

Integrated development environment

[Topic groups](#)

When you start Delphi, 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.

Delphi's development environment includes a visual form designer, Object Inspector, Object TreeView, Component palette, Project Manager, source code editor, and debugger among other tools. Some tools may not be included in all versions of the product. 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.

Designing applications

[Topic groups](#)

[See also](#)

Delphi includes all the tools necessary to start designing applications:

- A blank window, known as a *form*, on which to design the UI for your application.
- Extensive class libraries with 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 image editor on the toolbar and an integrated debugger on menus to support application development in the IDE.
- Command-line tools including compilers, linkers, and other utilities.

You can use Delphi to design any kind of 32-bit application—from general-purpose utilities to sophisticated data access programs or distributed applications. Delphi's database tools and data-aware components let you quickly develop powerful desktop database and client/server applications. Using Delphi'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.

[Creating applications](#) introduces Delphi's support for different types of applications.

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 Delphi 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.

You can also use Delphi for cross platform development on both Linux and Windows by using tCLX. CLX contains a set of classes that, if used instead of those in the VCL, allow your program to port between Windows and Linux.

Using the component libraries

[Topic groups](#)

VCL and CLX are class libraries made up of objects, some of which are also components or controls, that you use when developing applications. Both libraries look very similar and contain many of the same objects. Some objects in the VCL implement features that are available on Windows only such as objects that appear on the ADO, BDE, QReport, COM+, Web Services, and Servers tabs on the Component palette. Virtually all CLX objects are available on both Windows and Linux.

VCL and 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 VCL and CLX objects descend from the ancestor object *TObject* including objects that you develop in Object Pascal.

A subset of objects are components. 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 VCL or CLX components descend from the *TComponent* object.

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

- Encapsulate a 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.

Controls are a special kind of component that is visible to users at runtime. Controls are a subset of components. Controls are visual components that you can see 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 all controls have in common are inherited from *TControl*.

Refer to Chapter 10, “Using CLX for cross-platform development” [Developing cross-platform applications](#) for details about cross-platform programming and the differences between the Windows and Linux environments. CLX components.

If you are using Kylix while developing cross-platform applications, Kylix also includes a *Developer’s Guide* that is tailored for the Linux environment. You can refer to the manual both in the Kylix online Help or the printed manual provided with the Kylix product.

Properties, methods, and events

[Topic groups](#)

[See also](#)

Both the VCL and CLX form hierarchies of objects that are tied to the Delphi IDE, where you can develop applications quickly. The objects in both component libraries are based on properties, methods, and events. Each object includes data members (properties), functions that operate on the data (methods), and a way to interact with users of the class (events). The VCL is written in Object Pascal, whereas CLX is based on Qt, a C++ class library.

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* is 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, Delphi 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.

Types of events

[Topic groups](#)

[See also](#)

The kinds of events that can occur can be divided into two main categories:

- User events
- System events

Regardless of how the event was called, Delphi looks to see if you have assigned any code to handle that event. If you have, then that code is executed; otherwise, nothing is done.

User events

User events are actions that are initiated by the user. Examples of user events are *OnClick* (the user clicked the mouse), *OnKeyPress* (the user pressed a key on the keyboard), and *OnDbClick* (the user double-clicked a mouse button). These events are always tied to a user's actions.

System events

System events are events that the operating system fires for you. For example, the *OnTimer* event (the Timer 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 window needs to be redrawn), and so on. Usually, system events are not directly initiated by a user action.

- DevGuide: [Building applications with Delphi](#)

Object Pascal and the class libraries

[Topic groups](#)

[See also](#)

Object Pascal, a set of object-oriented extensions to standard Pascal, is the language of Delphi. Using Delphi's Component palette and Object Inspector, you can place VCL or CLX components on forms and manipulate their properties without writing code.

All objects descend from *TObject*, an abstract class whose methods encapsulate fundamental behavior like construction, destruction, and message handling. *TObject* is the immediate ancestor of many simple classes.

Components in the VCL or CLX descend from the abstract class *TComponent*. Components are objects that you can manipulate on forms at design time. Visual components—that is, components like *TForm* and *TSpeedButton* that appear on the screen at runtime—are called *controls*, and they descend from *TControl*.

In addition to the visual components, the component libraries contain many nonvisual objects. The IDE allows you to add many nonvisual components 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 (which 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.

When you write classes of your own in Object Pascal, they should descend from *TObject* in the class library that you plan to use. Use VCL if you're writing a Windows application or CLX if writing a cross-platform application. By deriving new classes from the appropriate base class (or one of its descendants), you provide your classes with essential functionality and ensure that they work with the other classes in the class library.

- DevGuide: [Building applications with Delphi](#)

Using the object model

[Topic groups](#)

[See also](#)

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

If you want to create new components and put them on the Component palette, see [Overview of component creation](#).

The following topics discuss how to use objects in your applications:

- [What is an object?](#)
- [Inheriting data and code from an object](#)
- [Scope and qualifiers](#)
- [Using object variables](#)
- [Creating, instantiating, and destroying objects](#)

- DevGuide: [Building applications with Delphi](#)

What is an object?

[Topic groups](#)

[See also](#)

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* or *structures* in C. Records 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 VCL and 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.

■ DevGuide: [Building applications with Delphi](#)

Examining a Delphi object

[Topic groups](#)

[See also](#)

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

```
unit Unit1;
interface
uses Windows, Classes, Graphics, Forms, Controls, Dialogs;
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.

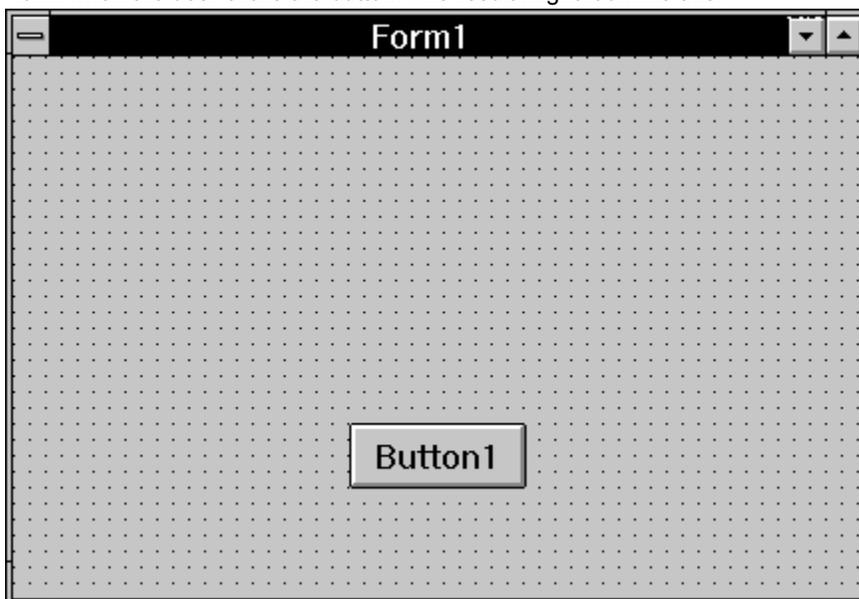
This variable declaration declares a variable named *Form1* of the new type *TForm1*.

```
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; you might want to do this, for example, to create multiple child windows in a Multiple Document Interface (MDI) application. 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 Delphi 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:



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 Windows, Classes, Graphics, Forms, Controls;
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 *.DFM}
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 Delphi 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.

■ DevGuide: [Building applications with Delphi](#)

Changing the name of a component

[Topic groups](#)

[See also](#)

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 .dfm or .xfm file (which you usually don't edit manually) and in the Object Pascal source code that Delphi generates:

```
unit Unit1;
interface
uses Windows, Classes, Graphics, Forms, Controls;
type
  TColorBox = class(TForm) { Changed from TForm1 to TColorBox }
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  ColorBox: TColorBox; { Changed from Form1 to ColorBox }
implementation
{$R *.DFM}
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;
```

- DevGuide: [Building applications with Delphi](#)

Inheriting data and code from an object

[Topic groups](#)

[See also](#)

The *TForm1* object in [Examining a Delphi object](#) 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 the 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, Delphi 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.

■ DevGuide: [Building applications with Delphi](#)

Scope and qualifiers

[Topic groups](#)

[See also](#)

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.

Delphi 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, see [Blocks and scope](#). For more information about the uses clause, see [Unit references and the uses clause](#). For more information about hiding and overriding inherited members, see [Classes and objects](#).

- DevGuide: [Building applications with Delphi](#)

Private, protected, public, and published declarations

[Topic groups](#)

[See also](#)

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 [Visibility of class members](#).

■ DevGuide: [Building applications with Delphi](#)

Using object variables

[Topic groups](#)

[See also](#)

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 *TDataForm* type declaration (VCL only) and a variable declaration section declaring two variables, *AForm* and *DataForm*:

```
type
  TDataForm = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    DataGrid1: TDataGrid;
    Database1: TDatabase;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  AForm: TForm;
  DataForm: TDataForm;
```

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

```
AForm := DataForm;
```

