampleCalendar*, including the **override** directive afterward.

```
procedure TSampleCalendar.Click;
var
  TempDay: Integer;
begin
  inherited Click;                          { remember to call the inherited method! }
  TempDay := DayNum(Col, Row);             { get the day number for the clicked cell }
  if TempDay <> -1 then Day := TempDay;                         { change day if valid }
end;
```

Providing an OnChange event

Now that users of the calendar can change the date within the calendar, it makes sense to allow applications to respond to those changes.

Add an *OnChange* event to *TSampleCalendar*.

1 Declare the event, a field to store the event, and a dynamic method to call the event:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FOnChange: TNotifyEvent;
  protected
    procedure Change; dynamic;
  ⋮
  published
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
  ⋮
```

2 Write the *Change* method:

```
procedure TSampleCalendar.Change;
begin
  if Assigned(FOnChange) then FOnChange(Self);
end;
```

3 Add statements calling *Change* to the end of the *SetCalendarDate* and *SetDateElement* methods:

```
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;
  UpdateCalendar;
  Change;                                         { this is the only new statement }
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
    ⋮                                      { many statements setting element values }
    FDate := EncodeDate(AYear, AMonth, ADay);
    UpdateCalendar;
    Change;                                                            { this is new }
  end;
end;
```

Applications using the calendar component can now respond to changes in the date of the component by attaching handlers to the *OnChange* event.

Excluding blank cells

As the calendar is written, the user can select a blank cell, but the date does not change. It makes sense, then, to disallow selection of the blank cells.

To control whether a given cell is selectable, override the *SelectCell* method of the grid.

SelectCell is a function that takes a column and row as parameters, and returns a Boolean value indicating whether the specified cell is selectable.

You can override *SelectCell* to return *False* if the cell does not contain a valid date:

```
function TSampleCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if DayNum(ACol, ARow) = -1 then Result := False           { -1 indicates invalid date }
  else Result := inherited SelectCell(ACol, ARow);      { otherwise, use inherited value }
end;
```

Now if the user clicks a blank cell or tries to move to one with an arrow key, the calendar leaves the current cell selected.

Chapter

34

Making a control data aware

When working with database connections, it is often convenient to have controls that are *data aware*. That is, the application can establish a link between the control and some part of a database. CLX includes data-aware labels, edit boxes, list boxes, combo boxes, lookup controls, and grids. You can also make your own controls data aware. For more information about using data-aware controls, see Chapter 15, "Using data controls".

There are several degrees of data awareness. The simplest is read-only data awareness, or *data browsing*, the ability to reflect the current state of a database. More complicated is editable data awareness, or *data editing*, where the user can edit the values in the database by manipulating the control. Note also that the degree of involvement with the database can vary, from the simplest case, a link with a single field, to more complex cases, such as multiple-record controls.

This chapter first illustrates the simplest case, making a read-only control that links to a single field in a dataset. The specific control used will be the *TSampleCalendar* calendar created in Chapter 33, "Customizing a grid".

The chapter then continues with an explanation of how to make the new data-browsing control a data-editing control.

Creating a data-browsing control

Creating a data-aware calendar control, whether it is a read-only control or one in which the user can change the underlying data in the dataset, involves the following steps:

- Creating and registering the component
- Adding the data link
- Responding to data changes

Creating and registering the component

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in “Creating a new component” on page 24-8.

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component’s unit *DBCal*.
- Derive a new component class called *TDBCalendar*, descended from *TSampleCalendar*. Chapter 33, “Customizing a grid,” shows you how to create the *TSampleCalendar* component.
- Register *TDBCalendar* on the Samples page of the Component palette.

The resulting unit should look like this:

```
unit DBCal;

interface

uses SysUtils, Types, Classes, QGraphics, QControls, QForms, QGrids, Calendar;

type
  TDBCalendar = class(TSampleCalendar)
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TDBCalendar]);
end;

end.
```

You can now proceed with making the new calendar a data browser.

Making the control read-only

Because this data calendar will be read-only with respect to the data, it makes sense to make the control itself read-only, so users will not make changes within the control and expect them to be reflected in the database.

Making the calendar read-only involves,

- Adding the ReadOnly property.
- Allowing needed updates.

Adding the ReadOnly property

By adding a *ReadOnly* property, you provide a way to make the control read-only at design time. When that property is set to *True*, you can make all cells in the control unselectable.

1 Add the property declaration and a **private** field to hold the value:

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FReadOnly: Boolean;                                   { field for internal storage }
  public
    constructor Create(AOwner: TComponent); override;     { must override to set default }
  published
    property ReadOnly: Boolean read FReadOnly write FReadOnly default True;
  end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                        { always call the inherited constructor! }
  FReadOnly := True;                                                { set the default value }
end;
```

2 Override the *SelectCell* method to disallow selection if the control is read-only. Use of *SelectCell* is explained in “Excluding blank cells” on page 33-12.

```
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if FReadOnly then Result := False                           { cannot select if read only }
  else Result := inherited SelectCell(ACol, ARow);     { otherwise, use inherited method }
end;
```

Remember to add the declaration of *SelectCell* to the type declaration of *TDBCalendar*, and append the **override** directive.