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;
```

■ DevGuide: [Building applications with Delphi](#)

Creating, instantiating, and destroying objects

[Topic groups](#)

[See also](#)

Many of the objects you use in Delphi, 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 Delphi. 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.

- DevGuide: [Building applications with Delphi](#)

Components and ownership

[Topic groups](#)

[See also](#)

Delphi 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*.

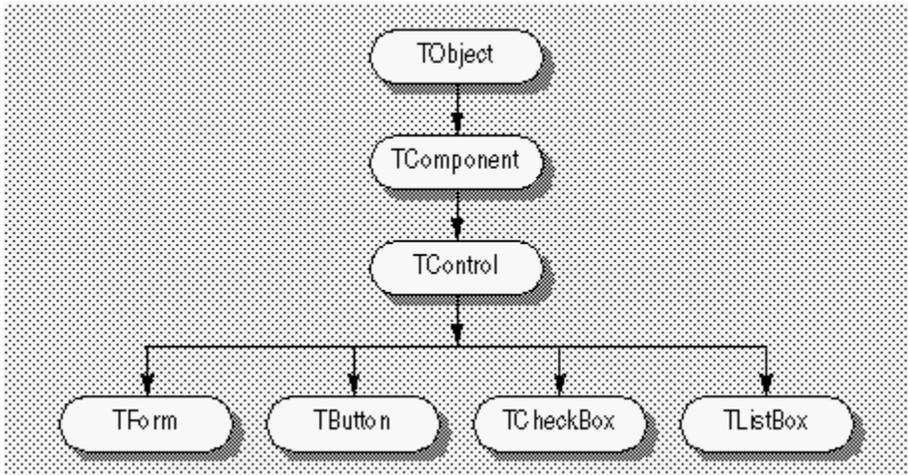
■ DevGuide: [Building applications with Delphi](#)

Objects, components, and controls

[Topic groups](#)

[See also](#)

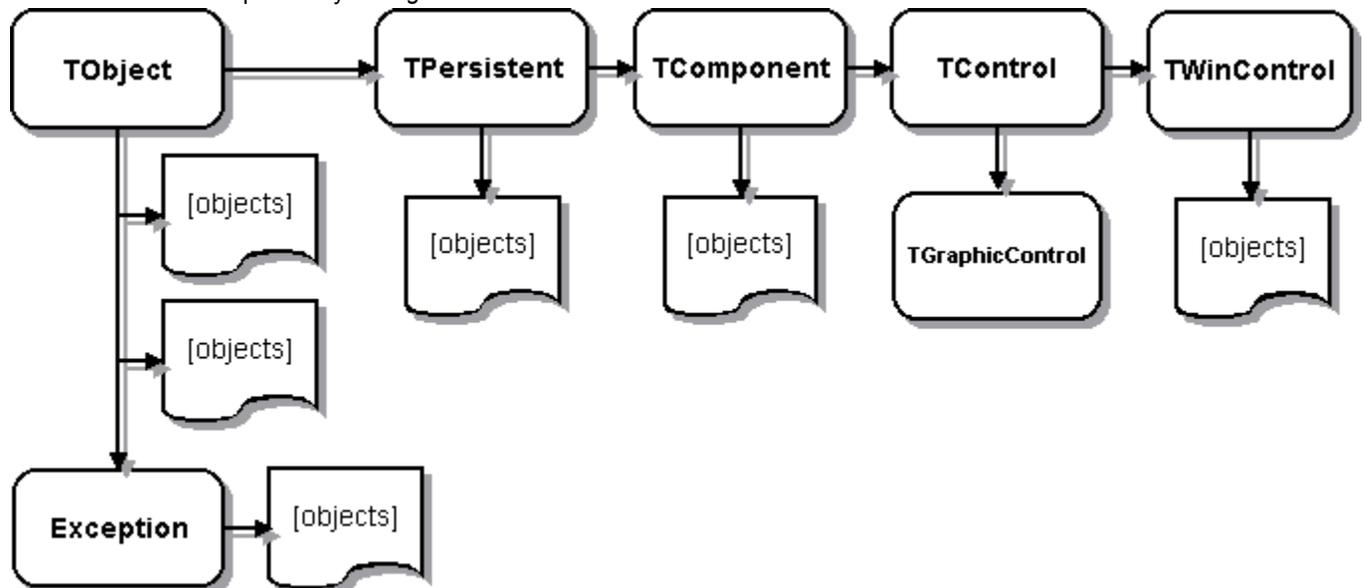
The following diagram is a greatly simplified view of the inheritance hierarchy that illustrates the relationship between objects, components, and controls.



Objects, components, and controls

Every object inherits from *TObject*, and many objects inherit from *TComponent*. Controls, which inherit from *TControl*, have the ability to display themselves at runtime. A control like *TCheckBox* inherits all the functionality of *TObject*, *TComponent*, and *TControl*, and adds specialized capabilities of its own.

The following diagram is an overview of the Visual Component Library (VCL) that shows the major branches of the inheritance tree. The Borland Component Library for Cross-Platform (CLX) look very much the same at this level but *TWinControl* is replaced by *TWidgetControl*.



Simplified object hierarchy

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

Class	Description
<i>TObject</i>	Signifies the base class and ultimate ancestor of everything in the VCL or CLX. <i>TObject</i> encapsulates the fundamental behavior common to all VCL/CLX objects by introducing methods that perform basic functions such as creating, maintaining, and destroying an instance of an object.
<i>Exception</i>	Specifies the base class of all classes that relate to exceptions. <i>Exception</i> provides a consistent interface for error conditions, and enables applications to

	handle error conditions gracefully.
<i>TPersistent</i>	Specifies the base class for all objects that implement properties. Classes under <i>TPersistent</i> deal with sending data to streams and allow for the assignment of classes.
<i>TComponent</i>	Specifies the base class for all nonvisual components such as <i>TApplication</i> . <i>TComponent</i> 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.
<i>TControl</i>	Represents the base class for all controls that are visible at runtime. <i>TControl</i> 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.
<i>TWinControl</i>	Specifies the base class of all user interface objects also called widgets. Controls under <i>TWinControl</i> are windowed controls that can capture keyboard input. (In CLX, <i>TWidgetControl</i> replaces <i>TWinControl</i> .)

For a complete overview of the VCL object hierarchy, refer to the VCL Object Hierarchy wall chart that is included with this product. For details on CLX, refer to the CLX Object Hierarchy wall chart included with the product and the Kylix documentation.

- DevGuide: [Building applications with Delphi](#)

TOject Branch

[Topic groups](#)

[See also](#)

The *TOject* branch includes all objects that descend from *TOject* but not from *TPersistent*. All VCL or CLX objects descend from *TOject*, an abstract class whose methods define fundamental behavior like construction, destruction, and message or system event handling. Much of the powerful capability of VCL and CLX objects are established by the methods that *TOject* introduces. *TOject* encapsulates the fundamental behavior common to all objects in the VCL and 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.
- Support for message-handling (VCL only).

TOject is the immediate ancestor of many simple classes. Classes that are contained within 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.

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 group in the *TOject* 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

In the VCL, you can also find wrappers for external objects like *TPrinter*, which encapsulates the Windows printer interface, and *TRegistry*, a low-level wrapper for the system registry and functions that operate on the registry. These are specific to the Windows environment.

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.

So you can see, this branch includes many different types of classes that are very useful to you as a developer.

- DevGuide: [Building applications with Delphi](#)

TPersistent Branch

[Topic groups](#)

[See also](#)

Objects in this branch of the VCL and 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.

Following are some of the other classes in the TPersistent branch of the hierarchy:

- *TGraphicsObject*, an abstract base class for graphics objects such as: *TBrush*, *TFont*, and *TPen*.
- *TGraphic*, an abstract base class for objects such as icons and bitmaps that can store and display visual images: *TBitmap* and *TIcon* (and for Windows development only: *TMetafile*).
- *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.

- [DevGuide: Building applications with Delphi](#)

TComponent Branch

[Topic groups](#)

[See also](#)

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 the VCL and CLX. Delphi 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 the VCL and 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 the timer functions.
- *TOpenDialog*, *TSaveDialog*, *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.

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.

In CLX, 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.

- DevGuide: [Building applications with Delphi](#)

TControl Branch

[Topic groups](#)

[See also](#)

The *TControl* branch consists of components that descend from *TControl* but not *TWinControl* (*TWidgetControl* in *CLX*). 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 window (or widget in *CLX*), 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 window (or widget) of their own
- Those that use the window (or widget) of their "parent"

Controls that have their own window are called "windowed" controls (*VCL*) or "widget-based" controls (*CLX*) and descend from *TWinControl* (*TWidgetControl* in *CLX*). Buttons and check boxes fall into this class. See [TWinControl branch](#) for details on windowed controls.

Controls that use a parent window (or widget) are called "graphic" controls and descend from *TGraphicControl*. Image and label controls fall into this class. In the *VCL*, the main difference between these types of components is that graphic controls do not maintain a window handle, and thus cannot receive the input focus. In *CLX*, 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:

Control	Description
TImage	Displays graphical images.
TLabel	Displays text on a form.
TBevel	Represents a beveled outline.
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.

- DevGuide: [Building applications with Delphi](#)

TWinControl Branch

[Topic groups](#)

[See also](#)

The *TWinControl* branch (*TWidgetControl* replaces *TWinControl* in *CLX*) includes all controls that descend from *TWinControl*. *TWinControl* is the base class for all windowed controls, including many of the items that you will use in the user interface of an application.

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 windowed and widget-based controls:

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

Descendants of *TWinControl* (*TWidgetControl* in *CLX*) 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 Delphi must draw (such as *TDBNavigator*, *TMediaPlayer* (VCL only), *TGauge* (VCL only), and so on). Direct descendants of *TWinControl* (*TWidgetControl* in *CLX*) 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 window handle but do not encapsulate a standard control that includes the ability to repaint itself. You never have to worry about how the controls render themselves or how they respond to events—Delphi completely encapsulates this behavior for you.

- DevGuide: [Building applications with Delphi](#)

Properties common to TControl

[Topic groups](#)

[See also](#)

All visual controls (descendants of *TControl*) share certain properties including:

- [Action properties](#)
- [Position, size, and alignment properties](#)
- [Display properties](#)
- [Parent properties](#)
- [A navigation property](#)
- [Drag-and-drop properties](#)
- [Drag-and-dock properties](#) (VCL only)

While these properties are inherited from *TControl*, they are published—and hence appear in the Object Inspector—only for components to which they are applicable. For example, *TImage* does not publish the *Color* property, since its color is determined by the graphic it displays.

- DevGuide: [Building applications with Delphi](#)

TControl action properties

[Topic groups](#)

[See also](#)

Actions let you share common code for performing actions (for example, when a tool bar 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.

- *Action* designates the action associated with the control.
- *ActionLink* contains the action link object associated with the control.

- DevGuide: [Building applications with Delphi](#)

TControl position, size, and alignment properties

[Topic groups](#)

[See also](#)

This set of properties defines the position and size of a control on the parent control:

- *Height* sets the vertical size.
- *Width* sets the horizontal size.
- *Top* positions the top edge.
- *Left* positions the left edge.
- *AutoSize* specifies whether the control sizes itself automatically to accommodate its contents.
- *Align* determines how the control aligns within its container (parent control).
- *Anchor* specifies how the control is anchored to its parent (VCL only).

This set of properties determine the height, width, and overall size of the control's client area:

- *ClientHeight* specifies the height of the control's client area in pixels.
- *ClientWidth* specifies the width of the control's client area in pixels.

These properties aren't accessible in nonvisual components, but Delphi does keep track of where you place the component icons on your forms. Most of the time you'll set and alter these properties by manipulating the control's image on the form or using the Alignment palette. You can, however, alter them at runtime.

- DevGuide: [Building applications with Delphi](#)

TControl display properties

[Topic groups](#)

[See also](#)

The following properties govern the general appearance of a control:

- *Color* changes the background color of a control.
- *Font* changes the color, type family, style, or size of text.
- *Cursor* specifies the image used to represent the mouse pointer when it passes into the region covered by the control.
- *DesktopFont* specifies whether the control uses the Windows icon font when writing text (VCL only).

- DevGuide: [Building applications with Delphi](#)

TControl parent properties

[Topic groups](#)

[See also](#)

To maintain a consistent appearance across your application, you can make any control look like its container—called its *parent*—by setting the *parent* properties to *True*.

- *ParentColor* determines where a control looks for its color information.
- *ParentFont* determines where a control looks for its font information.
- *ParentShowHint* determines where a control looks to find out if its Help Hint should be shown.

- DevGuide: [Building applications with Delphi](#)

TControl navigation property

[Topic groups](#)

[See also](#)

The following property determines how users navigate among the controls in a form:

- *Caption* contains the text string that labels a component. To underline a character in a string, include an ampersand (&) before the character. This type of character is called an accelerator key. The user can then select the control or menu item by pressing *Alt* while typing the underlined character.

- DevGuide: [Building applications with Delphi](#)

TControl Drag-and-drop properties

[Topic groups](#)

[See also](#)

Two component properties affect drag-and-drop behavior:

- *DragMode* determines how dragging starts. By default, *DragMode* is *dmManual*, and the application must call the *BeginDrag* method to start dragging. When *DragMode* is *dmAutomatic*, dragging starts as soon as the mouse button goes down.
- *DragCursor* determines the shape of the mouse pointer when it is over a draggable component (VCL only).

- DevGuide: [Building applications with Delphi](#)

TControl drag-and-dock properties (VCL only)

[Topic groups](#)

[See also](#)

The following properties control drag-and-dock behavior in *TWinControl* and *TWidgetControl*:

- *Floating* indicates whether the control is floating.
- *DragKind* specifies whether the control is being dragged normally or for docking.
- *DragMode* determines how the control initiates drag-and-drop or drag-and-dock operations.
- *FloatingDockSiteClass* specifies the class of the temporary control that hosts the control when it is floating.
- *DragCursor* is the cursor that is shown while dragging.
- *DockOrientation* specifies how the control is docked relative to other controls docked in the same parent.
- *HostDockSite* specifies the control in which the control is docked.
- *LRDockWidth* specifies the width of the control when it is docked horizontally.
- *TBDockHeight* specifies the height of the control when it is docked vertically.
- *UnDockHeight* specifies the height of the control when it is floating.
- *UnDockWidth* specifies the width of the control when it is floating.

For more information, see [Implementing drag-and-dock in controls](#) and [TDrag-and-dock properties](#).

- DevGuide: [Building applications with Delphi](#)

Standard events common to TControl

[Topic groups](#)

The VCL defines a set of standard events for its controls. The following events are declared as part of the *TControl* class, and are therefore available for all classes derived from *TControl*:

- *OnClick* occurs when the user clicks the control.
- *OnContextPopup* occurs when the user right-clicks the control or otherwise invokes the popup menu (such as using the keyboard).
- *OnCanResize* occurs when an attempt is made to resize the control.
- *OnResize* occurs immediately after the control is resized.
- *OnConstrainedResize* occurs immediately after *OnCanResize*.
- *OnStartDock* occurs when the user begins to drag a control with a *DragKind* of *dkDock* (VCL only).
- *OnEndDock* occurs when the dragging of an object ends, either by docking the object or by canceling the dragging (VCL only).
- *OnStartDrag* occurs when the user begins to drag the control or an object it contains by left-clicking on the control and holding the mouse button down.
- *OnEndDrag* occurs when the dragging of an object ends, either by dropping the object or by canceling the dragging.
- *OnDragDrop* occurs when the user drops an object being dragged.
- *OnMouseMove* occurs when the user moves the mouse pointer while the mouse pointer is over a control.
- *OnDbClick* occurs when the user double-clicks the primary mouse button when the mouse pointer is over the control.
- *OnDragOver* occurs when the user drags an object over a control (VCL only).
- *OnMouseDown* occurs when the user presses a mouse button with the mouse pointer over a control.
- *OnMouseUp* occurs when the user releases a mouse button that was pressed with the mouse pointer over a component.

- DevGuide: [Building applications with Delphi](#)

Properties common to `TWinControl` and `TWidgetControl`

[Topic groups](#)

[See also](#)

All windowed controls (descendants of *TWinControl* in the VCL and *TWidgetControl* in CLX) share certain properties including:

- [Information about the control](#)
- [Border style display properties](#)
- [Navigation properties](#)
- [Drag-and-dock properties \(VCL only\)](#)

While these properties are inherited from *TWinControl* and *TWidgetControl*, they are published—and hence appear in the Object Inspector—only for controls to which they are applicable.

- DevGuide: [Building applications with Delphi](#)

TWinControl general information properties

[Topic groups](#)

[See also](#)

The general information properties contain information about the appearance of the *TWinControl* and *TWidgetControl*, client area size and origin, windows assigned information, and help context information.

- *ClientOrigin* specifies the screen coordinates (in pixels) of the top left corner of a control's client area. The screen coordinates of a control that is descended from *TControl* and not *TWinControl* are the screen coordinates of the control's parent added to its *Left* and *Top* properties.
- *ClientRect* returns a rectangle with its *Top* and *Left* properties set to zero, and its *Bottom* and *Right* properties set to the control's *Height* and *Width*, respectively. *ClientRect* is equivalent to `Rect(0, 0, ClientWidth, ClientHeight)`.
- *Brush* determines the color and pattern used for painting the background of the control.
- *HelpContext* provides a context number for use in calling context-sensitive online Help.
- *Handle* provides access to the window or widget handle of the control.

- DevGuide: [Building applications with Delphi](#)

TWinControl border Style Display Properties

[Topic groups](#)

[See also](#)

The bevel properties control the appearance of the beveled lines, boxes, or frames on the forms and windowed controls in your application.

Many more objects in the VCL publish these properties; they are not all available in CLX and the border style properties are published on fewer objects.

- *InnerBevel* specifies whether the inner bevel has a raised, lowered, or flat look (VCL only).
- *BevelKind* specifies the type of bevel if the control has beveled edges (VCL only).
- *BevelOuter* specifies whether the outer bevel has a raised, lowered, or flat look.
- *BevelWidth* specifies the width, in pixels, of the inner and outer bevels.
- *BorderWidth* is used to get or set the width of the control's border.
- *BevelEdges* is used to get or set which edges of the control are beveled.

- DevGuide: [Building applications with Delphi](#)

TWinControl navigation properties

[Topic groups](#)

[See also](#)

Two additional properties determine how users navigate among the controls on a form:

- *TabOrder* indicates the position of the control in its parent's tab order, the order in which controls receive focus when the user presses the *Tab* key. Initially, tab order is the order in which the components are added to the form, but you can change this by changing *TabOrder*. *TabOrder* is meaningful only if *TabStop* is *True*.
- *TabStop* determines whether the user can tab to a control. If *TabStop* is *True*, the control is in the tab order.

- DevGuide: [Building applications with Delphi](#)

TWinControl drag-and-dock properties (VCL only)

[Topic groups](#)

[See also](#)

The following properties manage drag-and-dock behavior in VCL objects:

- *UseDockManager* specifies whether the dock manager is used in drag-and-dock operations.
- *DockClientCount* specifies the number of controls that are docked on the windowed control.
- *VisibleDockClientCount* specifies the number of visible controls that are docked on the windowed control.
- *DockManager* specifies the control's dock manager interface.
- *DockClients* lists the controls that are docked to the windowed control.
- *DockSite* specifies whether the control can be the target of drag-and-dock operations.

For more information, see [Implementing drag-and-dock in controls](#) and [TControl Drag-and-dock properties](#).

- DevGuide: [Building applications with Delphi](#)

Standard events common to `TWinControl` and `TWidgetControl`

[Topic groups](#)

The following events exist for all controls derived from `TWinControl` in the VCL (this also includes all the controls that Windows defines) and in `TWidgetControl` in CLX. These events are in addition to those that exist in all controls.

- `OnEnter` occurs when the control is about to receive focus.
- `OnKeyDown` occurs on the down stroke of a key press.
- `OnKeyPress` occurs when a user presses a single character key.
- `OnKeyUp` occurs when the user releases a key that has been pressed.
- `OnExit` occurs when the input focus shifts away from one control to another.
- `OnMouseWheel` occurs when the mouse wheel is rotated.
- `OnMouseWheelDown` occurs when the mouse wheel is rotated downward.
- `OnMouseWheelUp` occurs when the mouse wheel is rotated upward.

The following events relate to docking and are available in the VCL only:

- `OnUnDock` occurs when the application tries to undock a control that is docked to a windowed control (VCL only).
- `OnDockDrop` occurs when another control is docked to the control (VCL only).
- `OnDockOver` occurs when another control is dragged over the control (VCL only).
- `OnGetSiteInfo` returns the control's docking information (VCL only).

- DevGuide: [Building applications with Delphi](#)

Using components

[Topic groups](#)

[See also](#)

Many visual components are provided in the development environment itself on the Component palette. All visual design work in Delphi 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. Delphi takes care of the underlying programming details.

Delphi 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. Handy VCL controls such as a timer, paint box, media player, and OLE container are on the System page.

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

- All Delphi 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.
- Components can only be allocated on the heap, not on the stack.
- Properties of components intrinsically contain runtime type information.
- Components can be added to the Component palette in the Delphi user interface and manipulated on a form.

Components often achieve a better degree of encapsulation than is usually found in standard classes. For example, consider the use of a dialog containing a push button. In a Windows program developed using VCL components, when a user clicks on the button, the system generates a `WM_LBUTTONDOWN` message. The program must catch this message (typically in a **switch** statement, a message map, or a response table) and dispatch it to a routine that will execute in response to the message.

Most Windows messages (VCL) or system events (CLX) are handled by Delphi components. When you want to respond to a message, you only need to provide an event handler.

- DevGuide: [Building applications with Delphi](#)

Setting component properties

[Topic groups](#)

[See also](#)

Published properties can be set at design time in the Object Inspector and, in some cases, with special property editors.

To set properties at runtime, assign them new values in your application source code.

For information about the properties of each component, see the online Help.

- DevGuide: [Building applications with Delphi](#)

Using the Object Inspector

[Topic groups](#)

[See also](#)

When you select a component on a form, the Object Inspector displays its published properties and (when appropriate) allows you to edit them. Use the *Tab* key to toggle between the Value column and the Property column. When the cursor is in the Property column, you can navigate to any property by typing the first letters of its name. For properties of Boolean or enumerated types, you can choose values from a drop-down list or toggle their settings by double-clicking in Value column.

If a plus (+) symbol appears next to a property name, clicking the plus symbol or typing '+' when the property has focus displays a list of subvalues for the property. Similarly, if a minus (-) symbol appears next to the property name, clicking the minus symbol or typing '-' hides the subvalues.

By default, properties in the Legacy category are not shown; to change the display filters, right-click in the Object Inspector and choose View. For more information, see [Property and event categories in the Object Inspector](#).

When more than one component is selected, the Object Inspector displays all properties—except *Name*—that are shared by the selected components. If the value for a shared property differs among the selected components, the Object Inspector displays either the default value or the value from the first component selected. When you change a shared property, the change applies to all selected components.

- DevGuide: [Building applications with Delphi](#)

Using property editors

[Topic groups](#)

Some properties, such as *Font*, have special property editors. Such properties appear with ellipsis marks (...) next to their values when the property is selected in the Object Inspector. To open the property editor, double-click in the Value column, click the ellipsis mark, or type *Ctrl+Enter* when focus is on the property or its value. With some components, double-clicking the component on the form also opens a property editor.

Property editors let you set complex properties from a single dialog box. They provide input validation and often let you preview the results of an assignment.

- DevGuide: [Building applications with Delphi](#)

Setting properties at runtime

[Topic groups](#)

[See also](#)

Any writable property can be set at runtime in your source code. For example, you can dynamically assign a caption to a form:

```
Form1.Caption := MyString;
```

- DevGuide: [Building applications with Delphi](#)

Calling methods

[Topic groups](#)

[See also](#)

Methods are called just like ordinary procedures and functions. For example, visual controls have a *Repaint* method that refreshes the control's image on the screen. You could call the *Repaint* method in a draw-grid object like this:

```
DrawGrid1.Repaint;
```

As with properties, the scope of a method name determines the need for qualifiers. If you want, for example, to repaint a form within an event handler of one of the form's child controls, you don't have to prepend the name of the form to the method call:

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

For more information about scope, see [Scope and qualifiers](#).

- DevGuide: [Building applications with Delphi](#)

Working with events and event handlers

[Topic groups](#)

[See also](#)

In Delphi, almost all the code you write is executed, directly or indirectly, in response to *events*. An event is a special kind of property that represents a runtime occurrence, often a user action. The code that responds directly to an event—called an *event handler*—is an Object Pascal procedure. The sections that follow show how to

- [Generate a new event handler](#)
- [Generate a handler for a component's default event](#)
- [Locate event handlers](#)
- [Associate an event with an existing event handler](#)
- [Associate menu events with event handlers](#)
- [Delete event handlers](#)

- DevGuide: [Building applications with Delphi](#)

Generating a new event handler

[Topic groups](#)

[See also](#)

Delphi can generate skeleton event handlers for forms and other components. To create an event handler,

- 1 Select a component.
- 2 Click the Events tab in the Object Inspector. The Events page of the Object Inspector displays all events defined for the component.
- 3 Select the event you want, then double-click the Value column or press *Ctrl+Enter*. Delphi generates the event handler in the Code editor and places the cursor inside the **begin...end** block.
- 4 Inside the **begin...end** block, type the code that you want to execute when the event occurs.

- DevGuide: [Building applications with Delphi](#)

Generating a handler for a component's default event

[Topic groups](#)

[See also](#)

Some components have a *default* event, which is the event the component most commonly needs to handle. For example, a button's default event is *OnClick*. To create a default event handler, double-click the component in the Form Designer; this generates a skeleton event-handling procedure and opens the Code editor with the cursor in the body of the procedure, where you can easily add code.

Not all components have a default event. Some components, such as *TBevel*, don't respond to any events. Other components respond differently when you double-click on them in the Form Designer. For example, many components open a default property editor or other dialog when they are double-clicked at design time.

- DevGuide: [Building applications with Delphi](#)

Locating event handlers

[Topic groups](#)

[See also](#)

If you generated a default event handler for a component by double-clicking it in the Form Designer, you can locate that event handler in the same way. Double-click the component, and the Code editor opens with the cursor at the beginning of the event-handler body.

To locate an event handler that's not the default,

- 1 In the form, select the component whose event handler you want to locate.
- 2 In the Object Inspector, click the Events tab.
- 3 Select the event whose handler you want to view and double-click in the Value column. The Code editor opens with the cursor at the beginning of the event-handler body.

■ DevGuide: [Building applications with Delphi](#)

Associating an event with an existing event handler

[Topic groups](#)

[See also](#)

You can reuse code by writing event handlers that respond to more than one event. For example, many applications provide speed buttons that are equivalent to drop-down menu commands. When a button initiates the same action as a menu command, you can write a single event handler and assign it to both the button's and the menu item's *OnClick* event.

To associate an event with an existing event handler,

- 1 On the form, select the component whose event you want to handle.
- 2 On the Events page of the Object Inspector, select the event to which you want to attach a handler.
- 3 Click the down arrow in the Value column next to the event to open a list of previously written event handlers. (The list includes only event handlers written for events of the same name on the same form.) Select from the list by clicking an event-handler name.

The procedure above is an easy way to reuse event handlers. [Action lists](#) and in the VCL, [action bandsSettingUpActionBands](#), however, provide powerful tools for centrally organizing the code that responds to user commands. Action lists can be used in cross-platform applications, whereas action bands cannot.

- DevGuide: [Building applications with Delphi](#)

Using the Sender parameter

[Topic groups](#)

[See also](#)

In an event handler, the *Sender* parameter indicates which component received the event and therefore called the handler. Sometimes it is useful to have several components share an event handler that behaves differently depending on which component calls it. You can do this by using the *Sender* parameter in an **if...then...else** statement. For example, the following code displays the title of the application in the caption of a dialog box only if the *OnClick* event was received by *Button1*.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  if Sender = Button1 then
    AboutBox.Caption := 'About ' + Application.Title
  else
    AboutBox.Caption := '';
  AboutBox.ShowModal;
end;
```

- DevGuide: [Building applications with Delphi](#)

Displaying and coding shared events

[Topic groups](#)

[See also](#)

When components share events, you can display their shared events in the Object Inspector. First, select the components by holding down the *Shift* key and clicking on them in the Form Designer; then choose the Events tab in the Object Inspector. From the Value column in the Object Inspector, you can now create a new event handler for, or assign an existing event handler to, any of the shared events.

- DevGuide: [Building applications with Delphi](#)

Associating menu events with event handlers

[Topic groups](#)

[See also](#)

The Menu Designer, along with the *MainMenu* and *PopupMenu* components, make it easy to supply your application with drop-down and pop-up menus. For the menus to work, however, each menu item must respond to the *OnClick* event, which occurs whenever the user chooses the menu item or presses its accelerator or shortcut key. This topic explains how to associate event handlers with menu items. For information about the Menu Designer and related components, see [Creating and managing menus](#).

To create an event handler for a menu item,

- 1 Open the Menu Designer by double-clicking on a *MainMenu* or *PopupMenu* object.
- 2 Select a menu item in the Menu Designer. In the Object Inspector, make sure that a value is assigned to the item's *Name* property.
- 3 From the Menu Designer, double-click the menu item. Delphi generates an event handler in the Code editor and places the cursor inside the **begin...end** block.
- 4 Inside the **begin...end** block, type the code that you want to execute when the user selects the menu command.

To associate a menu item with an existing *OnClick* event handler,

- 1 Open the Menu Designer by double-clicking on a *MainMenu* or *PopupMenu* object.
- 2 Select a menu item in the Menu Designer. In the Object Inspector, make sure that a value is assigned to the item's *Name* property.
- 3 On the Events page of the Object Inspector, click the down arrow in the Value column next to *OnClick* to open a list of previously written event handlers. (The list includes only event handlers written for *OnClick* events on this form.) Select from the list by clicking an event handler name.

- DevGuide: [Building applications with Delphi](#)

Deleting event handlers

[Topic groups](#)

[See also](#)

When you delete a component from a form using the Form Designer, Delphi removes the component from the form's type declaration. It does not, however, delete any associated methods from the unit file, since these methods may still be called by other components on the form. You can manually delete a method—such as an event handler—but if you do so, be sure to delete both the method's forward declaration (in the **interface** section of the unit) and its implementation (in the **implementation** section); otherwise you'll get a compiler error when you build your project.

■ DevGuide: [Building applications with Delphi](#)

VCL and CLX components

[Topic groups](#)

[See also](#)

The Component palette contains a selection of components that handle a wide variety of programming tasks. You can add, remove, and rearrange components on the palette, and you can create component *templates* and *frames* that group several components.

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 Delphi you are running. The following table lists typical default pages and components available for creating applications. Some of the tabs and components are not cross platform and the table points them out. You can use some VCL-specific nonvisual components in Windows-only CLX applications, however, the applications will not be cross-platform unless you isolate these portions of the code.

<u>Page name</u>	<u>Description</u>	<u>Cross platform?</u>
Standard	Standard controls, menus	Yes
Additional	Specialized controls	Yes except ApplicationEvents and CustomizeDlg
Win32	Windows common controls	Many of the same components are on the Common Controls tab that appears instead when creating CLX applications; RichEdit, UpDown, HotKey, Animate, DateTimePicker, MonthCalendar, Coolbar, PageScroller, and ComboBoxEx are not cross-platform
System	Components and controls for system-level access, including timers, multimedia, and DDE	Timer is but PaintBox, MediaPlayer, OleContainer, and the Dde components are not
Data Access	Components for working with database data that are not tied to any particular data access mechanism	Yes
Data Controls	Visual, data-aware controls	Yes except for DBRichEdit, DBCtrlGrid, and DBChart
dbExpress	Database controls that use dbExpress, a cross-platform, database-independent layer that provides methods for dynamic SQL processing. It defines a common interface for accessing SQL servers.	Yes
DataSnap	Components used for creating multi-tiered database applications	No but can be used in Windows CLX applications
BDE	Components that provide data access through the Borland Database Engine	No but can be used in Windows CLX applications
ADO	Components that provide data access through the ADO framework	No but can be used in Windows CLX applications
InterBase	Components that provide direct access to InterBase	Yes
InternetExpress	Components that are simultaneously a Web Server application and the client of a multi-tiered database application	No but can be used in Windows CLX applications
Internet	Components for Internet communication protocols and Web applications	Yes except for ClientSocket, ServerSocket, QueryTableProducer, XMLDoc, and WebBrowser
WebSnap	Components for building Web server applications	No but can be used in Windows CLX applications
FastNet	NetMasters Internet controls	No but can be used in Windows CLX applications
QReport	QuickReport components for creating embedded reports	No but can be used in Windows CLX applications
Dialogs	Commonly used dialog boxes	Yes except for OpenPictureDialog, SavePictureDialog, PrinterSetup-Dialog, and PageSetupDialog
Win 3.1	Old style Win 3.1 components	No
Samples	Sample custom components	No

ActiveX	Sample ActiveX controls; see Microsoft documentation (msdn.microsoft.com)	No
COM+	Component for handling COM+ events	No but can be used in Windows CLX applications
WebServices	Components for writing applications that implement or use SOAP-based Web Services	No but can be used in Windows CLX applications
Servers	COM Server examples for Microsoft Excel, Word, and so on (see Microsoft MSDN documentation)	No but can be used in Windows CLX applications
Indy Clients	Cross-platform Internet components for the client (open source Winshoes Internet components)	Yes
Indy Servers	Cross-platform Internet components for the server (open source Winshoes Internet components)	Yes
Indy Misc	Additional cross-platform Internet components (open source Winshoes Internet components)	Yes

The online Help provides information about the components on the Component palette. Some of the components on the ActiveX, Servers, and Samples pages, however, are provided as examples only and are not documented.

- DevGuide: [Building applications with Delphi](#)

Adding custom components to the component palette

[Topic groups](#)

[See also](#)

You can install custom components—written by yourself or third parties—on the Component palette and use them in your applications. To write a component, see [Overview of component creation](#). To install an existing component, see [Installing component packages](#).

■ DevGuide: [Building applications with Delphi](#)

Text controls

[Topic groups](#)

[See also](#)

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

TRichEdit

Edit multiple lines of text using rich text format (VCL only)

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.

- DevGuide: [Building applications with Delphi](#)

Text control properties

[Topic groups](#)

[See also](#)

Following are some of the important properties of text controls:

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

- DevGuide: [Building applications with Delphi](#)

Properties of memo and rich text controls

[Topic groups](#)

[See also](#)

Memo and rich text controls, which handle multiple lines of text, have several properties in common. Note that rich text controls are not cross-platform.

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 and rich text 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.
- *OEMConvert* determines whether the text is temporarily converted from ANSI to OEM as it is entered. This is useful for validating file names (VCL only).
- *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.
- *AutoSelect* determines whether the text is automatically selected (highlighted) when the control becomes active.
- *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.

- DevGuide: [Building applications with Delphi](#)

Rich text controls (VCL only)

[Topic groups](#)

[See also](#)

The rich edit (*TRichEdit*) component is a memo control that supports rich text formatting, printing, searching, and drag-and-drop of text. It allows you to specify font properties, alignment, tabs, indentation, and numbering.

- DevGuide: [Building applications with Delphi](#)

Specialized input controls

[Topic groups](#)

[See also](#)

The following components provide additional ways of capturing input.

Use this component:

TScrollBar

TTrackBar

TUpDown

THotKey

TSpinEdit

When you want users to do this:

Select values on a continuous range

Select values on a continuous range (more visually effective than a scroll bar)

Select a value from a spinner attached to an edit component (VCL only)

Enter *Ctrl/Shift/Alt* keyboard sequences (VCL only)

Select a value from a spinner widget (CLX only)

- DevGuide: [Building applications with Delphi](#)

Scroll bars

[Topic groups](#)

[See also](#)

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*.

- DevGuide: [Building applications with Delphi](#)

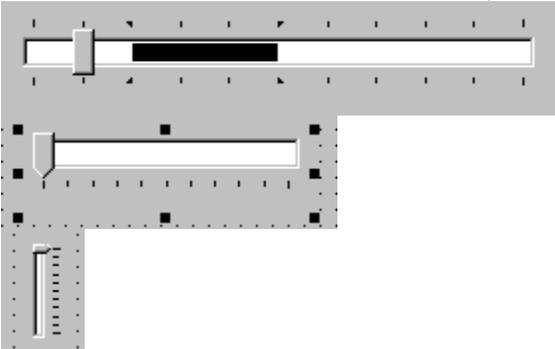
Track bars

[Topic groups](#)

[See also](#)

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.
- 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.



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 *Page Up* and *Page Down*.

- DevGuide: [Building applications with Delphi](#)

Up-down controls (VCL only)

[Topic groups](#)

[See also](#)

An up-down control (*TUpDown*) consists of a pair of arrow buttons that allow users to change an integer value in fixed increments. The current value is given by the *Position* 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.

- DevGuide: [Building applications with Delphi](#)

Spin edit controls (CLX only)

[Topic groups](#)

[See also](#)

A spin edit control (*TSpinEdit*) 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.

- DevGuide: [Building applications with Delphi](#)

Hot key controls (VCL only)

[Topic groups](#)

[See also](#)

Use the hot key component (*THotKey*) to assign a keyboard shortcut that transfers focus to any control. The *HotKey* property contains the current key combination and the *Modifiers* property determines which keys are available for *HotKey*.

The hot key component can be assigned as the *Shortcut* property of a menu item. Then, when a user enters the key combination specified by the *HotKey* and *Modifiers* properties, Windows activates the menu item.

- DevGuide: [Building applications with Delphi](#)

Splitter control

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Buttons and similar controls

[Topic groups](#)

[See also](#)

Aside from menus, buttons provide the most common way to invoke a command in an application. Delphi offers several button-like controls:

Use this component:

To do this:

TButton

Present command choices on buttons with text

TBitBtn

Present command choices on buttons with text and glyphs

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

TCoolBar

Display a collection of windowed controls within movable, resizable bands
(VCL only)

- DevGuide: [Building applications with Delphi](#)

Button controls

[Topic groups](#)

[See also](#)

Users click button controls to initiate actions. You can assign an action to a *TButton* 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.

- DevGuide: [Building applications with Delphi](#)

Bitmap buttons

[Topic groups](#)

[See also](#)

A bitmap button (*BitBtn*) 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 is to the left of any text. To move it, use the *Layout* property.
- The glyph and text are automatically centered in 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.

- DevGuide: [Building applications with Delphi](#)

Speed buttons

[Topic groups](#)

[See also](#)

Speed buttons, 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.

- DevGuide: [Building applications with Delphi](#)

Check boxes

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Radio buttons

[Topic groups](#)

[See also](#)

Radio 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](#) 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.

- DevGuide: [Building applications with Delphi](#)

Toolbars

[Topic groups](#)

[See also](#)

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 *ToolBar* component, then right-click and choose New Button to add buttons to the toolbar.

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 using [action lists](#) or [action bands](#).

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

- DevGuide: [Building applications with Delphi](#)

Cool bars (VCL only)

[Topic groups](#)

[See also](#)

A cool bar contains child controls that can be moved and resized independently. Each control resides on an individual band. The user positions the controls by dragging the sizing grip to the left of each band.

The cool bar requires version 4.70 or later of COMCTL32.DLL (usually located in the Windows\System or Windows\System32 directory) at both design time and runtime. Cool bars cannot be used in cross-platform applications.

- The *Bands* property holds a collection of *TCoolBand* objects. At design time, you can add, remove, or modify bands with the Bands editor. To open the Bands editor, select the *Bands* property in the Object Inspector, then double-click in the Value column to the right, or click the ellipsis (...) button. You can also create bands by adding new windowed controls from the palette.
- The *FixedOrder* property determines whether users can reorder the bands.
- The *FixedSize* property determines whether the bands maintain a uniform height.

- DevGuide: [Building applications with Delphi](#)

Handling lists

[Topic groups](#)

[See also](#)

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

[*TCheckListBox*](#)

A list with a check box in front of each item

[*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

[*TDateTimePicker*](#)

A list box for entering dates or times (VCL only)

[*TMonthCalendar*](#)

A calendar for selecting dates (VCL only)

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](#).

- DevGuide: [Building applications with Delphi](#)

List boxes and check-list boxes

[Topic groups](#)

[See also](#)

List boxes (*TListBox*) and check-list boxes display lists from which users can select items.

- *Items* uses a *TStrings* object to fill the 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.
- *IntegralHeight* specifies whether the list box shows only entries that fit completely in the vertical space (VCL only).
- *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](#).

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.

- DevGuide: [Building applications with Delphi](#)

Combo boxes

[Topic groups](#)

[See also](#)

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* if you want 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 combo box with a fixed list that does not close. Be sure to resize the combo box so that the list items are 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](#).

At runtime, CLX combo boxes work differently than VCL combo boxes. In CLX (but not in the VCL combo box), 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.

- DevGuide: [Building applications with Delphi](#)

Tree views

[Topic groups](#)

[See also](#)

A tree view (*TTreeView*) displays items in an indented outline. The control provides buttons that allow nodes to be expanded and collapsed. 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.

- *Indent* sets the number of pixels horizontally separating items from their parents.
- *ShowButtons* enables the display of '+' and '-' buttons to indicate whether an item can be expanded.
- *ShowLines* enables display of connecting lines to show hierarchical relationships (VCL only).
- *ShowRoot* determines whether lines connecting the top-level items are displayed (VCL only).

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. You can change the items at runtime by using the methods of the *Items* property, which is an object of type *TTreeNode*. *TTreeNode* has methods for adding, deleting, and navigating the items in the tree view.

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

- DevGuide: [Building applications with Delphi](#)

List views

[Topic groups](#)

[See also](#)

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

- *vsIcon* and *vsSmallIcon* display each item as an icon with a label. Users can drag items within the list view window (VCL only).
- *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.

- DevGuide: [Building applications with Delphi](#)

Date-time pickers and month calendars (VCL only)

[Topic groups](#)

[See also](#)

The DateTimePicker component displays a list box for entering dates or times, while the MonthCalendar component presents a calendar for entering dates or ranges of dates. To use these components, you must have version 4.70 or later of COMCTL32.DLL (usually located in the Windows\System or Windows\System32 directory) at both design time and runtime. They are not available for use in cross-platform applications.

- DevGuide: [Building applications with Delphi](#)

Grouping components

[Topic groups](#)

[See also](#)

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

Use this component:

[TGroupBox](#)

[TRadioGroup](#)

[TPanel](#)

[TScrollBar](#)

[TTabControl](#)

[TPageControl](#)

[THeaderControl](#)

When you want this:

A standard group box with a title

A simple group of radio buttons

A more visually flexible group of controls

A scrollable region containing controls

A set of mutually exclusive notebook-style tabs

A set of mutually exclusive notebook-style tabs with corresponding pages, each of which may contain other controls

Resizable column headers

- DevGuide: [Building applications with Delphi](#)

Group boxes and radio groups

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Panels

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Scroll boxes

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Tab controls

[Topic groups](#)

[See also](#)

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 change the appearance of the control when the tabs are clicked, you need to write an *OnChange* event handler. To create a multipage dialog box, use a page control instead.

- DevGuide: [Building applications with Delphi](#)

Page controls

[Topic groups](#)

[See also](#)

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

- DevGuide: [Building applications with Delphi](#)

Header controls

[Topic groups](#)

[See also](#)

A header control (*THeaderControl*) 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*).

- DevGuide: [Building applications with Delphi](#)

Providing visual feedback

[Topic groups](#)

[See also](#)

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. 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:

[*TLabel*](#) and [*TStaticText*](#)

[*TStatusBar*](#)

[*TProgressBar*](#)

Hint and *ShowHint*

HelpContext and *HelpFile*

To do this:

Display non-editable text

Display a status region (usually at the bottom of a window)

Show the amount of work completed for a particular task

Activate fly-by or “tooltip” help

Link context-sensitive online Help

- DevGuide: [Building applications with Delphi](#)

Labels and static text components

[Topic groups](#)

[See also](#)

Labels (*TLabel*) display text 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 a non-windowed control (not widget-based in CLX), so it cannot receive focus; when you need a label with a window handle, use *TStaticText* instead.

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.
- *ShowAccelChar* determines whether the label can display an underlined accelerator character. If *ShowAccelChar* is *True*, any character preceded by an ampersand (&) appears underlined and enables an 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*.

- DevGuide: [Building applications with Delphi](#)

Status bars

[Topic groups](#)

[See also](#)

Although you can use a panel to make a status bar, it is simpler to use the status bar component. 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.

- DevGuide: [Building applications with Delphi](#)

Progress bars

[Topic groups](#)

[See also](#)

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 displays a dotted line that grows from left to right.



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*.

- DevGuide: [Building applications with Delphi](#)

Help and hint properties

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Grids

[Topic groups](#)

[See also](#)

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

- DevGuide: [Building applications with Delphi](#)

Draw grids

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

String grids

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Value list editors (VCL only)

[Topic groups](#)

[See also](#)

TValueListEditor is a specialized grid for editing string lists that contain name/value pairs in the form Name=Value. The names and values are stored as a *TStrings* descendant that is the value of the *Strings* property. You can look up the value for any name using the *Values* property. *TValueListEditor* is not available for cross-platform programming.

The grid contains two columns, one for the names and one for the values. By default, the Name column is named "Key" and the Value column is named "Value". You can change these defaults by setting the *TitleCaptions* property. You can omit these titles using the *DisplayOptions* property (which also controls resize when you resize the control.)

You can control whether users can edit the Name column using the *KeyOptions* property. *KeyOptions* contains separate options to allow editing, adding new names, deleting names, and controlling whether new names must be unique.

You can control how users edit the entries in the Value column using the *ItemProps* property. Each item has a separate *TItemProp* object that lets you

- Supply an edit mask to limit the valid input.
- Specify a maximum length for values.
- Mark the value as read-only.
- Specify that the value list editor displays a drop-down arrow that opens a pick list of values from which the user can choose or an ellipsis button that triggers an event you can use for displaying a dialog in which users enter values.

If you specify that there is a drop-down arrow, you must supply the list of values from which the user chooses. These can be a static list (the *PickList* property of the *TItemProp* object) or they can be dynamically added at runtime using the value list editor's *OnGetPickList* event. You can also combine these approaches and have a static list that the *OnGetPickList* event handler modifies.

If you specify that there is an ellipsis button, you must supply the response that occurs when the user clicks that button (including the setting of a value, if appropriate). You provide this response by writing an *OnEditButtonClick* event handler.

- DevGuide: [Building applications with Delphi](#)

Displaying graphics

[Topic groups](#)

[See also](#)

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

Use this component:

[TImage](#)

[TShape](#)

[TBevel](#)

[TPaintBox](#)

[TAnimate](#)

To display:

Graphics files

Geometric shapes

3-D lines and frames

Graphics drawn by your program at runtime

AVI files (VCL only)

- DevGuide: [Building applications with Delphi](#)

Images

[Topic groups](#)

[See also](#)

The image component displays a graphical image, like a bitmap, icon, or metafile. 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](#).

- DevGuide: [Building applications with Delphi](#)

Shapes

[Topic groups](#)

[See also](#)

The shape component displays a geometric shape. It is a nonwindowed control (not widget-based in CLX) and therefore, 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*.

- DevGuide: [Building applications with Delphi](#)

Bevels

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Paint boxes

[Topic groups](#)

[See also](#)

The paint box (*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](#).

- DevGuide: [Building applications with Delphi](#)

Animation control (VCL only)

[Topic groups](#)

[See also](#)

The animation component is a window that silently displays an Audio Video Interleaved (AVI) clip. An AVI clip is a series of bitmap frames, like a movie. Although AVI clips can have sound, animation controls work only with silent AVI clips. The files you use must be either uncompressed AVI files or AVI clips compressed using run-length encoding (RLE). Animation control cannot be used in cross-platform programming.

Following are some of the properties of an animation component:

- *ResHandle* is the Windows handle for the module that contains the AVI clip as a resource. Set *ResHandle* at runtime to the instance handle or module handle of the module that includes the animation resource. After setting *ResHandle*, set the *ResID* or *ResName* property to specify which resource in the indicated module is the AVI clip that should be displayed by the animation control.
- Set *AutoSize* to *True* to have the animation control adjust its size to the size of the frames in the AVI clip.
- *StartFrame* and *StopFrame* specify in which frames to start and stop the clip.
- Set *CommonAVI* to display one of the common Windows AVI clips provided in Shell32.DLL.
- Specify when to start and interrupt the animation by setting the *Active* property to *True* and *False*, respectively, and how many repetitions to play by setting the *Repetitions* property.
- The *Timers* property lets you display the frames using a timer. This is useful for synchronizing the animation sequence with other actions, such as playing a sound track.

- DevGuide: [Building applications with Delphi](#)

Developing dialog boxes

[Topic groups](#)

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, saving, and printing 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*.

If you are developing cross-platform applications, you can use the dialogs provided with CLX in the QDialogs unit. For operating systems that have native dialog box types for common tasks, such as for opening or saving a file or for changing font or color, you can use the *UseNativeDialog* property. Set *UseNativeDialog* to *True* if your application will run in such an environment, and if you want it to use the native dialogs instead of the Qt dialogs.

- DevGuide: [Building applications with Delphi](#)

Using open dialog boxes

[Topic groups](#)

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 the *TOpenDialog 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.

- DevGuide: [Building applications with Delphi](#)

Using helper objects

[Topic groups](#)

[See also](#)

The VCL and CLX include a variety of nonvisual **objects** that simplify common programming tasks. The following topics describe a few Helper **objects** that make it easier to perform the following tasks:

- [Working with lists](#)
- [Working with string lists](#)
- [Changing the Windows registry and .INI files](#)
- [Using streams](#)

■ DevGuide: [Building applications with Delphi](#)

Working with lists

[Topic groups](#)

[See also](#)

The following objects provide functionality for creating and managing lists:

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

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

- DevGuide: [Building applications with Delphi](#)

Working with string lists

[Topic groups](#)

[See also](#)

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 VCL and CLX provide 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*).

A string-list property appears in the Object Inspector with *TStrings* in the Value column. Double-click *TStrings* to open the String List editor, where you can edit, add, or delete lines.