If you now add the calendar to a form, you will find that the component ignores clicks and keystrokes. It also fails to update the selection position when you change the date.

Allowing needed updates

The read-only calendar uses the *SelectCell* method for all kinds of changes, including setting the *Row* and *Col* properties. The *UpdateCalendar* method sets *Row* and *Col* every time the date changes, but because *SelectCell* disallows changes, the selection remains in place, even though the date changes.

To get around this absolute prohibition on changes, you can add an internal Boolean flag to the calendar, and permit changes when that flag is set to *True*:

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FUpdating: Boolean;                                  { private flag for internal use }
  protected
    function SelectCell(ACol, ARow: Longint): Boolean; override;
  public
    procedure UpdateCalendar; override;                 { remember the override directive }
  end;
⋮
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
```

```
    if (not FUpdating) and FReadOnly then Result := False       { allow select if updating }
    else Result := inherited SelectCell(ACol, ARow);     { otherwise, use inherited method }
  end;

  procedure TDBCalendar.UpdateCalendar;
  begin
    FUpdating := True;                                           { set flag to allow updates }
    try
      inherited UpdateCalendar;                                            { update as usual }
    finally
      FUpdating := False;                                            { always clear the flag }
    end;
  end;
```

The calendar still disallows user changes, but now correctly reflects changes made in the date by changing the date properties. Now that you have a true read-only calendar control, you are ready to add the data-browsing ability.

Adding the data link

The connection between a control and a database is handled by a class called a *data link*. The datalink class that connects a control with a single field in a database is *TFieldDataLink.* There are also data links for entire tables.

A data-aware control *owns* its datalink class. That is, the control has the responsibility for constructing and destroying the data link.

Establishing a data link as an owned class requires these three steps:

1. Declaring the class field
2. Declaring the access properties
3. Initializing the data link

Declaring the class field

A component needs a field for each of its owned classes, as explained in “Declaring the class fields” on page 32-5. In this case, the calendar needs a field of type *TFieldDataLink* for its data link.

Declare a field for the data link in the calendar:

```
  type
    TDBCalendar = class(TSampleCalendar)
    private
      FDataLink: TFieldDataLink;
    ⋮
    end;
```

Before you can compile the application, you need to add DB and DBCtrls to the unit’s **uses** clause.

Declaring the access properties

Every data-aware control has a *DataSource* property that specifies which data-source class in the application provides the data to the control. In addition, a control that accesses a single field needs a *DataField* property to specify that field in the data source.

Unlike the access properties for the owned classes, which provide access to private data members in the class, these access properties maintained by the owned class. That is, you will create properties that allow the control and its data link to share the same data source and field.

Declare the *DataSource* and *DataField* properties and their implementation methods, then write the methods as “pass-through” methods to the corresponding properties of the datalink class:

An example of declaring access properties

```
type
  TDBCalendar = class(TSampleCalendar)
  private                                           { implementation methods are private }
    ...
    function GetDataField: string;                 { returns the name of the data field }
    function GetDataSource: TDataSource;         { returns reference to the data source }
    procedure SetDataField(const Value: string);           { assigns name of data field }
    procedure SetDataSource(Value: TDataSource);             { assigns new data source }
  published                                 { make properties available at design time }
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
⋮
function TDBCalendar.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

function TDBCalendar.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TDBCalendar.SetDataField(const Value: string);
begin
  FDataLink.FieldName := Value;
end;

procedure TDBCalendar.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
end;
```

Now that you have established the links between the calendar and its data link, there is one more important step. You must construct the data link class when the calendar control is constructed, and destroy the data link before destroying the calendar.

Initializing the data link

A data-aware control needs access to its data link throughout its existence, so it must construct the datalink object as part of its own constructor, and destroy the datalink object before it is itself destroyed.

Override the *Create* and *Destroy* methods of the calendar to construct and destroy the datalink object, respectively:

```
type
  TDBCalendar = class(TSampleCalendar)
  public                                 { constructors and destructors are always public }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    ⋮
  end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                { always call the inherited constructor first
}
  FDataLink := TFieldDataLink.Create;                     { construct the datalink object }
  FDataLink.Control := self;              {let the datalink know about the calendar }
  FReadOnly := True;                                                { this is already here }
end;

destructor TDBCalendar.Destroy;
begin
  FDataLink.Free;                                { always destroy owned objects first... }
  inherited Destroy;                                  { ...then call inherited destructor
}
end;
```

Now you have a complete data link, but you have not yet told the control what data it should read from the linked field. The next section explains how to do that.

Responding to data changes

Once a control has a data link and properties to specify the data source and data field, it needs to respond to changes in the data in that field, either because of a move to a different record or because of a change made to that field.

Datalink classes all have events named *OnDataChange*. When the data source indicates a change in its data, the datalink object calls any event handler attached to its *OnDataChange* event.

To update a control in response to data changes, attach a handler to the data link's *OnDataChange* event.

In this case, you will add a method to the calendar, then designate it as the handler for the data link's *OnDataChange*.

Declare and implement the *DataChange* method, then assign it to the data link's *OnDataChange* event in the constructor. In the destructor, detach the *OnDataChange* handler before destroying the object.

```
type
  TDBCalendar = class(TSampleCalendar)
  private { this is an internal detail, so make it private }
    procedure DataChange(Sender: TObject);        { must have proper parameters for event
}
  end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);               { always call the inherited constructor first
}
  FReadOnly := True;                                          { this is already here }
  FDataLink := TFieldDataLink.Create;                    { construct the datalink object }
  FDataLink.OnDataChange := DataChange;                        { attach handler to event }
end;

destructor TDBCalendar.Destroy;
begin
  FDataLink.OnDataChange := nil;              { detach handler before destroying object }
  FDataLink.Free;                               { always destroy owned objects first... }
  inherited Destroy;                                { ...then call inherited destructor
}
end;

procedure TDBCalendar.DataChange(Sender: TObject);
begin
  if FDataLink.Field = nil then                    { if there is no field assigned...
}
    CalendarDate := 0                                         { ...set to invalid date }
  else CalendarDate := FDataLink.Field.AsDateTime;   { otherwise, set calendar to the date }
end;
```

You now have a data-browsing control.

Creating a data-editing control

When you create a data-editing control, you create and register the component and add the data link just as you do for a data-browsing control. You also respond to data changes in the underlying field in a similar manner, but you must handle a few more issues.

For example, you probably want your control to respond to both key and mouse events. Your control must respond when the user changes the contents of the control. When the user exits the control, you want the changes made in the control to be reflected in the dataset.

The data-editing control described here is the same calendar control described in the first part of the chapter. The control is modified so that it can edit as well as view the data in its linked field.

Modifying the existing control to make it a data-editing control involves:

- Changing the default value of FReadOnly.
- Handling mouse-down and key-down events.

- Updating the field datalink class.
- Modifying the Change method.
- Updating the dataset.

Changing the default value of FReadOnly

Because this is a data-editing control, the *ReadOnly* property should be set to *False* by default. To make the *ReadOnly* property *False*, change the value of *FReadOnly* in the constructor:

```
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  ⋮
  FReadOnly := False;  { set the default value }
  ⋮
end;
```

Handling mouse-down and key-down events

When the user of the control begins interacting with it, the control receives either mouse-down events or a key-down event. To enable a control to respond to these messages, you must write handlers that respond to these messages.

- Responding to mouse-down events
- Responding to key-down events

Responding to mouse-down events

A *MouseDown* method is a protected method for a control's *OnMouseDown* event. The control itself calls *MouseDown* in response to a notification from the operating system. When you override the inherited *MouseDown* method, you can include code that provides other responses in addition to calling the *OnMouseDown* event.

To override *MouseDown*, add the *MouseDown* method to the *TDBCalendar* class:

```
type
  TDBCalendar = class(TSampleCalendar);
   ⋮
  protected
    procedure MouseDown(Button: TButton, Shift: TShiftState, X: Integer, Y: Integer);
      override;
   ⋮
  end;

procedure TDBCalendar.MouseDown(Button: TButton; Shift: TShiftState; X, Y: Integer);
var
  MyMouseDown: TMouseEvent;
begin
  if not ReadOnly and FDataLink.Edit then
    inherited MouseDown(Button, Shift, X, Y)
  else
  begin
    MyMouseDown := OnMouseDown;
    if Assigned(MyMouseDown then MyMouseDown(Self, Button, Shift, X, Y);
  end;
end;
```

When *MouseDown* responds to a mouse-down message, the inherited *MouseDown* method is called only if the control's *ReadOnly* property is *False* and the datalink object is in edit mode, which means the field can be edited. If the field cannot be edited, the code the programmer put in the *OnMouseDown* event handler, if one exists, is executed.

Responding to key-down events

A *KeyDown* method is a protected method for a control's *OnKeyDown* event. The control itself calls *KeyDown* in response to a notification from the operating system. When overriding the inherited *KeyDown* method, you can include code that provides other responses in addition to calling the *OnKeyDown* event.

To override *KeyDown*, follow these steps:

1 Add a *KeyDown* method to the *TDBCalendar* class:

```
type
  TDBCalendar = class(TSampleCalendar);
   ⋮
  protected
    procedure KeyDown(var Key: Word; Shift: TShiftState; X: Integer; Y: Integer);
      override;
   ⋮
  end;
```

2 Implement the *KeyDown* method:

```
procedure KeyDown(var Key: Word; Shift: TShiftState);
var
  MyKeyDown: TKeyEvent;
begin
  if not ReadOnly and (Key in [Key_Up, Key_Down, Key_Left, Key_Right, Key_End,
    Key_Home, Key_Prior, Key_Next]) and FDataLink.Edit then
    inherited KeyDown(Key, Shift)
  else
  begin
    MyKeyDown := OnKeyDown;
    if Assigned(MyKeyDown) then MyKeyDown(Self, Key, Shift);
  end;
end;
```

When *KeyDown* responds to a mouse-down event, the inherited *KeyDown* method is called only if the control's *ReadOnly* property is *False*, the key pressed is one of the cursor control keys, and the datalink object is in edit mode, which means the field can be edited. If the field cannot be edited or some other key is pressed, the code the programmer put in the *OnKeyDown* event handler, if one exists, is executed.

Updating the field datalink class

There are two types of data changes:

- A change in a field value that must be reflected in the data-aware control.
- A change in the data-aware control that must be reflected in the field value.

The *TDBCalendar* component already has a *DataChange* method that handles a change in the field's value in the dataset by assigning that value to the *CalendarDate* property. The *DataChange* method is the handler for the *OnDataChange* event. So the calendar component can handle the first type of data change.

Similarly, the field datalink class also has an *OnUpdateData* event that occurs as the user of the control modifies the contents of the data-aware control. The calendar control has a *UpdateData* method that becomes the event handler for the *OnUpdateData* event. *UpdateData* assigns the changed value in the data-aware control to the field data link.

1 To reflect a change made to the value in the calendar in the field value, add an *UpdateData* method to the private section of the calendar component:

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure UpdateData(Sender: TObject);
   ⋮
  end;
```

2 Implement the *UpdateData* method:

```
procedure UpdateData(Sender: TObject);
begin
  FDataLink.Field.AsDateTime := CalendarDate;       { set field link to calendar date }
end;
```

3 Within the constructor for *TDBCalendar*, assign the *UpdateData* method to the *OnUpdateData* event:

```
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FReadOnly := True;
  FDataLink := TFieldDataLink.Create;
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData := UpdateData;
end;
```

Modifying the Change method

The *Change* method of the *TDBCalendar* is called whenever a new date value is set. *Change* calls the *OnChange* event handler, if one exists. The component user can write code in the *OnChange* event handler to respond to changes in the date.

When the calendar date changes, the underlying dataset should be notified that a change has occurred. You can do that by overriding the *Change* method and adding one more line of code. These are the steps to follow:

1 Add a new *Change* method to the *TDBCalendar* component:

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure Change; override;
   ⋮
  end;
```

2 Write the *Change* method, calling the *Modified* method that informs the dataset the data has changed, then call the inherited *Change* method:

```
procedure TDBCalendar.Change;
begin
  FDataLink.Modified;                  { call the Modified method }
  inherited Change;                    { call the inherited Change method }
end;
```

Updating the dataset

So far, a change within the data-aware control has changed values in the field datalink class. The final step in creating a data-editing control is to update the dataset with the new value. This should happen after the person changing the value in the data-aware control exits the control by clicking outside the control or pressing the *Tab* key.

TWidgetControl has a protected *DoExit* method that is called when input focus shifts away from the control. This method calls the event handler for the *OnExit* event. You can override this method to update the record in the dataset before generating the *OnExit* event handler.

To update the dataset when the user exits the control, follow these steps:

1 Add an override for the *DoExit* method to the *TDBCalendar* component:

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure DoExit; override;
   ⋮
  end;
```

2 Implement the *DoExit* method so it looks something like this:

```
procedure TDBCalendar.CMExit(var Message: TWMNoParams);
begin
  try
    FDataLink.UpdateRecord;                          { tell data link to update database }
  except
    on Exception do SetFocus;                      { if it failed, don't let focus leave }
 end;
  inherited;                           { let the inherited method generate an OnExit event }
end;
```

Index