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](#)

- DevGuide: [Building applications with Delphi](#)

Loading and saving string lists

[Topic groups](#)

[See also](#)

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 WIN.INI file into a memo field and makes a backup copy called WIN.BAK.

```
procedure EditWinIni;
var
  FileName: string; { storage for file name }
begin
  FileName := 'C:\WINDOWS\WIN.INI'; { 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;
```

■ DevGuide: [Building applications with Delphi](#)

Creating a new string list

[Topic groups](#)

[See also](#)

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 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
{For CLX: 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 *.DFM}
```

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

- DevGuide: [Building applications with Delphi](#)

Manipulating strings in a list

[Topic groups](#)

[See also](#)

Operations commonly performed on string lists include:

- [Counting the strings in a list](#)
- [Accessing a particular string](#)
- [Finding the position of a string in the 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](#)

- DevGuide: [Building applications with Delphi](#)

Counting the strings in a list

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Accessing a particular string

[Topic groups](#)

[See also](#)

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.';
```

■ DevGuide: [Building applications with Delphi](#)

^\$ Locating items in a string list

[Topic groups](#)

[See also](#)

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('WIN.INI') > -1 ...
```

- DevGuide: [Building applications with Delphi](#)

Iterating through strings in a list

[Topic groups](#)

[See also](#)

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;
```

- DevGuide: [Building applications with Delphi](#)

Adding a string to a list

[Topic groups](#)

[See also](#)

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 }
```

- DevGuide: [Building applications with Delphi](#)

Moving a string within a list

[Topic groups](#)

[See also](#)

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)
```

- DevGuide: [Building applications with Delphi](#)

Deleting a string from a list

[Topic groups](#)

[See also](#)

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;
```

■ DevGuide: [Building applications with Delphi](#)

Copying a complete string list

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Associating objects with a string list

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Windows registry and INI files (VCL only)

[Topic groups](#)

[See also](#)

The Windows system registry is a hierarchical database where applications store configuration information. The VCL class *TRegistry* supplies methods that read and write to the registry.

Until Windows 95, most applications stored configuration information in initialization files, usually named with the extension .INI. The VCL provides the following classes to facilitate maintenance and migration of programs that use INI files:

- *TRegistry* to work with the registry (VCL only).
- *TIniFile* (VCL only) or *TMemIniFile* to work with INI files.
- *TRegistryIniFile* when you want to work with both the registry and INI files (VCL only). *TRegistryIniFile* has properties and methods similar to those of *TIniFile*, but it reads and writes to the system registry. By using a variable of type *TCustomIniFile* (the common ancestor of *TIniFile*, *TMemIniFile*, and *TRegistryIniFile*), you can write generic code that accesses either the registry or an INI file, depending on where it is called.

Only *TMemIniFile* can be used in cross-platform programming.

- DevGuide: [Building applications with Delphi](#)

Using TIniFile (VCL only)

[Topic groups](#)

[See also](#)

The INI file format is still popular, many of the Delphi configuration files (such as the DSK Desktop settings file) are in this format. Because this file format was and is prevalent, VCL provides a class to make reading and writing these files very easy. *TIniFile* is not available for cross-platform programming.

When you instantiate the *TIniFile* object, you pass as a parameter to the constructor the name of the INI file. If the file does not exist, it is automatically created. You are then free to read values using *ReadString*, *ReadInteger*, or *ReadBool*. Alternatively, if you want to read an entire section of the INI file, you can use the *ReadSection* method. Similarly, you can write values using *WriteBool*, *WriteInteger*, or *WriteString*.

Each of the Read routines takes three parameters. The first parameter identifies the section of the INI file. The second parameter identifies the value you want to read, and the third is a default value in case the section or value doesn't exist in the INI file. Similarly, the Write routines will create the section and/or value if they do not exist. The example code creates an INI file the first time it is run that looks like this:

```
[Form]
Top=185
Left=280
Caption=Default Caption
InitMax=0
```

On subsequent execution of this application, the INI values are read in during creation of the form and written back out in the *OnClose* event.

- DevGuide: [Building applications with Delphi](#)

Using TRegistry (VCL only)

[Topic groups](#)

[See also](#)

Most 32-bit applications store their information in the registry instead of INI files because the registry is hierarchical, more robust, and doesn't suffer from the size limitations of INI files. The *TRegistry* object contains methods to open, close, save, move, copy, and delete keys.

TRegistry is not available for cross-platform programming.

- DevGuide: [Building applications with Delphi](#)

Using TRegIniFile (VCL only)

[Topic groups](#)

[See also](#)

If you are accustomed to using INI files and want to move your configuration information to the registry instead, you can use the *TRegIniFile* class. *TRegIniFile* is designed to make registry entries look like INI file entries. All the methods from *TIniFile* (read and write) exist in *TRegIniFile*.

When you construct a *TRegIniFile* object, the parameter you pass (the filename for an *IniFile* object) becomes a key value under the user key in the registry, and all sections and values branch from that root. In fact, this object simplifies the registry interface considerably, so you may want to use it instead of the *TRegistry* component even if you aren't porting existing code.

TRegIniFile is not available for cross-platform programming.

- DevGuide: [Building applications with Delphi](#)

Creating drawing spaces

[Topic groups](#)

[See also](#)

The *TCanvas* encapsulates a Windows device context in the VCL and a paint device (Qt painter) in CLX. which handles all drawing for both forms, visual containers (such as panels) and the printer object (covered in “Printing”[Printing](#)). 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, fonts, etc. onto any control that contains a canvas. For example, here is a button event handler that draws a line from the upper left corner to the middle of the form and outputs some raw 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 := clBtnFace;  
    Canvas.Font.Name := 'Arial';  
    Canvas.TextOut( Canvas.PenPos.x, Canvas.PenPos.y, 'This is the end of the line' );  
end;
```

In Windows applications, the *TCanvas* object also protects you against common Windows graphics errors, such as restoring device contexts, pens, brushes, and so on to the value they had before the drawing operation. *TCanvas* is used everywhere in Delphi that drawing is required or possible, and makes drawing graphics both fail-safe and easy.

■ DevGuide: [Building applications with Delphi](#)

TPrinter

[Topic groups](#)

[See also](#)

The VCL *TPrinter* object encapsulates details of Windows printers. To get a list of installed and available printers, use the *Printers* property. The CLX *TPrinter* object is a paint device that paints on a printer. It generates postscript and sends that to lpr, lp, or another print command.

Both printer objects use a *TCanvas* (which is identical to the form's *TCanvas*) which means 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;
```

■ DevGuide: [Building applications with Delphi](#)

Using streams

[Topic groups](#)

[See also](#)

Streams are just ways of reading and writing data. Streams 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;
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.

- DevGuide: [Building applications with Delphi](#)

Common programming tasks

[Topic groups](#)

This section of the Help system discusses how to perform some of the common programming tasks in Delphi:

- [Understanding classes](#)
- [Defining classes](#)
- [Handling exceptions](#)
- [Using interfaces](#)
- [Defining custom variants](#)
- [Working with strings](#)
- [Working with files](#)
- [Converting measurements](#)
- [Defining data types](#)

- DevGuide: [Building applications with Delphi](#)

Understanding classes

[Topic groups](#)

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 Delphi 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 Delphi includes many classes in its object hierarchy, 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 [Class types](#).

■ DevGuide: [Building applications with Delphi](#)

Defining classes

[Topic groups](#)

Delphi allows you to declare classes that implement the programming features you need to use in your application. Some versions of Delphi 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: The object that holds the custom variant's data must be compiled with RTTI. This means it must be compiled using the {\$M+} compiler directive, or descend 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 Delphi 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). Delphi 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, Delphi 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  
    if fsize < > value then  
    begin  
        FSize := Value;  
        DoSomething;  
    end;  
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 [Class types](#) and [methods](#).

- DevGuide: [Building applications with Delphi](#)

Handling exceptions

[Topic groups](#)

Delphi 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 Delphi are indicated by exceptions. This topic 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](#)

- DevGuide: [Building applications with Delphi](#)

Protecting blocks of code

[Topic groups](#)

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](#)

- DevGuide: [Building applications with Delphi](#)

Responding to exceptions

[Topic groups](#)

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](#)

- DevGuide: [Building applications with Delphi](#)

Exceptions and the flow of control

[Topic groups](#)

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 }
```

■ DevGuide: [Building applications with Delphi](#)

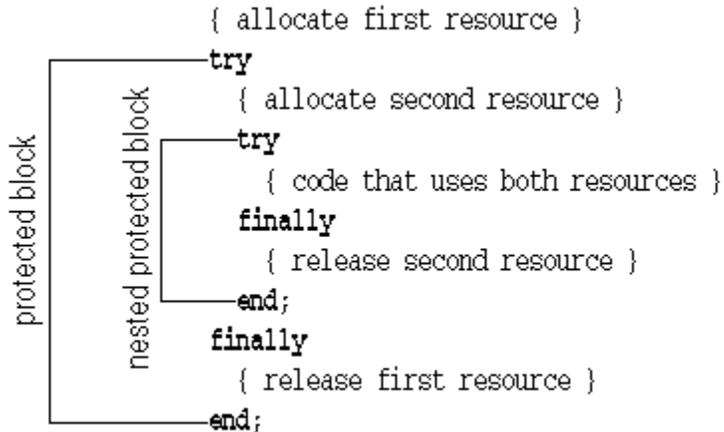
Nesting exception responses

[Topic groups](#)

Your code defines responses to exceptions that occur within blocks. Because Pascal allows you to nest blocks of code inside other blocks, 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:

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

- DevGuide: [Building applications with Delphi](#)

Protecting resource allocations

[Topic groups](#)

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](#)

- DevGuide: [Building applications with Delphi](#)

What kind of resources need protection?

[Topic groups](#)

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
- Windows resources (VCL only)
- 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.

■ DevGuide: [Building applications with Delphi](#)

Creating a resource protection block

[Topic groups](#)

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.

- DevGuide: [Building applications with Delphi](#)

Handling RTL exceptions

[Topic groups](#)

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](#)

■ DevGuide: [Building applications with Delphi](#)

What are RTL exceptions?

[Topic groups](#)

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

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

<u>Error type</u>	<u>Cause</u>	<u>Meaning</u>
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 <i>IntToStr</i> , <i>StrToInt</i> , and <i>StrToFloat</i> 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 the RTL exception types, see the code in the *SysUtils* unit.

- DevGuide: [Building applications with Delphi](#)

Creating an exception handler

[Topic groups](#)

An exception handler is code that handles a specific exception or exceptions that occur within a protected block of code. In cross-platform 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](#) and [Protecting resource allocations](#).

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.

■ DevGuide: [Building applications with Delphi](#)

Exception handling statements

[Topic groups](#)

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.

■ DevGuide: [Building applications with Delphi](#)

Using the exception instance

[Topic groups](#)

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.

- DevGuide: [Building applications with Delphi](#)

Scope of exception handlers

[Topic groups](#)

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 block (or returns to 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.

■ DevGuide: [Building applications with Delphi](#)

Providing default exception handlers

[Topic groups](#)

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](#).

- DevGuide: [Building applications with Delphi](#)

Handling classes of exceptions

[Topic groups](#)

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*.

■ DevGuide: [Building applications with Delphi](#)

Reraising the exception

[Topic groups](#)

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 display a message to the user or 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.

■ DevGuide: [Building applications with Delphi](#)

Handling component exceptions

[Topic groups](#)

Delphi'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.

■ DevGuide: [Building applications with Delphi](#)

Exception handling with external sources

[Topic groups](#)

HandleException provides default handling of exceptions for the application. Normally when developing cross-platform applications, 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 the built-in components do, 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.

■ DevGuide: [Building applications with Delphi](#)

Silent exceptions

[Topic groups](#)

Delphi 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 Delphi VCL and 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;
```

- DevGuide: [Building applications with Delphi](#)

Defining your own exceptions

[Topic groups](#)

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](#)

- DevGuide: [Building applications with Delphi](#)

Declaring an exception object type

[Topic groups](#)

Because exceptions are objects, defining a new kind of exception is as simple as declaring a new object type. Although you can raise any object instance as an exception, the standard exception handlers handle only exceptions descended from `Exception`.

As a convention, new exception types 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.

■ DevGuide: [Building applications with Delphi](#)

Raising an exception

[Topic groups](#)

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 address set 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');
```

- DevGuide: [Building applications with Delphi](#)

Using interfaces

[Topic groups](#)

Delphi's **interface** keyword allows you to create and use interfaces in your application. Interfaces are a way extending the single-inheritance model of Object Pascal 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 are also a fundamental aspect of the COM (the Component Object Model) and CORBA (Common Object Request Broker Architecture) distributed object models.

■ DevGuide: [Building applications with Delphi](#)

Interfaces as a language feature

[Topic groups](#)

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;
  procedure Cut;
  procedure Paste;
  function Undo: Boolean;
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;
  procedure Cut;
  procedure Paste;
  function Undo: Boolean;
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.

■ DevGuide: [Building applications with Delphi](#)

Implementing interfaces across the hierarchy

[Topic groups](#)

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](#) and [Memory management of interface objects](#).

■ DevGuide: [Building applications with Delphi](#)

Using interfaces with procedures

[Topic groups](#)

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](#).

- DevGuide: [Building applications with Delphi](#)

Implementing *IInterface*

[Topic groups](#)

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.

- DevGuide: [Building applications with Delphi](#)

TInterfacedObject

[Topic groups](#)

Delphi 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](#), which also discusses writing your own classes that implement interfaces but that do not follow the reference-counting mechanism inherent in *TInterfacedObject*.

■ DevGuide: [Building applications with Delphi](#)

Using the **as** operator

[Topic groups](#)

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 of 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.

- DevGuide: [Building applications with Delphi](#)

Reusing code and delegation

[Topic groups](#)

One approach to reusing code with interfaces is to have an object contain, or be contained by another. Using properties that are object types provides an approach to containment and code reuse. To support this design for interfaces, Object Pascal has a keyword **implements**, that makes it easy to write code to delegate all or part of the implementation of an interface to a subobject. Aggregation is another way of reusing code through containment and delegation. In aggregation, an outer object contains an inner object that implements interfaces which are exposed only by the outer object. The VCL and CLX have classes that support aggregation.

■ DevGuide: [Building applications with Delphi](#)

Using implements for delegation

[Topic groups](#)

Many classes 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 as a way of overriding 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](#).

- DevGuide: [Building applications with Delphi](#)

Aggregation

[Topic groups](#)

Aggregation offers a modular approach to code reuse through sub-objects that define the functionality of a containing object, but that hide the implementation details from that object. In aggregation, an outer object implements one or more interfaces. The only requirement is that it implement *Interface*. The inner object, or objects, can implement one or more interfaces, however only the outer object exposes the interfaces. These include both the interfaces it implements and the ones implemented by its contained objects. Clients know nothing about inner objects. While the outer object provides access to the inner object interfaces, their implementation is completely transparent. Therefore, the outer object class can exchange the inner object class type for any class that implements the same interface. Correspondingly, the code for the inner object classes can be shared by other classes that want to use it.

The implementation model for aggregation defines explicit rules for implementing *Interface* using delegation. The inner object must implement an *Interface* on itself, that controls the inner object's reference count. This implementation of *Interface* tracks the relationship between the outer and the inner object. For example, when an object of its type (the inner object) is created, the creation succeeds only for a requested interface of type *Interface*. The inner object also implements a second *Interface* for all the interfaces it implements. These are the interfaces exposed by the outer object. This second *Interface* delegates calls to *QueryInterface*, *AddRef*, and *Release* to the outer object. The outer *Interface* is referred to as the "controlling Unknown."

Refer to the MS online help for the rules about creating an aggregation. When writing your own aggregation classes, you can also refer to the implementation details of *Interface* in *TComObject*. *TComObject* is a COM class that supports aggregation. If you are writing COM applications, you can also use *TComObject* directly as a base class.

- DevGuide: [Building applications with Delphi](#)

Memory management of interface objects

[Topic groups](#)

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 to implement this lifetime management. `_AddRef` and `_Release` 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.

If you are creating COM objects for distributed applications (in the Windows environment only), then you should strictly adhere to the reference counting rules. However, if you are using interfaces only internally in your application, then you have a choice that depends upon the nature of your object and how you decide to use it.

■ DevGuide: [Building applications with Delphi](#)

Using reference counting

[Topic groups](#)

[See also](#)

Delphi provides most of the *Interface* memory management for you by 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 non-CoClass that provides this behavior. If you decide to use reference counting, then 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 either of 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.

■ DevGuide: [Building applications with Delphi](#)

Not using reference counting

[Topic groups](#)

[See also](#)

If your object is a 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 VCL or CLX components and interface reference counting. If you want to create a component that supports interfaces, you can implement the *Interface* *_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.

Note that, 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). You can also use aggregation. If the outer object is a component, the inner object implements reference counting as usual, by delegating to the “controlling Unknown.” It is at the level of the outer, component object that the decision is made to circumvent the *_AddRef* and *_Release* methods, and to handle memory management via the component-based approach. In fact, you can use *TInterfacedObject* as a base class for an inner object of an aggregation that has a component as its containing outer object.

Note: The “controlling Unknown” is the *IUnknown* implemented by the outer object and the one for which the reference count of the entire object is maintained. *IUnknown* is the same as *Interface*, but is used instead in COM-based applications (Windows only). For more information distinguishing the various implementations of the *IUnknown* or *Interface* interface by the inner and outer objects, see [Aggregation](#) and the Microsoft online Help topics on the “controlling Unknown.”

- DevGuide: [Building applications with Delphi](#)

Using interfaces in distributed applications (VCL only)

[Topic groups](#)

Interfaces are a fundamental element in the COM, SOAP, and CORBA distributed object models. Delphi provides base classes for these technologies that extend the basic interface functionality in *TInterfacedObject*, which simply implements the *IInterface* interface methods.

When using COM, classes and interfaces are defined in terms of *IUnknown* rather than *IInterface*. There is no semantic difference between *IUnknown* and *IInterface*, the use of *IUnknown* is simply a way to adapt Delphi interfaces to the COM definition. COM classes add functionality for using class factories and class identifiers (CLSIDs). Class factories are responsible for creating class instances via CLSIDs. The CLSIDs are used to register and manipulate COM classes. COM classes that have class factories and class identifiers are called CoClasses. CoClasses take advantage of the versioning capabilities of *QueryInterface*, so that when a software module is updated *QueryInterface* can be invoked at runtime to query the current capabilities of an object.

New versions of old interfaces, as well as any new interfaces or features of an object, can become immediately available to new clients. At the same time, objects retain complete compatibility with existing client code; no recompilation is necessary because interface implementations are hidden (while the methods and parameters remain constant). In COM applications, developers can change the implementation to improve performance, or for any internal reason, without breaking any client code that relies on that interface. For more information about COM interfaces, see [Overview of COM technologies](#).

When distributing an application using SOAP, interfaces are required to carry their own runtime type information (RTTI). The compiler only adds RTTI to an interface when it is compiled using the {\$M+} switch. Such interfaces are called *invokable interfaces*. The descendant of any invokable interface is also invokable. However, if an invokable interface descends from another interface that is not invokable, client applications can only call the methods defined in the invokable interface and its descendants. Methods inherited from the non-invokable ancestors are not compiled with type information and so can't be called by clients.

The easiest way to define invokable interfaces is to define your interface so that it descends from *IInvokable*. *IInvokable* is the same as *IInterface*, except that it is compiled using the {\$M+} switch. For more information about Web Service applications that are distributed using SOAP, and about invokable interfaces, see [Using Web Services](#).

Another distributed application technology is CORBA. The use of interfaces in CORBA applications is mediated by stub classes on the client and skeleton classes on the server. These stub and skeleton classes handle the details of marshaling interface calls so that parameter values and return values can be transmitted correctly. Applications must use either a stub or skeleton class, or employ the Dynamic Invocation Interface (DII) which converts all parameters to special variants (so that they carry their own type information).

- DevGuide: [Building applications with Delphi](#)

Defining custom variants

[Topic groups](#)

[See also](#)

One powerful built-in type of the Object Pascal language is the Variant type. Variants represent values whose type is not determined at compile time. Instead, the type of their value can change at runtime. Variants can mix with other variants and with integer, real, string, and boolean values in expressions and assignments; the compiler automatically performs type conversions.

By default, variants can't hold values that are records, sets, static arrays, files, classes, class references, or pointers. You can, however, extend the Variant type to work with any particular example of these types. All you need to do is create a descendant of the *TCustomVariantType* class that indicates how the Variant type performs standard operations.

To create a Variant type,

- 1 Map the storage of the variant's data on to the *TVarData* record.
- 2 Declare a class that descends from *TCustomVariantType*. Implement all required behavior (including type conversion rules) in the new class.
- 3 Write utility methods for creating instances of your custom variant and recognizing its type.

The above steps extend the Variant type so that the standard operators work with your new type and the new Variant type can be cast to other data types. You can further enhance your new Variant type so that it supports properties and methods that you define. When creating a Variant type that supports properties or methods, you use *TInvokeableVariantType* or *TPublishableVariantType* as a base class rather than *TCustomVariantType*.

- DevGuide: [Building applications with Delphi](#)

Storing a custom variant type's data

[Topic groups](#)

[See also](#)

Variants store their data in the *TVarData* record type. This type is a record that contains 16 bytes. The first Word indicates the type of the variant, and the remaining 14 bytes are available to store the data. While your new Variant type can work directly with a *TVarData* record, it is usually easier to define a record type whose members have names that are meaningful for your new type, and cast that new type onto the *TVarData* record type.

For example, the *VarConv* unit defines a custom variant type that represents a measurement. The data for this type includes the units (*TConvType*) of measurement, as well as the value (a double). The *VarConv* unit defines its own type to represent such a value:

```
TConvertVarData = packed record
  VType: TVarType;
  VConvType: TConvType;
  Reserved1, Reserved2: Word;
  VValue: Double;
end;
```

This type is exactly the same size as the *TVarData* record. When working with a custom variant of the new type, the variant (or its *TVarData* record) can be cast to *TConvertVarData*, and the custom Variant type simply works with the *TVarData* record as if it were a *TConvertVarData* type.

Note: When defining a record that maps onto the *TVarData* record in this way, be sure to define it as a packed record.

If your new custom Variant type needs more than 14 bytes to store its data, you can define a new record type that includes a pointer or object instance. For example, the *VarCmplx* unit uses an instance of the class *TComplexData* to represent the data in a complex-valued variant. It therefore defines a record type the same size as *TVarData* that includes a reference to a *TComplexData* object:

```
TComplexVarData = packed record
  VType: TVarType;
  Reserved1, Reserved2, Reserved3: Word;
  VComplex: TComplexData;
  Reserved4: LongInt;
end;
```

Object references are actually pointers (two Words), so this type is the same size as the *TVarData* record. As before, a complex custom variant (or its *TVarData* record), can be cast to *TComplexVarData*, and the custom variant type works with the *TVarData* record as if it were a *TComplexVarData* type.

- DevGuide: [Building applications with Delphi](#)

Creating a class to enable the custom variant type

[Topic groups](#)

[See also](#)

Custom variants work by using a special helper class that indicates how variants of the custom type can perform standard operations. You create this helper class by writing a descendant of *[TCustomVariantType](#)*. This involves overriding the appropriate virtual methods of *TCustomVariantType*.

The following topics provide details on how to implement and use a *TCustomVariantType* descendant:

- [Enabling casting](#)
- [Implementing binary operations](#)
- [Implementing comparison operations](#)
- [Implementing unary operations](#)
- [Copying and clearing custom variants](#)
- [Loading and saving custom variant values](#)
- [Using the *TCustomVariantType* descendant](#)

■ DevGuide: [Building applications with Delphi](#)

Enabling casting

[Topic groups](#)

[See also](#)

One of the most important features of the custom variant type for you to implement is typecasting. The flexibility of variants arises, in part, from their implicit typecasts.

There are two methods for you to implement that enable the custom Variant type to perform typecasts: [Cast](#), which converts another variant type to your custom variant, and [CastTo](#), which converts your custom Variant type to another type of Variant.

When implementing either of these methods, it is relatively easy to perform the logical conversions from the built-in variant types. You must consider, however, the possibility that the variant to or from which you are casting may be another custom Variant type. To handle this situation, you can try casting to one of the built-in Variant types as an intermediate step.

For example, the following *Cast* method, from the *TComplexVariantType* class uses the type Double as an intermediate type:

```
procedure TComplexVariantType.Cast(var Dest: TVarData; const Source: TVarData);
var
  LSource, LTemp: TVarData;
begin
  VarDataInit(LSource);
  try
    VarDataCopyNoInd(LSource, Source);
    if VarDataIsStr(LSource) then
      TComplexVarData(Dest).VComplex := TComplexData.Create(VarDataToStr(LSource))
    else
      begin
        VarDataInit(LTemp);
        try
          VarDataCastTo(LTemp, LSource, varDouble);
          TComplexVarData(Dest).VComplex := TComplexData.Create(LTemp.VDouble, 0);
        finally
          VarDataClear(LTemp);
        end;
      end;
    Dest.VType := VarType;
  finally
    VarDataClear(LSource);
  end;
end;
```

In addition to the use of Double as an intermediate Variant type, there are a few things to note in this implementation:

- The last step of this method sets the *VType* member of the returned *TVarData* record. This member gives the Variant type code. It is set to the [VarType](#) property of *TComplexVariantType*, which is the Variant type code assigned to the custom variant.
- The custom variant's data (*Dest*) is typecast from *TVarData* to the record type that is actually used to [store its data](#) (*TComplexVarData*). This makes the data easier to work with.
- The method makes a local copy of the source variant rather than working directly with its data. This prevents side effects that may affect the source data.

When casting from a complex variant to another type, the *CastTo* method also uses an intermediate type of Double (for any destination type other than a string):

```
procedure TComplexVariantType.CastTo(var Dest: TVarData; const Source: TVarData;
  const AVarType: TVarType);
var
  LTemp: TVarData;
begin
  if Source.VType = VarType then
    case AVarType of
      varOleStr:
        VarDataFromOleStr(Dest, TComplexVarData(Source).VComplex.AsString);
      varString:
        VarDataFromStr(Dest, TComplexVarData(Source).VComplex.AsString);
```

```
    else
      VarDataInit(LTemp);
      try
        LTemp.VType := varDouble;
        LTemp.VDouble := TComplexVarData(LTemp).VComplex.Real;
        VarDataCastTo(Dest, LTemp, AVarType);
      finally
        VarDataClear(LTemp);
      end;
    end
  end
else
  RaiseCastError;
end;
```

Note that the *CastTo* method includes a case where the source variant data does not have a type code that matches the *VarType* property. This case only occurs for empty (unassigned) source variants.

■ DevGuide: [Building applications with Delphi](#)

Implementing binary operations

[Topic groups](#)

[See also](#)

To allow the custom variant type to work with standard binary operators (+, -, *, /, div, mod, shl, shr, and, or, xor listed in the System unit), you must override the *BinaryOp* method. *BinaryOp* has three parameters: the value of the left-hand operand, the value of the right-hand operand, and the operator. Implement this method to perform the operation and return the result using the same variable that contained the left-hand operand.

For example, the following *BinaryOp* method comes from the *TComplexVariantType* defined in the *VarCmplx* unit:

```
procedure TComplexVariantType.BinaryOp(var Left: TVarData; const Right: TVarData;
  const Operator: TVarOp);
begin
  if Right.VType = VarType then
    case Left.VType of
      varString:
        case Operator of
          opAdd: Variant(Left) := Variant(Left) +
TComplexVarData(Right).VComplex.AsString;
          else
            RaiseInvalidOp;
          end;
        else
          if Left.VType = VarType then
            case Operator of
              opAdd:
                TComplexVarData(Left).VComplex.DoAdd(TComplexVarData(Right).VComplex);
              opSubtract:
                TComplexVarData(Left).VComplex.DoSubtract(TComplexVarData(Right).VComplex);
              opMultiply:
                TComplexVarData(Left).VComplex.DoMultiply(TComplexVarData(Right).VComplex);
              opDivide:
                TComplexVarData(Left).VComplex.DoDivide(TComplexVarData(Right).VComplex);
            else
              RaiseInvalidOp;
            end
          else
            RaiseInvalidOp;
          end
        end
      else
        RaiseInvalidOp;
    end;
end;
```

There are several things to note in this implementation:

This method only handles the case where the variant on the right side of the operator is a custom variant that represents a complex number. If the left-hand operand is a complex variant and the right-hand operand is not, the complex variant forces the right-hand operand first to be cast to a complex variant. It does this by overriding the *RightPromotion* method so that it always requires the type in the *VarType* property:

```
function TComplexVariantType.RightPromotion(const V: TVarData;
  const Operator: TVarOp; out RequiredVarType: TVarType): Boolean;
begin
  { Complex Op TypeX }
  RequiredVarType := VarType;
  Result := True;
end;
```

The addition operator is implemented for a string and a complex number (by casting the complex value to a string and concatenating), and the addition, subtraction, multiplication, and division operators are implemented for two complex numbers using the methods of the *TComplexData* object that is stored in the complex variant's data. This is accessed by casting the *TVarData* record to a *TComplexVarData* record and using its *VComplex* member.

Attempting any other operator or combination of types causes the method to call the *RaiseInvalidOp* method, which

causes a runtime error. The *TCustomVariantType* class includes a number of utility methods such as *RaiseInvalidOp* that can be used in the implementation of custom variant types.

BinaryOp only deals with a limited number of types: strings and other complex variants. It is possible, however, to perform operations between complex numbers and other numeric types. For the *BinaryOp* method to work, the operands must be cast to complex variants before the values are passed to this method. We have already seen (above) how to use the *RightPromotion* method to force the right-hand operand to be a complex variant if the left-hand operand is complex. A similar method, *LeftPromotion*, forces a cast of the left-hand operand when the right-hand operand is complex:

```
function TComplexVariantType.LeftPromotion(const V: TVarData;  
      const Operator: TVarOp; out RequiredVarType: TVarType): Boolean;  
begin  
  { TypeX Op Complex }  
  if (Operator = opAdd) and VarDataIsStr(V) then  
    RequiredVarType := varString  
  else  
    RequiredVarType := VarType;  
    Result := True;  
end;
```

This *LeftPromotion* method forces the left-hand operand to be cast to another complex variant, unless it is a string and the operation is addition, in which case *LeftPromotion* allows the operand to remain a string.

■ DevGuide: [Building applications with Delphi](#)

Implementing comparison operations

[Topic groups](#)

[See also](#)

There are two ways to enable a custom variant type to support comparison operators (=, <>, <, <=, >, >=). You can override the *Compare* method, or you can override the *CompareOp* method.

The *Compare* method is easiest if your custom variant type supports the full range of comparison operators. *Compare* takes three parameters: the left-hand operand, the right-hand operand, and a var Parameter that returns the relationship between the two. For example, the *TConvertVariantType* object in the *VarConv* unit implements the following *Compare* method:

```
procedure TConvertVariantType.Compare(const Left, Right: TVarData;
  var Relationship: TVarCompareResult);
const
  CRelationshipToRelationship: array [TValueRelationship] of TVarCompareResult =
    (crLessThan, crEqual, crGreaterThan);
var
  LValue: Double;
  LType: TConvType;
  LRelationship: TValueRelationship;
begin
  // supports...
  // convvar cmp number
  // Compare the value of convvar and the given number
  // convvar1 cmp convvar2
  // Compare after converting convvar2 to convvar1's unit type
  // The right can also be a string. If the string has unit info then it is
  // treated like a varConvert else it is treated as a double
  LRelationship := EqualsValue;
  case Right.VType of
    varString:
      if TryStrToConvUnit(Variant(Right), LValue, LType) then
        if LType = CIllegalConvType then
          LRelationship := CompareValue(TConvertVarData(Left).VValue, LValue)
        else
          LRelationship := ConvUnitCompareValue(TConvertVarData(Left).VValue,
            TConvertVarData(Left).VConvType,
            LValue, LType)
        else
          RaiseCastError;
      varDouble:
        LRelationship := CompareValue(TConvertVarData(Left).VValue,
          TVarData(Right).VDouble);
    else
      if Left.VType = VarType then
        LRelationship := ConvUnitCompareValue(TConvertVarData(Left).VValue,
          TConvertVarData(Left).VConvType,
          TConvertVarData(Right).VValue,
          TConvertVarData(Right).VConvType)
      else
        RaiseInvalidOp;
  end;
  Relationship := CRelationshipToRelationship[LRelationship];
end;
```

If the custom type does not support the concept of “greater than” or “less than,” only “equal” or “not equal,” however, it is difficult to implement the *Compare* method, because *Compare* must return *crLessThan*, *crEqual*, or *crGreaterThan*. When the only valid response is “not equal,” it is impossible to know whether to return *crLessThan* or *crGreaterThan*. Thus, for types that do not support the concept of ordering, you can override the *CompareOp* method instead.

CompareOp has three parameters: the value of the left-hand operand, the value of the right-hand operand, and the comparison operator. Implement this method to perform the operation and return a boolean that indicates whether the comparison is *True*. You can then call the *RaiseInvalidOp* method when the comparison makes no sense.

For example, the following *CompareOp* method comes from the *TComplexVariantType* object in the *VarCmplx* unit. It supports only a test of equality or inequality:

```
function TComplexVariantType.CompareOp(const Left, Right: TVarData;  
    const Operator: Integer): Boolean;  
begin  
    Result := False;  
    if (Left.VType = VarType) and (Right.VType = VarType) then  
        case Operator of  
            opCmpEQ:  
                Result :=  
TComplexVarData(Left).VComplex.Equal(TComplexVarData(Right).VComplex);  
            opCmpNE:  
                Result := not  
TComplexVarData(Left).VComplex.Equal(TComplexVarData(Right).VComplex);  
            else  
                RaiseInvalidOp;  
            end  
        else  
            RaiseInvalidOp;  
        end;  
end;
```

Note that the types of operands that both these implementations support are very limited. As with binary operations, you can use the *RightPromotion* and *LeftPromotion* methods to limit the cases you must consider by forcing a cast before *Compare* or *CompareOp* is called.

- DevGuide: [Building applications with Delphi](#)

Implementing unary operations

[Topic groups](#)

[See also](#)

To allow the custom variant type to work with standard unary operators (-, not), you must override the *UnaryOp* method. *UnaryOp* has two parameters: the value of the operand and the operator. Implement this method to perform the operation and return the result using the same variable that contained the operand.

For example, the following *UnaryOp* method comes from the *TComplexVariantType* defined in the *VarCmplx* unit:

```
procedure TComplexVariantType.UnaryOp(var Right: TVarData; const Operator: TVarOp);
begin
  if Right.VType = VarType then
    case Operator of
      opNegate:
        TComplexVarData(Right).VComplex.DoNegate;
      else
        RaiseInvalidOp;
    end
  else
    RaiseInvalidOp;
end;
```

Note that for the logical **not** operator, which does not make sense for complex values, this method calls *RaiseInvalidOp* to cause a runtime error.

■ DevGuide: [Building applications with Delphi](#)

Copying and clearing custom variants

[Topic groups](#)

[See also](#)

In addition to typecasting and the implementation of operators, you must indicate how to copy and clear variants of your custom Variant type.

To indicate how to copy the variant's value, implement the [Copy](#) method. Typically, this is an easy operation, although you must remember to allocate memory for any classes or structures you use to hold the variant's value:

```
procedure TComplexVariantType.Copy(var Dest: TVarData; const Source: TVarData;
  const Indirect: Boolean);
begin
  if Indirect and VarDataIsByRef(Source) then
    VarDataCopyNoInd(Dest, Source)
  else
    with TComplexVarData(Dest) do
      begin
        VType := VarType;
        VComplex := TComplexData.Create(TComplexVarData(Source).VComplex);
      end;
    end;
end;
```

Note: The *Indirect* parameter in the *Copy* method signals that the copy must take into account the case when the variant holds only an indirect reference to its data.

Tip: If your custom variant type does not allocate any memory to hold its data (if the data fits entirely in the *TVarData* record), your implementation of the *Copy* method can simply call the [SimplisticCopy](#) method.

To indicate how to clear the variant's value, implement the [Clear](#) method. As with the *Copy* method, the only tricky thing about doing this is ensuring that you free any resources allocated to store the variant's data:

```
procedure TComplexVariantType.Clear(var V: TVarData);
begin
  V.VType := varEmpty;
  FreeAndNil(TComplexVarData(V).VComplex);
end;
```

You will also need to implement the [IsClear](#) method. This way, you can detect any invalid values or special values that represent "blank" data:

```
function TComplexVariantType.IsClear(const V: TVarData): Boolean;
begin
  Result := (TComplexVarData(V).VComplex = nil) or
    TComplexVarData(V).VComplex.IsZero;
end;
```

- DevGuide: [Building applications with Delphi](#)

Loading and saving custom variant values

[Topic groups](#)

[See also](#)

By default, when the custom variant is assigned as the value of a published property, it is typecast to a string when that property is saved to a form file, and converted back from a string when the property is read from a form file. You can, however, provide your own mechanism for loading and saving custom variant values in a more natural representation. To do so, the *TCustomVariantType* descendant must implement the [IVarStreamable](#) interface from *Classes.pas*.

IVarStreamable defines two methods, *StreamIn* and *StreamOut*, for reading a variant's value from a stream and for writing the variant's value to the stream. For example, *TComplexVariantType*, in the *VarCmplx* unit, implements the *IVarStreamable* methods as follows:

```
procedure TComplexVariantType.StreamIn(var Dest: TVarData; const Stream: TStream);
begin
  with TReader.Create(Stream, 1024) do
  try
    with TComplexVarData(Dest) do
    begin
      VComplex := TComplexData.Create;
      VComplex.Real := ReadFloat;
      VComplex.Imaginary := ReadFloat;
    end;
  finally
    Free;
  end;
end;

procedure TComplexVariantType.StreamOut(const Source: TVarData; const Stream:
TStream);
begin
  with TWriter.Create(Stream, 1024) do
  try
    with TComplexVarData(Source).VComplex do
    begin
      WriteFloat(Real);
      WriteFloat(Imaginary);
    end;
  finally
    Free;
  end;
end;
```

Note how these methods create a Reader or Writer object for the *Stream* parameter to handle the details of reading or writing values.

- DevGuide: [Building applications with Delphi](#)

Using the `TCustomVariantType` descendant

[Topic groups](#)

[See also](#)

In the initialization section of the unit that defines your *TCustomVariantType* descendant, create an instance of your class. When you instantiate your object, it automatically registers itself with the variant-handling system so that the new Variant type is enabled. For example, here is the initialization section of the `VarCmplx` unit:

initialization

```
ComplexVariantType := TComplexVariantType.Create;
```

In the finalization section of the unit that defines your *TCustomVariantType* descendant, free the instance of your class. This automatically unregisters the variant type. Here is the finalization section of the `VarCmplx` unit:

finalization

```
FreeAndNil(ComplexVariantType);
```

■ DevGuide: [Building applications with Delphi](#)

Writing utilities to work with a custom variant type

[Topic groups](#)

[See also](#)

Once you have created a *TCustomVariantType* descendant to implement your custom variant type, it is possible to use the new Variant type in applications. However, without a few utilities, this is not as easy as it should be.

It is a good idea to create a method that creates an instance of your custom variant type from an appropriate value or set of values. This function or set of functions fills out [the structure you defined](#) to store your custom variant's data. For example, the following function could be used to create a complex-valued variant:

```
function VarComplexCreate(const AReal, AImaginary: Double): Variant;
begin
  VarClear(Result);
  TComplexVarData(Result).VType := ComplexVariantType.VType;
  TComplexVarData(ADest).VComplex := TComplexData.Create(ARead, AImaginary);
end;
```

This function does not actually exist in the VarCmplx unit, but is a synthesis of methods that do exist, provided to simplify the example. Note that the returned variant is cast to the record that was defined to map onto the *TVarData* structure (*TComplexVarData*), and then filled out.

Another useful utility to create is one that returns the variant type code for your new Variant type. This type code is not a constant. It is automatically generated when you instantiate your *TCustomVariantType* descendant. It is therefore useful to provide a way to easily determine the type code for your custom variant type. The following function from the VarCmplx unit illustrates how to write one, by simply returning the *VarType* property of the *TCustomVariantType* descendant:

```
function VarComplex: TVarType;
begin
  Result := ComplexVariantType.VType;
end;
```

Two other standard utilities provided for most custom variants check whether a given variant is of the custom type and cast an arbitrary variant to the new custom type. Here is the implementation of those utilities from the VarCmplx unit:

```
function VarIsComplex(const AValue: Variant): Boolean;
begin
  Result := (TVarData(AValue).VType and varTypeMask) = VarComplex;
end;
function VarAsComplex(const AValue: Variant): Variant;
begin
  if not VarIsComplex(AValue) then
    VarCast(Result, AValue, VarComplex)
  else
    Result := AValue;
end;
```

Note that these use standard features of all variants: the *VType* member of the *TVarData* record and the *VarCast* function, which works because of the [methods implemented in the *TCustomVariantType* descendant for casting data](#).

In addition to the standard utilities mentioned above, you can write any number of utilities specific to your new custom variant type. For example, the VarCmplx unit defines a large number of functions that implement mathematical operations on complex-valued variants.

- DevGuide: [Building applications with Delphi](#)

Supporting properties and methods in custom variants

[Topic groups](#)

[See also](#)

Some variants have properties and methods. For example, when the value of a variant is an interface, you can use the variant to read or write the values of properties on that interface and call its methods. Even if your custom variant type does not represent an interface, you may want to give it properties and methods that an application can use in the same way.

Using `TInvokeableVariantType`

To provide support for properties and methods, the class you create to [enable the new custom variant type](#) should descend from [TInvokeableVariantType](#) instead of directly from `TCustomVariantType`.

`TInvokeableVariantType` defines four methods:

- [DoFunction](#)
- [DoProcedure](#)
- [GetProperty](#)
- [SetProperty](#)

that you can implement to support properties and methods on your custom variant type.

For example, the `VarConv` unit uses `TInvokeableVariantType` as the base class for `TConvertVariantType` so that the resulting custom variants can support properties. The following example shows the property getter for these properties:

```
function TConvertVariantType.GetProperty(var Dest: TVarData;
  const V: TVarData; const Name: String): Boolean;
var
  LType: TConvType;
begin
  // supports...
  // 'Value'
  // 'Type'
  // 'TypeName'
  // 'Family'
  // 'FamilyName'
  // 'As[Type]'
  Result := True;
  if Name = 'VALUE' then
    Variant(Dest) := TConvertVarData(V).VValue
  else if Name = 'TYPE' then
    Variant(Dest) := TConvertVarData(V).VConvType
  else if Name = 'TYPENAME' then
    Variant(Dest) := ConvTypeToDescription(TConvertVarData(V).VConvType)
  else if Name = 'FAMILY' then
    Variant(Dest) := ConvTypeToFamily(TConvertVarData(V).VConvType)
  else if Name = 'FAMILYNAME' then
    Variant(Dest) :=
      ConvFamilyToDescription(ConvTypeToFamily(TConvertVarData(V).VConvType))
  else if System.Copy(Name, 1, 2) = 'AS' then
    begin
      if DescriptionToConvType(ConvTypeToFamily(TConvertVarData(V).VConvType),
        System.Copy(Name, 3, MaxInt), LType) then
        VarConvertCreateInto(Variant(Dest), Convert(TConvertVarData(V).VValue,
          TConvertVarData(V).VConvType, LType), LType)
      else
        Result := False;
    end
  else
    Result := False;
end;
```

The `GetProperty` method checks the `Name` parameter to determine what property is wanted. It then retrieves the information from the `TVarData` record of the `Variant (V)`, and returns it as a `Variant (Dest)`. Note that this method supports properties whose names are dynamically generated at runtime (`As[Type]`), based on the current value of the custom variant.

Similarly, the *SetProperty*, *DoFunction*, and *DoProcedure* methods are sufficiently generic that you can dynamically generate method names, or respond to variable numbers and types of parameters.

Using *TPublishableVariantType*

If the custom variant type stores its data using an object instance, then there is an easier way to implement properties, as long as they are also properties of the object that represents the variant's data. If you use *TPublishableVariantType* as the base class for your custom variant type, then you need only implement the *GetInstance* method, and all the published properties of the object that represents the variant's data are automatically implemented for the custom variants.

For example, as was seen in [Storing a custom variant type's data](#), *TComplexVariantType* stores the data of a complex-valued variant using an instance of *TComplexData*. *TComplexData* has a number of published properties (*Real*, *Imaginary*, *Radius*, *Theta*, and *FixedTheta*), that provide information about the complex value. *TComplexVariantType* descends from *TPublishableVariantType*, and implements the *GetInstance* method to return the *TComplexData* object (in `TyplInfo.pas`) that is stored in a complex-valued variant's *TVarData* record:

```
function TComplexVariantType.GetInstance(const V: TVarData): TObject;
begin
    Result := TComplexVarData(V).VComplex;
end;
```

TPublishableVariantType does the rest. It overrides the *GetProperty* and *SetProperty* methods to use the runtime type information (RTTI) of the *TComplexData* object for getting and setting property values.

Note: For *TPublishableVariantType* to work, the object that holds the custom variant's data must be compiled with RTTI. This means it must be compiled using the `{$M+}` compiler directive, or descend from *TPersistent*.

- DevGuide: [Building applications with Delphi](#)

Working with strings

[Topic groups](#)

[See also](#)

Delphi has a number of different character and string types that have been introduced throughout the development of the Object Pascal language. This section of the Help is an overview of these types, their purpose, and usage. For language details, see [String types](#).

- DevGuide: [Building applications with Delphi](#)

Character types

[Topic groups](#)

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

The *Char* character type came from standard Pascal, and was used in Turbo Pascal and then in Object Pascal. Later Object Pascal added *AnsiChar* and *WideChar* as specific character types that were used to support standards for character representation on the Windows operating system. *AnsiChar* was introduced to support an 8-bit character ANSI standard, and *WideChar* was introduced to support 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. If you use *Char* in your application, remember that its implementation is subject to change in future versions of Delphi.

Note: For cross-platform programming: 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:

<u>Type</u>	<u>Bytes</u>	<u>Contents</u>	<u>Purpose</u>
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 [Character types](#). For more information about Unicode characters, see [About extended character sets](#).

- DevGuide: [Building applications with Delphi](#)

String types

[Topic groups](#)

Delphi has three categories of types that you can use when working with strings:

- Character pointers
- String types
- String classes

This topic summarizes string types, and discusses using them with character pointers. For information about using string classes, see [TStrings](#).

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

- DevGuide: [Building applications with Delphi](#)

Short strings

[Topic groups](#)

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 Delphi 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 or in DLLs when you don't want to use the *ShareMem* unit (see also [Memory Management](#)). 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 [Short strings](#).

- DevGuide: [Building applications with Delphi](#)

Long strings

[Topic groups](#)

Long strings are dynamically allocated strings with a maximum length of 2 Gigabytes, but the practical limit is usually dependent on the amount of available memory. Like short strings, long strings use 8-bit Ansi 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 long strings, see [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 VCL 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**.

- DevGuide: [Building applications with Delphi](#)

WideString

[Topic groups](#)

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, wide strings are dynamically allocated with a maximum length of two Gigabytes, but the practical limit is usually dependent on the amount of available memory. In Delphi, wide strings are not reference-counted. Every assignment of a wide string to a wide string var creates a copy of the string data. In Kylix, WideStrings are reference counted.

The dynamically allocated memory that contains the string is deallocated when the wide string goes out of scope. In all other respects wide strings possess the same attributes as long strings. The *WideString* type is denoted by the predefined identifier *WideString*.

Since the 32-bit version of OLE (Windows only) uses Unicode for all strings, strings must be of wide string type in any OLE automated properties and method parameters. Also, most OLE API functions use null-terminated wide strings.

For more information, see [WideString](#).

- DevGuide: [Building applications with Delphi](#)

PChar types

[Topic groups](#)

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. They were created primarily as a C language and Windows API compatibility type.

- DevGuide: [Building applications with Delphi](#)

OpenString

[Topic groups](#)

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](#).

- DevGuide: [Building applications with Delphi](#)

Runtime library string handling routines

[Topic groups](#)

The runtime library provides many specialized string handling routines specific to a string type. These are routines for wide strings, long 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](#).

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.

- DevGuide: [Building applications with Delphi](#)

Wide character routines

[Topic groups](#)

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. The runtime library includes the following wide character string functions for converting between standard single-byte character strings (or MBCS strings) and Unicode strings:

- `StringToWideChar`
- `WideCharLenToString`
- `WideCharLenToStrVar`
- `WideCharToString`
- `WideCharToStrVar`

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.

A disadvantage of working with wide characters is that Windows 95 does not support wide character API function calls. Because of this, the VCL components represent all string values as single byte or MBCS strings. Translating between the wide character system and the MBCS system every time you set a string property or read its value would require tremendous amounts of extra code and slow your application down. However, you may want to translate into wide characters for some special string processing algorithms that need to take advantage of the 1:1 mapping between characters and *WideChars*.

Note: Typically, CLX components represent string values as wide strings.

■ DevGuide: [Building applications with Delphi](#)

Commonly used long string routines

[Topic groups](#)

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 following tables list these routines by these functional areas:

- Comparison
- Case conversion
- Modification
- Sub-string

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

- Uses case sensitivity: If locale settings are used, it determines the definition of case. If the routine does not use 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 lower case characters that is determined by a predefined pattern.
- Uses 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. Routines that use the Windows locale are typically prefaced with *Ansi* (that is, *AnsiXXX*).
- 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- and two-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 parse one- and two-byte characters.

ByteType and *StrByteType* determine whether a particular byte is the lead byte of a two-byte character. Be careful when using multi-byte characters not to truncate a string by cutting a two-byte 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 character or string. For more information about MBCS, see [Enabling application code](#).

String comparison routines

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

Case conversion routines

<u>Routine</u>	<u>Uses locale settings</u>	<u>Supports MBCS</u>
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 Windows locale. You should always use file names that are portable because the locale (character set) used for file names can and might differ from the default user interface.

String modification routines

<u>Routine</u>	<u>Case-sensitive</u>	<u>Supports MBCS</u>
AdjustLineBreaks	NA	yes
AnsiQuotedStr	NA	yes
StringReplace	optional by flag	yes
Trim	NA	yes
TrimLeft	NA	yes
TrimRight	NA	yes
WrapText	NA	yes

Sub-string routines

<u>Routine</u>	<u>Case-sensitive</u>	<u>Supports MBCS</u>
AnsiExtractQuotedStr	NA	yes
AnsiPos	yes	yes
IsDelimiter	yes	yes
IsPathDelimiter	yes	yes
LastDelimiter	yes	yes
QuotedStr	no	no

String handling routines

<u>Routine</u>	<u>Case-sensitive</u>	<u>Supports MBCS</u>
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

■ DevGuide: [Building applications with Delphi](#)

Declaring and initializing strings

[Topic groups](#)

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);
```

- DevGuide: [Building applications with Delphi](#)

Mixing and converting string types

[Topic groups](#)

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 CLX string types. These functions are located in *Qtypes.pas*.

- DevGuide: [Building applications with Delphi](#)

String to PChar conversions

[Topic groups](#)

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.

- DevGuide: [Building applications with Delphi](#)

String dependencies

[Topic groups](#)

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;
```

■ DevGuide: [Building applications with Delphi](#)

Returning a PChar local variable

[Topic groups](#)

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.

■ DevGuide: [Building applications with Delphi](#)

Passing a local variable as a PChar

[Topic groups](#)

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:

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

Or, for cross-platform programs, the code is nearly identical:

```
// assume FillBuffer is a predefined function
function FillBuffer(Buf:PChar;Count:Integer):Integer
begin
  . . .
end;
// assume MAX_SIZE 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. The *Length* of the string is automatically set to the right value after assigning *buf* to the string.

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;
```

■ DevGuide: [Building applications with Delphi](#)

Compiler directives for strings

[Topic groups](#)

The following compiler directives affect character and string types.

<u>Directive</u>	<u>Description</u>
{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 <i>ShortString</i> 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: <pre>var S: string[3]; procedure Test(var T: string); begin T := '1234'; end; begin Test(S); end.</pre>
{X+/-}	The {X+} compiler directive enables support for null-terminated strings by activating the special rules that apply to the built-in <i>PChar</i> type and zero-based character arrays. (These rules allow zero-based arrays and character pointers to be used with <i>Write</i> , <i>WriteLn</i> , <i>Val</i> , <i>Assign</i> , and <i>Rename</i> from the System unit.)

- DevGuide: [Building applications with Delphi](#)

Strings and characters: related topics

[Topic groups](#)

The following *Object Pascal Language Guide* topics discuss strings and character sets. Also see [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](#)

- DevGuide: [Building applications with Delphi](#)

Working with files

[Topic groups](#)

This topic describes working with files and distinguishes between manipulating files on disk, and input/output operations such as reading and writing to files. The first topic discusses the runtime library and Windows API routines you would use for common programming tasks that involve [manipulating files on disk](#). The next topic is an overview of [file types used with file I/O](#). The last topic 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 in cross-platform applications.

Note: Previous versions of the Object Pascal language performed operations on files themselves, rather than on the filename 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.

- DevGuide: [Building applications with Delphi](#)

Manipulating files

[Topic groups](#)

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. 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 in cross-platform applications.

- DevGuide: [Building applications with Delphi](#)

Deleting a file

[Topic groups](#)

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.

■ DevGuide: [Building applications with Delphi](#)

Finding a file

[Topic groups](#)

There are three routines used for finding a file: *FindFirst*, *FindNext*, and *FindClose*. *FindFirst* searches for the first instance of a filename 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 terminate 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 filename 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:

<u>Constant</u>	<u>Value</u>	<u>Description</u>
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 filename and size is displayed in the label:

```
var
  SearchRec: TSearchRec;
procedure TForm1.SearchClick(Sender: TObject);
begin
  FindFirst('c:\Program Files\delphi6\bin\*.*', faAnyFile, SearchRec);
  Labell.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in
size';
end;
procedure TForm1.AgainClick(Sender: TObject);
begin
  if (FindNext(SearchRec) = 0)
    Labell.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes
in size';
  else
    FindClose(SearchRec);
end;
```

In cross-platform applications, you should replace any hardcoded pathnames such as *c:\Program*

Files\delphi6\bin*. * with the correct pathname for the system or use environment variables (on the Environment Variables page when you choose Tools|Environment Options) to represent them.

■ DevGuide: [Building applications with Delphi](#)

Renaming a file

[Topic groups](#)

To change a filename, simply use the *RenameFile* function:

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

which changes a filename, 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.

Note: *RenameFile* in the VCL is a wrapper around the Windows API *MoveFile* function, so *MoveFile* will not work across drives either.

- DevGuide: [Building applications with Delphi](#)

File date-time routines

[Topic groups](#)

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 filename. *FileGetDate* and *FileSetDate*, however, take a *Handle* type as a parameter. To get access to a Windows file *Handle* either

- Call the Windows API *CreateFile* function. *CreateFile* is a 32-bit only function that creates or opens a file and returns a *Handle* that can be used to access the file.
- Instantiate *TFileStream* to create or open a file. Then use the *Handle* property as you would a Windows' file *Handle*. See [Using file streams](#) for more information.

- DevGuide: [Building applications with Delphi](#)

Copying a file

[Topic groups](#)

The runtime library does not provide any routines for copying a file. However, if you are writing Windows-only applications, you can directly call the Windows API *CopyFile* function to copy a file. Like most of the Delphi runtime library file routines, *CopyFile* takes a filename as a parameter, not a *Handle*. When copying a file, be aware that the file attributes for the existing file are copied to the new file, but the security attributes are not. *CopyFile* is also useful when moving files across drives because neither the Delphi *RenameFile* function nor the Windows API *MoveFile* function can rename/move files across drives.

■ DevGuide: [Building applications with Delphi](#)

File types with file I/O

[Topic groups](#)

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.

<u>File type</u>	<u>Description</u>
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 file-handling routines, such as <i>AssignPrn</i> and <i>writeln</i> , use them. These file types are obsolete and are incompatible with Windows file handles. If you need to work with them, see Untyped files and File types .
File handles	<p>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 (for example, using <i>FileOpen</i> or <i>FileCreate</i>). 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).</p> <p>In Windows programming, the Object Pascal file handles are wrappers for the Windows file handle type. The runtime library file-handling routines that use Windows file Handles are typically wrappers around Windows API functions. For example, the <i>FileRead</i> calls the Windows <i>ReadFile</i> function. Because the Delphi functions use Object Pascal syntax, and occasionally provide default parameter values, they are a convenient interface to the Windows API. Using these routines is straightforward, and if you are familiar and comfortable with the Windows API file routines, you may want to use them when working with file I/O.</p>
File streams	File streams are object instances of the <i>TFileStream</i> class used to access information in disk files. File streams are a portable and high-level approach to file I/O. <i>TFileStream</i> has a <i>Handle</i> property that lets you access the file handle. The Using file streams discusses <i>TFileStream</i> .

- DevGuide: [Building applications with Delphi](#)

Using file streams

[Topic groups](#)

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 *TFilter* classes, *TReader*, and *TWriter*, to stream objects out to disk. Therefore, when you have a file stream, you can use that same code for the component streaming mechanism. For more information about using the component streaming system, see [TStream](#), [TFilter](#), [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 following topics focus on this aspect of file streams:

- [Creating and opening files](#)

- [Using the file handle](#)

- [Reading and writing to files](#)

- [Reading and writing strings](#)

- [File position and size](#)

- [Seeking a file](#)

- [Copying](#)

■ DevGuide: [Building applications with Delphi](#)

Creating and opening files

[Topic groups](#)

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 1.11 **Open modes**

<u>Value</u>	<u>Meaning</u>
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 1.12 **Share modes**

<u>Value</u>	<u>Meaning</u>
fmShareCompat	Sharing is compatible with the way FCBs are opened.
fmShareExclusive	Other applications can not open the file for any reason.
fmShareDenyWrite	Other applications can open the file for reading but not for writing.
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.

<u>Open Mode</u>	<u>fmShareCompat</u>	<u>fmShareExclusive</u>	<u>fmShareDenyWrite</u>	<u>fmShareDenyRead</u>	<u>fmShareDenyNone</u>
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.

- DevGuide: [Building applications with Delphi](#)

Using the file handle

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Reading and writing to files

[Topic groups](#)

[See also](#)

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.

■ DevGuide: [Building applications with Delphi](#)

Reading and writing strings

[Topic groups](#)

[See also](#)

If you are passing a string to a read or write function, you need to be aware of 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;
```

- DevGuide: [Building applications with Delphi](#)

Seeking a file

[Topic groups](#)

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:

<u>Value</u>	<u>Meaning</u>
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.

- DevGuide: [Building applications with Delphi](#)

File position and size

[Topic groups](#)

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 that was opened in *fmOpenRead* mode raises an exception.

- DevGuide: [Building applications with Delphi](#)

Copying

[Topic groups](#)

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*.

- DevGuide: [Building applications with Delphi](#)

Converting measurements

[Topic groups](#)

The ConvUtils unit declares a general-purpose [Convert function](#) that you can use to convert a measurement from one set of units to another. You can perform conversions between compatible units of measurement such as feet and inches or days and weeks. Units that measure the same types of things are said to be in the same *conversion family*. The units you're converting must be in the same conversion family. For information on doing conversions, see [Performing conversions](#).

The StdConvs unit defines several conversion families and measurement units within each family. In addition, you can create customized conversion families and associated units using the [RegisterConversionType](#) and [RegisterConversionFamily](#) functions. For information on extending conversion and conversion units, see [Adding new measurement types](#).

- DevGuide: [Building applications with Delphi](#)

Performing conversions

[Topic groups](#)

[See also](#)

You can use the *Convert* function to perform both simple and complex conversions. It includes a simple syntax and a second syntax for performing conversions between complex measurement types.

Performing simple conversions

You can use the *Convert* function to convert a measurement from one set of units to another. The *Convert* function converts between units that measure the same type of thing (distance, area, time, temperature, and so on).

To use *Convert*, you must specify the units from which to convert and to which to convert. You use the *TConvType* type to identify the units of measurement.

For example, this converts a temperature from degrees Fahrenheit to degrees Kelvin:

```
TempInKelvin := Convert(StrToFloat(Edit1.Text), tuFahrenheit, tuKelvin);
```

Performing complex conversions

You can also use the *Convert* function to perform more complex conversions between the ratio of two measurement types. Examples of when you might need to use this are when converting miles per hour to meters per minute for calculating speed or when converting gallons per minute to liters per hour for calculating flow.

For example, the following call converts miles per gallon to kilometers per liter:

```
nKPL := Convert(StrToFloat(Edit1.Text), duMiles, vuGallons, duKilometers, vuLiter);
```

The units you're converting must be in the same conversion family (they must measure the same thing). If the units are not compatible, *Convert* raises an *EConversionError* exception. You can check whether two *TConvType* values are in the same conversion family by calling [CompatibleConversionTypes](#).

The *StdConvs* unit defines several families of *TConvType* values. See [Conversion family variables](#) for a list of the predefined families of measurement units and the measurement units in each family.

- DevGuide: [Building applications with Delphi](#)

Adding new measurement types

[Topic groups](#)

[See also](#)

If you want to perform conversions between measurement units not already defined in the StdConvs unit, you need to create a new conversion family to represent the measurement units (*TConvType* values). When two *TConvType* values are registered with the same conversion family, the *Convert* function can convert between measurements made using the units represented by those *TConvType* values.

You first need to obtain *TConvFamily* values by registering a conversion family using the [RegisterConversionFamily](#) function. After you get a *TConvFamily* value (by registering a new conversion family or using one of the global variables in the StdConvs unit), you can use the [RegisterConversionType](#) function to add the new units to the conversion family. The following examples show how to do this:

[Creating a simple conversion family and adding units](#)

[Using a conversion function](#)

[Using a class to manage conversions](#)

For more examples, refer to the source code for the standard conversions unit (stdconvs.pas). (Note that the source is not included in all versions of Delphi.)

- DevGuide: [Building applications with Delphi](#)

Creating a simple conversion family and adding units

[Topic groups](#)

[See also](#)

One example of when you could create a new conversion family and add new measurement types might be when performing conversions between long periods of time (such as months to centuries) where a loss of precision can occur.

To explain this further, the *cbTime* family uses a day as its base unit. The base unit is the one that is used when performing all conversions within that family. Therefore, all conversions must be done in terms of days. An inaccuracy can occur when performing conversions using units of months or larger (months, years, decades, centuries, millennia) because there is not an exact conversion between days and months, days and years, and so on. Months have different lengths; years have correction factors for leap years, leap seconds, and so on.

If you are only using units of measurement greater than or equal to months, you can create a more accurate conversion family with years as its base unit. This example creates a new conversion family called *cbLongTime*.

Declare variables

First, you need to declare variables for the identifiers. The identifiers are used in the new LongTime conversion family, and the units of measurement that are its members:

```
var
  cbLongTime: TConvFamily;
  ltMonths: TConvType;
  ltYears: TConvType;
  ltDecades: TConvType;
  ltCenturies: TConvType;
  ltMillennia: TConvType;
```

Register the conversion family

Next, register the conversion family:

```
cbLongTime := RegisterConversionFamily ('Long Times');
```

Although an *UnregisterConversionFamily* procedure is provided, you don't need to unregister conversion families unless the unit that defines them is removed at runtime. They are automatically cleaned up when your application shuts down.

Register measurement units

Next, you need to register the measurement units within the conversion family that you just created. You use the *RegisterConversionType* function, which registers units of measurement within a specified family. You need to define the base unit which in the example is years, and the other units are defined using a factor that indicates their relation to the base unit. So, the factor for *ltMonths* is 1/12 because the base unit for the LongTime family is years. You also include a description of the units to which you are converting.

The code to register the measurement units is shown here:

```
ltMonths:=RegisterConversionType (cbLongTime, 'Months', 1/12);
ltYears:=RegisterConversionType (cbLongTime, 'Years', 1);
ltDecades:=RegisterConversionType (cbLongTime, 'Decades', 10);
ltCenturies:=RegisterConversionType (cbLongTime, 'Centuries', 100);
ltMillennia:=RegisterConversionType (cbLongTime, 'Millennia', 1000);
```

Use the new units

You can now use the newly registered units to perform conversions. The global *Convert* function can convert between any of the conversion types that you registered with the *cbLongTime* conversion family.

So instead of using the following *Convert* call,

```
Convert (StrToFloat (Edit1.Text), tuMonths, tuMillennia);
```

you can now use this one for greater accuracy:

```
Convert (StrToFloat (Edit1.Text), ltMonths, ltMillennia);
```

■ DevGuide: [Building applications with Delphi](#)

Using a conversion function

[Topic groups](#)

[See also](#)

For cases when the conversion is more complex, you can use a different syntax to specify a function to perform the conversion instead of using a conversion factor. For example, you can't convert temperature values using a conversion factor, because different temperature scales have a different origins.

This example, which comes from the `StdConvs` unit, shows how to register a conversion type by providing functions to convert to and from the base units.

Declare variables

First, declare variables for the identifiers. The identifiers are used in the `cbTemperature` conversion family, and the units of measurement are its members:

```
var
  cbTemperature: TConvFamily;
  tuCelsius: TConvType;
  tuKelvin: TConvType;
  tuFahrenheit: TConvType;
```

Note: The units of measurement listed here are a subset of the temperature units actually registered in the `StdConvs` unit.

Register the conversion family

Next, register the conversion family:

```
cbTemperature := RegisterConversionFamily ('Temperature');
```

Register the base unit

Next, define and register the base unit of the conversion family, which in the example is degrees Celsius. Note that in the case of the base unit, we can use a simple conversion factor, because there is no actual conversion to make:

```
tuCelsius:=RegisterConversionType(cbTemperature, 'Celsius', 1);
```

Write methods to convert to and from the base unit

You need to write the code that performs the conversion from each temperature scale to and from degrees Celsius, because these do not rely on a simple conversion factor. These functions are taken from the `StdConvs` unit:

```
function FahrenheitToCelsius(const AValue: Double): Double;
begin
  Result := ((AValue - 32) * 5) / 9;
end;
function CelsiusToFahrenheit(const AValue: Double): Double;
begin
  Result := ((AValue * 9) / 5) + 32;
end;
function KelvinToCelsius(const AValue: Double): Double;
begin
  Result := AValue - 273.15;
end;
function CelsiusToKelvin(const AValue: Double): Double;
begin
  Result := AValue + 273.15;
end;
```

Register the other units

Now that you have the conversion functions, you can register the other measurement units within the conversion family. You also include a description of the units.

The code to register the other units in the family is shown here:

```
tuKelvin := RegisterConversionType(cbTemperature, 'Kelvin', KelvinToCelsius,
  CelsiusToKelvin);
tuFahrenheit := RegisterConversionType(cbTemperature, 'Fahrenheit',
  FahrenheitToCelsius, CelsiusToFahrenheit);
```

Use the new units

You can now use the newly registered units to perform conversions in your applications. The global *Convert* function can convert between any of the conversion types that you registered with the *cbTemperature* conversion family. For example the following code converts a value from degrees Fahrenheit to degrees Kelvin.

```
Convert(StrToFloat(Edit1.Text), tuFahrenheit, tuKelvin);
```

- DevGuide: [Building applications with Delphi](#)

Using a class to manage conversions

[Topic groups](#)

[See also](#)

You can always use conversion functions to register a conversion unit. There are times, however, when this requires you to create an unnecessarily large number of functions that all do essentially the same thing.

If you can write a set of conversion functions that differ only in the value of a parameter or variable, you can create a class to handle those conversions. For example, there is a set standard techniques for converting between the various European currencies since the introduction of the Euro. Even though the conversion factors remain constant (unlike the conversion factor between, say, dollars and Euros), you can't use a simple conversion factor approach to properly convert between European currencies for two reasons:

- The conversion must round to a currency-specific number of digits.
- The conversion factor approach uses an inverse factor to the one specified by the standard Euro conversions.

However, this can all be handled by the conversion functions such as the following:

```
function FromEuro(const AValue: Double, Factor, FRound): Double;
begin
    Result := RoundTo(AValue * Factor, FRound);
end;
function ToEuro(const AValue: Double, Factor): Double;
begin
    Result := AValue / Factor;
end;
```

The problem is, this approach requires extra parameters on the conversion function, which means you can't simply register the same function with every European currency. In order to avoid having to write two new conversion functions for every European currency, you can make use of the same two functions by making them the members of a class.

Creating the conversion class

The class must be a descendant of *TConvTypeFactor*. *TConvTypeFactor* defines two methods, *ToCommon* and *FromCommon*, for converting to and from the base units of a conversion family (in this case, to and from Euros). Just as with the functions you use directly when registering a conversion unit, these methods have no extra parameters, so you must supply the number of digits to round off and the conversion factor as private members of your conversion class. This is shown in the EuroConv example in the demos\ConvertIt directory (see euroconv.pas):

```
type
    TConvTypeEuroFactor = class(TConvTypeFactor)
    private
        FRound: TRoundToRange;
    public
        constructor Create(const AConvFamily: TConvFamily;
            const ADescription: string; const AFactor: Double;
            const ARound: TRoundToRange);
        function ToCommon(const AValue: Double): Double; override;
        function FromCommon(const AValue: Double): Double; override;
    end;
end;
```

The constructor assigns values to those private members:

```
constructor TConvTypeEuroFactor.Create(const AConvFamily: TConvFamily;
    const ADescription: string; const AFactor: Double;
    const ARound: TRoundToRange);
begin
    inherited Create(AConvFamily, ADescription, AFactor);
    FRound := ARound;
end;
```

The two conversion functions simply use these private members:

```
function TConvTypeEuroFactor.FromCommon(const AValue: Double): Double;
begin
    Result := SimpleRoundTo(AValue * Factor, FRound);
end;
function TConvTypeEuroFactor.ToCommon(const AValue: Double): Double;
```

```

begin
    Result := AValue / Factor;
end;

```

Declare variables

Now that you have a conversion class, begin as with any other conversion family, by declaring identifiers:

```

var
    euEUR: TConvType; { EU euro }
    euBEF: TConvType; { Belgian francs }
    euDEM: TConvType; { German marks }
    euGRD: TConvType; { Greek drachmas }
    euESP: TConvType; { Spanish pesetas }
    euFFR: TConvType; { French francs }
    euIEP: TConvType; { Irish pounds }
    euITL: TConvType; { Italian lire }
    euLUF: TConvType; { Luxembourg francs }
    euNLG: TConvType; { Dutch guilders }
    euATS: TConvType; { Austrian schillings }
    euPTE: TConvType; { Portuguese escudos }
    euFIM: TConvType; { Finnish marks }
    euUSD: TConvType; { US dollars }
    euGBP: TConvType; { British pounds }
    euJPY: TConvType; { Japanese yen }

```

Register the conversion family and the other units

Now you are ready to register the conversion family and the European monetary units, using your new conversion class:

```

cbEuro := RegisterConversionFamily ('European currency');
...
// Euro's various conversion types
euEUR := RegisterEuroConversionType(cbEuro, SEURDescription, EURToEUR, EURSubUnit);
euBEF := RegisterEuroConversionType(cbEuro, SBEFDescription, BEFToEUR, BEFSubUnit);
euDEM := RegisterEuroConversionType(cbEuro, SDEMDescription, DEMToEUR, DEMSubUnit);
euGRD := RegisterEuroConversionType(cbEuro, SGRDDescription, GRDToEUR, GRDSubUnit);
euESP := RegisterEuroConversionType(cbEuro, SESPDescription, ESPToEUR, ESPSubUnit);
euFFR := RegisterEuroConversionType(cbEuro, SFFRDescription, FFRTToEUR, FFRSubUnit);
euIEP := RegisterEuroConversionType(cbEuro, SIEPDescription, IEPTToEUR, IEPSubUnit);
euITL := RegisterEuroConversionType(cbEuro, SITLDescription, ITLToEUR, ITLSubUnit);
euLUF := RegisterEuroConversionType(cbEuro, SLUFDescription, LUFTToEUR, LUFSubUnit);
euNLG := RegisterEuroConversionType(cbEuro, SNLGDescription, NLGToEUR, NLGSubUnit);
euATS := RegisterEuroConversionType(cbEuro, SATSDescription, ATSTToEUR, ATSSubUnit);
euPTE := RegisterEuroConversionType(cbEuro, SPTEDescription, PTETToEUR, PTESubUnit);
euFIM := RegisterEuroConversionType(cbEuro, SFIMDescription, FIMToEUR, FIMSubUnit);
euUSD := RegisterEuroConversionType(cbEuro, SUSDDescription,
    ConvertUSDToEUR, ConvertEURToUSD);
euGBP := RegisterEuroConversionType(cbEuro, SGBPDescription,
    ConvertGBPToEUR, ConvertEURToGBP);
euJPY := RegisterEuroConversionType(cbEuro, SJPYDescription,
    ConvertJPYToEUR, ConvertEURToJPY);

```

Note that *RegisterEuroConversionType* is a wrapper function that simplifies the registering of the monetary types. See the example code for details.

Use the new units

You can now use the newly registered units to perform conversions in your applications. The global *Convert* function can convert between any of the European currencies you have registered with the new *cbEuro* family. For example, the following code converts a value from Italian Lire to German Marks:

```

Edit2.Text = FloatToStr(Convert(StrToFloat(Edit1.Text), euITL, euDEM));

```

- DevGuide: [Building applications with Delphi](#)

Defining data types

[Topic groups](#)

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 [About types](#). The syntax for declaring new types is described in [Declaring types](#).

- DevGuide: [Building applications with Delphi](#)

Creating applications

[Topic groups](#)

[See also](#)

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

- [GUI applications](#)
- [Console applications](#)
- [Service applications \(for Windows applications only\)](#)
- [Packages and DLLs](#)

GUI applications generally have an easy-to-use interface. Console applications run from a console window. Service applications are run as Windows services. These types of applications compile as executables with start-up code.

You can create other types of projects such as packages and DLLs that result in creating packages or dynamically linkable libraries. These applications produce executable code without start-up code. Refer to [Creating packages and DLLs](#).

- DevGuide: [Building applications with Delphi](#)

GUI applications

[Topic groups](#)

[See also](#)

A graphical user interface (GUI) application is one that is designed using graphical features such as windows, menus, dialog boxes, and features that make the application easy to use. When you compile a GUI application, an executable file with start-up code 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 DLLs, packages, and other support files from the executable.

Delphi 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.

- DevGuide: [Building applications with Delphi](#)

User interface models

[Topic groups](#)

[See also](#)

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 [Developing the application user interface](#).

- DevGuide: [Building applications with Delphi](#)

SDI applications

[Topic groups](#)

[See also](#)

To create a new SDI application,

- 1 Select File|New|Other 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 Delphi assumes that all new applications are SDI applications.

- DevGuide: [Building applications with Delphi](#)

MDI applications

[Topic groups](#)

[See also](#)

To create a new MDI application,

- 1 Select File|New|Other 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.

- DevGuide: [Building applications with Delphi](#)

Setting IDE, project, and compilation options

[Topic groups](#)

[See also](#)

Choose [Project|Options](#) to specify various options for your project.

Setting default project options

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 selected by default.

■ DevGuide: [Building applications with Delphi](#)

Programming templates

[Topic groups](#)

Programming templates are commonly used “skeleton” structures that you can add to your source code and then fill in. Some standard code templates such as those for array, class, and function declarations, and many statements, are included with Delphi.

You can also write your own templates for coding structures that you often use. 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 Type a name for the template after Shortcut name and enter a brief description of the new template.
- 5 Add the template code to the Code text box.
- 6 Click OK.

■ DevGuide: [Building applications with Delphi](#)

Console applications

[Topic groups](#)

[See also](#)

Console applications are 32-bit programs that run without a graphical interface, usually in a console window. 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|Other and select Console Application from the New Items dialog box.

Delphi 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.

Console applications should handle all exceptions to prevent windows from displaying a dialog during its execution. For example, your application should include exception handling such as shown in the following code:

```
program Project1;
{$APPTYPE CONSOLE}
begin
try
  raise exception.create('hi');
except
  WriteLn('exception occurred');
end;
end.
```

■ DevGuide: [Building applications with Delphi](#)

Service applications

[Topic groups](#)

[See also](#)

Service applications take requests from client applications, process those requests, and return information to the client applications. They typically run in the background, without much user input. A web, FTP, or e-mail server is an example of a service application.

To create an application that implements a Win32 service, Choose File|New, and select Service Application from the New Items page. This adds a global variable named *Application* to your project, which is of type *TServiceApplication*.

Once you have created a service application, you will see a window in the designer that corresponds to a service (*TService*). Implement the service by setting its properties and event handlers in the Object Inspector. You can add additional services to your service application by choosing Service from the new items dialog. Do not add services to an application that is not a service application. While a *TService* object can be added, the application will not generate the requisite events or make the appropriate Windows calls on behalf of the service.

Once your service application is built, you can install its services with the Service Control Manager (SCM). Other applications can then launch your services by sending requests to the SCM.

To install your application's services, run it using the /INSTALL option. The application installs its services and exits, giving a confirmation message if the services are successfully installed. You can suppress the confirmation message by running the service application using the /SILENT option.

To uninstall the services, run it from the command line using the /UNINSTALL option. (You can also use the /SILENT option to suppress the confirmation message when uninstalling).

Example: This service has a *TServerSocket* whose port is set to 80. This is the default port for Web Browsers to make requests to Web Servers and for Web Servers to make responses to Web Browsers. This particular example produces a text document in the C:\Temp directory called WebLogxxx.log (where xxx is the ThreadID). There should be only one Server listening on any given port, so if you have a web server, you should make sure that it is not listening (the service is stopped).

To see the results: open up a web browser on the local machine and for the address, type 'localhost' (with no quotes). The Browser will time out eventually, but you should now have a file called weblogxxx.log in the C:\temp directory.

- 1 To create the example, choose File|New and select Service Application from the New Items dialog. You will see a window appear named Service1. From the Internet page of the Component palette, add a ServerSocket component to the service window (Service1).
- 2 Next, add a private data member of type *TMemoryStream* to the TService1 class. The interface section of your unit should now look like this:

```
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, SvcMgr, Dialogs,
  ScktComp;
type
  TService1 = class(TService)
    ServerSocket1: TServerSocket;
    procedure ServerSocket1ClientRead(Sender: TObject;
      Socket: TCustomWinSocket);
    procedure Service1Execute(Sender: TService);
  private
    { Private declarations }
    Stream: TMemoryStream; // Add this line here
  public
    function GetServiceController: PServiceController; override;
    { Public declarations }
  end;
var
  Service1: TService1;
```

- 3 Next, select ServerSocket1, the component you added in step 1. In the Object Inspector, double click the *OnClientRead* event and add the following event handler:

```
procedure TService1.ServerSocket1ClientRead(Sender: TObject;
  Socket: TCustomWinSocket);
var
```

```

    Buffer: PChar;
begin
    Buffer := nil;
while Socket.ReceiveLength > 0 do begin
    Buffer := AllocMem(Socket.ReceiveLength);
    try
        Socket.ReceiveBuf(Buffer^, Socket.ReceiveLength);
        Stream.Write(Buffer^, StrLen(Buffer));
    finally
        FreeMem(Buffer);
    end;
    Stream.Seek(0, soFromBeginning);
    Stream.SaveToFile('c:\Temp\Weblog' + IntToStr(ServiceThread.ThreadID) + '.log');
end;
end;
4 Finally, select Service1 by clicking in the window's client area (but not on the ServiceSocket). In the Object Inspector, double click the OnExecute event and add the following event handler:
procedure TService1.Service1Execute(Sender: TService);
begin
    Stream := TMemoryStream.Create;
    try
        ServerSocket1.Port := 80; // WWW port
        ServerSocket1.Active := True;
        while not Terminated do begin
            ServiceThread.ProcessRequests(True);
        end;
        ServerSocket1.Active := False;
    finally
        Stream.Free;
    end;
end;

```

When writing your service application, you should be aware of:

- [Service threads](#)
- [Service name properties](#)
- [Debugging service applications](#)

■ DevGuide: [Building applications with Delphi](#)

Service threads

[Topic groups](#)

[See also](#)

Each service has its own thread (*TServiceThread*), so if your service application implements more than one service you must ensure that the implementation of your services is thread-safe. *TServiceThread* is designed so that you can implement the service in the *TService.OnExecute* event handler. The service thread has its own *Execute* method which contains a loop that calls the service's *OnStart* and *OnExecute* handlers before processing new requests.

Because service requests can take a long time to process and the service application can receive simultaneous requests from more than one client, it is more efficient to spawn a new thread (derived from *TThread*, not *TServiceThread*) for each request and move the implementation of that service to the new thread's *Execute* method. This allows the service thread's *Execute* loop to process new requests continually without having to wait for the service's *OnExecute* handler to finish. The following example demonstrates.

Example: This service beeps every 500 milliseconds from within the standard thread. It handles pausing, continuing, and stopping of the thread when the service is told to pause, continue, or stop.

- 1 Choose File|New|Other and select Service Application from the New Items dialog. You will see a window appear named Service1.
- 2 In the interface section of your unit, declare a new descendant of *TThread* named *TSparkyThread*. This is the thread that does the work for your service. The declaration should appear as follows:

```
TSparkyThread = class(TThread)
    public
        procedure Execute; override;
    end;
```

- 3 Next, in the implementation section of your unit, create a global variable for a *TSparkyThread* instance:

```
var
    SparkyThread: TSparkyThread;
```

- 4 Add the following code to the implementation section for the *TSparkyThread* *Execute* method (the thread function):

```
procedure TSparkyThread.Execute;
begin
    while not Terminated do
    begin
        Beep;
        Sleep(500);
    end;
end;
```

- 5 Select the Service window (Service1), and double-click the *OnStart* event in the Object Inspector. Add the following *OnStart* event handler:

```
procedure TService1.Service1Start(Sender: TService; var Started: Boolean);
begin
    SparkyThread := TSparkyThread.Create(False);
    Started := True;
end;
```

- 6 Double-click the *OnContinue* event in the Object Inspector. Add the following *OnContinue* event handler:

```
procedure TService1.Service1Continue(Sender: TService; var Continued: Boolean);
begin
    SparkyThread.Resume;
    Continued := True;
end;
```

- 7 Double-click the *OnPause* event in the Object Inspector. Add the following *OnPause* event handler:

```
procedure TService1.Service1Pause(Sender: TService; var Paused: Boolean);
begin
    SparkyThread.Suspend;
    Paused := True;
end;
```

- 8 Finally, double-click the *OnStop* event in the Object Inspector and add the following *OnStop* event handler:

```
procedure TService1.Service1Stop(Sender: TService; var Stopped: Boolean);
begin
    SparkyThread.Terminate;
```

```
    Stopped := True;  
end;
```

When developing server applications, choosing to spawn a new thread depends on the nature of the service being provided, the anticipated number of connections, and the expected number of processors on the computer running the service.

- DevGuide: [Building applications with Delphi](#)

Service name properties (Windows only)

[Topic groups](#)

[See also](#)

The VCL provides classes for creating service applications on the Windows platform (not available for cross-platform applications). These include *TService* and *TDependency*. When using these classes, the various name properties can be confusing. This section describes the differences.

Services have user names (called Service start names) that are associated with passwords, display names for display in manager and editor windows, and actual names (the name of the service). Dependencies can be services or they can be load ordering groups. They also have names and display names. And because service objects are derived from *TComponent*, they inherit the *Name* property. The following sections summarize the name properties:

TDependency properties

The *TDependency DisplayName* is both a display name and the actual name of the service. It is nearly always the same as the *TDependency Name* property.

TService name properties

The *TService Name* property is inherited from *TComponent*. It is the name of the component, and is also the name of the service. For dependencies that are services, this property is the same as the *TDependency Name* and *DisplayName* properties.

TService's DisplayName is the name displayed in the Service Manager window. This often differs from the actual service name (*TService.Name*, *TDependency.DisplayName*, *TDependency.Name*). Note that the *DisplayName* for the Dependency and the *DisplayName* for the Service usually differ.

Service start names are distinct from both the service display names and the actual service names. A *ServiceStartName* is the user name input on the Start dialog selected from the Service Control Manager.

- DevGuide: [Building applications with Delphi](#)

Debugging service applications

[Topic groups](#)

[See also](#)

You can debug service applications by attaching to the service application process when it is already running (that is, by starting the service first, and then attaching to the debugger). To attach to the service application process, choose Run|Attach To Process, and select the service application in the resulting dialog.

In some cases, this approach may fail, due to insufficient rights. If that happens, you can use the Service Control Manager to enable your service to work with the debugger:

- 1 First create a key called **Image File Execution Options** in the following registry location:
`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion`
- 2 Create a subkey with the same name as your service (for example, MYSERV.EXE). To this subkey, add a value of type REG_SZ, named Debugger. Use the full path to Delphi32.exe as the string value.
- 3 In the Services control panel applet, select your service, click Startup and check Allow Service to Interact with Desktop.

On Windows NT systems, you can use another approach for debugging service applications. However, this approach can be tricky, because it requires short time intervals:

- 1 First, launch the application in the debugger. Wait a few seconds until it has finished loading.
- 2 Quickly start the service from the control panel or from the command line:

```
start MyServ
```

You must launch the service quickly (within 15-30 seconds of application startup) because the application will terminate if no service is launched.

- DevGuide: [Building applications with Delphi](#)

Creating packages and DLLs

[Topic groups](#)

[See also](#)

Dynamic link libraries (DLLs) are modules of compiled code that work in conjunction with an executable to provide functionality to an application. You can create DLLs in cross-platform programs. However, on Linux, DLLs (and packages) recompile as shared objects.

Packages are special DLLs used by Delphi 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.

DLLs and libraries should handle all exceptions to prevent the display of errors and warnings through Windows dialogs.

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

Compiler Directive	Description
<code>{\$LIBPREFIX 'string'}</code>	Adds a specified prefix to the output file name. For example, you could specify <code>{\$LIBPREFIX 'dcl'}</code> for a design-time package, or use <code>{\$LIBPREFIX ''}</code> to eliminate the prefix entirely.
<code>{\$LIBSUFFIX 'string'}</code>	Adds a specified suffix to the output file name before the extension. For example, use <code>{\$LIBSUFFIX '-2.1.3'}</code> in <code>something.pas</code> to generate <code>something-2.1.3.bpl</code> .
<code>{\$LIBVERSION 'string'}</code>	Adds a second extension to the output file name after the <code>.bpl</code> extension. For example, use <code>{\$LIBVERSION '2.1.3'}</code> in <code>something.pas</code> to generate <code>something.bpl.2.1.3</code> .

For more information on packages, see [Working with packages and components](#).

- DevGuide: [Building applications with Delphi](#)

When to use packages and DLLs

[Topic groups](#)

For most applications written in Delphi, packages provide greater flexibility and are easier to create than DLLs. However, there are several situations where DLLs would be better suited to your projects than packages:

- Your code module will be called from non-Delphi applications.
- You are extending the functionality of a web server.
- You are creating a code module to be used by third-party developers.
- Your project is an OLE container.

You cannot pass runtime type information (RTTI) across DLLs or from a DLL to an executable. That's because DLLs all maintain their own symbol information. If you need to pass a *TStrings* object from a DLL then using an **is** or **as** operator, you need to create a package rather than a DLL. Packages share symbol information.

■ DevGuide: [Building applications with Delphi](#)

Writing database applications

[Topic groups](#)

One of Delphi's strengths is its support for creating advanced database applications. Delphi supports tools that allow you to connect to SQL servers and databases such as Oracle, Sybase, InterBase, MySQL, MS-SQL, Informix, and DB2 while providing transparent data sharing between applications.

Delphi includes many components for accessing databases and representing the information they contain. On the Component palette, the database components are grouped according to the data access mechanism and function.

<u>Palette page</u>	<u>Contents</u>
BDE	Components that use the Borland Database Engine (BDE), a large API for interacting with databases. The BDE supports the broadest range of functions and comes with the most supporting utilities including Database Desktop, Database Explorer, SQL Monitor, and BDE Administrator. See Using the Borland Database Engine for details.
ADO	Components that use ActiveX Data Objects (ADO), developed by Microsoft, to access database information. Many ADO drivers are available for connecting to different database servers. ADO-based components let you integrate your application into an ADO-based environment. See Working with ADO components for details.
dbExpress	Cross-platform components that use dbExpress to access database information. dbExpress drivers provide fast access to databases but need to be used with <i>TClientDataSet</i> and <i>TDataSetProvider</i> to perform updates. See Using unidirectional datasets for details.
InterBase	Components that access InterBase databases directly, without going through a separate engine layer. For more information about using the InterBase components, see the online Help Getting started with InterBase Express .
Data Access	Components that can be used with any data access mechanism such as <i>TClientDataSet</i> and <i>TDataSetProvider</i> . See Using client datasets for information about client datasets. See Using provider components for information about providers.
Data Controls	Data-aware controls that can access information from a data source. See Using data controls for details.

When designing a database application, you must decide which data access mechanism to use. Each data access mechanism differs in its range of functional support, the ease of deployment, and the availability of drivers to support different database servers.

Refer to [Designing database applications](#) for details on what type of database support is available and considerations when designing database client applications and application servers.

- DevGuide: [Building applications with Delphi](#)

Distributing database applications

[Topic groups](#)

[See also](#)

Delphi provides support for creating distributed database applications using a coordinated set of components. Distributed database applications can be built on a variety of communications protocols, including DCOM, CORBA, TCP/IP, and SOAP.

For more information about building distributed database applications, see [Creating multi-tiered applications](#).

Distributing database applications often requires you to distribute the Borland Database Engine (BDE) in addition to the application files. For information on deploying the BDE, see [Deploying database applications](#).

- DevGuide: [Building applications with Delphi](#)

Creating Web server applications

[Topic groups](#)

[See also](#)

Web server applications are applications that run on servers that deliver Web content such as HTML Web pages or XML documents over the Internet. Examples of Web server applications include those which control access to a Web site, generate purchase orders, or respond to information requests.

You can create several different types of Web server applications using the following Delphi technologies:

- [Web Broker](#)
- [WebSnap](#)
- [InternetExpress](#)
- [Web Services](#)

■ DevGuide: [Building applications with Delphi](#)

Using Web Broker

[Topic groups](#)

[See also](#)

You can use Web Broker (also called NetCLX architecture) to create Web server applications such as CGI applications or dynamic-link libraries (DLLs). These Web server applications can contain any nonvisual component. Components on the Internet page of the Component palette enable you to create event handlers, programmatically construct HTML or XML documents, and transfer them to the client.

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

Web server application type	Description
ISAPI and NSAPI Dynamic Link Library	ISAPI and NSAPI Web server applications are DLLs that are loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by TISAPIApplication. Each request message is handled in a separate execution thread. Selecting this type of application adds the library header of the project files and required entries to the uses list and exports clause of the project file.
CGI Stand-alone executable	CGI Web server applications are console applications that receive requests from clients on standard input, process those requests, and sends back the results to the server on standard output to be sent to the client. Selecting this type of application adds the required entries to the uses clause of the project file and adds the appropriate \$APPTYPE directive to the source.
Win-CGI Stand-alone executable	Win-CGI Web server applications are Windows applications that receive requests from clients from a configuration settings (INI) file written by the server and writes the results to a file that the server passes back to the client. The INI file is evaluated by TCGIApplication. Each request message is handled by a separate instance of the application. Selecting this type of application adds the required entries to the uses clause of the project file and adds the appropriate \$APPTYPE directive to the source.
Apache Shared Module (DLL)	Selecting this type of application sets up your project as a DLL. Apache Web server applications are DLLs loaded by the Web server. Information is passed to the DLL, processed, and returned to the client by the Web server.
Web App Debugger Stand-alone executable	Selecting this type of application sets up an environment for developing and testing Web server applications. Web App Debugger applications are executable files loaded by the Web server. This type of application is not intended for deployment.

CGI and Win-CGI applications use more system resources on the server, so complex applications are better created as ISAPI , NSAPI, or Apache DLL applications. If writing cross-platform applications, you should select CGI stand-alone or Apache Shared Module (DLL) for Web server development. These are also the same options you see when creating WebSnap and Web Service applications.

For more information on building Web server applications, see [Creating Internet server applications](#).

- DevGuide: [Building applications with Delphi](#)

Creating WebSnap applications

[Topic groups](#)

[See also](#)

WebSnap provides a set of components and wizards for building advanced Web servers that interact with Web browsers. WebSnap components generate HTML or other mime content for Web pages. WebSnap is for server side development. WebSnap cannot be used in cross-platform applications at this time.

To create a new WebSnap application, select File|New|Other and select the WebSnap tab in the New Items dialog box. Choose WebSnap Application. Then select the Web server application type (ISAPI/NSAPI, CGI, Win-CGI, Apache). See for details.

For more information on WebSnap, see [Creating Internet server applications](#).

- DevGuide: [Building applications with Delphi](#)

Using InternetExpress

[Topic groups](#)

[See also](#)

InternetExpress is a set of components that extends the basic Web server application architecture to act as the client of an application server. You use InternetExpress for applications wherein browser-based clients can fetch data from a provider, resolve updates to the provider, while executing on a client.

InternetExpress applications generate HTML pages that contain a mixture of HTML, XML, and javascript. The HTML determines the layout and appearance of the pages displayed in end-user browsers. The XML encodes the data packets and delta packets that represent database information. The javascript allows the HTML controls to interpret and manipulate the data in the XML data packets on the client machine.

For more information on InternetExpress, see [Building Web applications using InternetExpress](#).

- DevGuide: [Building applications with Delphi](#)

Creating Web Services applications

[Topic groups](#)

[See also](#)

Web Services are self-contained modular applications that can be published and invoked over a network (such as the World Wide Web). Web Services provide well-defined interfaces that describe the services provided. You use Web Services to produce or consume programmable services over the Internet using emerging standards such as XML, XML Schema, SOAP (Simple Object Access Protocol), and WSDL (Web Service Definition Language).

Web Services use SOAP, a standard lightweight protocol for exchanging information in a distributed environment. It uses HTTP as a communications protocol and XML to encode remote procedure calls.

You can use Delphi to build servers to implement Web Services and clients that call on those services. You can write clients for arbitrary servers to implement Web Services that respond to SOAP messages, and Delphi servers to publish Web Services for use by arbitrary clients.

Refer to [Using Web Services](#) for more information on Web Services.

- DevGuide: [Building applications with Delphi](#)

Writing applications using COM

[Topic groups](#)

[See also](#)

COM is the Component Object Model, a Windows-based distributed object architecture designed to provide object interoperability using predefined routines called interfaces. COM applications use objects that are implemented by a different process or, if you use DCOM, on a separate machine. You can also use COM+, ActiveX and Active Server Pages.

COM is a language-independent software component model that enables interaction between software components and applications running on a Windows platform. The key aspect of COM is that it enables communication between components, between applications, and between clients and servers through clearly defined interfaces. Interfaces provide a way for clients to ask a COM component which features it supports at runtime. To provide additional features for your component, you simply add an additional interface for those features.

Using COM and DCOM

Delphi has classes and wizards that make it easy to create COM, OLE, or ActiveX applications. You can create COM clients or servers that implement COM objects, Automation servers (including Active Server Objects), ActiveX controls, or ActiveForms. COM also serves as the basis for other technologies such as Automation, ActiveX controls, Active Documents, and Active Directories.

Using Delphi to create COM-based applications offers a wide range of possibilities, from improving software design by using interfaces internally in an application, to creating objects that can interact with other COM-based API objects on the system, such as the Win9x Shell extensions and DirectX multimedia support. Applications can access the interfaces of COM components that exist on the same computer as the application or that exist on another computer on the network using a mechanism called Distributed COM (DCOM).

For more information on COM and Active X controls, see [Overview of COM technologies](#), [Creating an ActiveX control](#) and [Distributing a client application as an ActiveX control](#).

For more information on DCOM, see [Using DCOM connections](#).

Using MTS and COM+

COM applications can be augmented with special services for managing objects in a large distributed environment. These services include transaction services, security, and resource management supplied by Microsoft Transaction Server (MTS) on versions of Windows prior to Windows 2000) or COM+ (for Windows 2000 and later).

For more information on MTS and COM+, see [Creating MTS or COM+ objects](#) and [Using transactional data modules](#).

- DevGuide: [Building applications with Delphi](#)

Using data modules

[Topic groups](#)

[See also](#)

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 several types of data modules, including standard, remote, Web modules, applet modules, and services, depending on which edition of Delphi you have. Each type of data module serves a special purpose.

- Standard data modules 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 and editing data modules](#).
- Remote data modules form the basis of an application server in a multi-tiered database application. They are not available in all editions. In addition to holding the nonvisual components in the application server, remote data modules expose the interface that clients use to communicate with the application server. For more information about using them, see [Adding a remote data module to an application server project](#).
- 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 [Creating Internet server applications](#) for more information about using Web modules.
- Applet modules form the basis of control panel applets. In addition to holding the nonvisual controls that implement the control panel applet, they define the properties that determine how the applet's icon appears in the control panel and include the events that are called when users execute the applet. For more information about applet modules, see [Control Panel Application wizard](#).
- Services encapsulate individual services in an NT service application. In addition to holding any nonvisual controls used to implement a service, services include the events that are called when the service is started or stopped. For more information about services, see [Service Applications](#).

■ DevGuide: [Building applications with Delphi](#)

Creating and editing standard data modules

[Topic groups](#)

[See also](#)

To create a standard data module for a project, choose File|New|Data Module. Delphi opens a data module container on the desktop, displays the unit file for the new module in the Code editor, and adds the module to the current project.

At design time, a data module looks like a standard Delphi form with a white background and no alignment grid. As with forms, you can place nonvisual components from the Component palette onto a module, and edit their properties in the Object Inspector. You can resize a data module to accommodate the components you add to it.

You can also right-click a module to display a context menu for it. The following table summarizes the context menu options for a data module.

<u>Menu item</u>	<u>Purpose</u>
Edit	Displays a context menu with which you can cut, copy, paste, delete, and select the components in the data module.
Position	Aligns nonvisual components to the module's invisible grid (<i>Align To Grid</i>) or according to criteria you supply in the Alignment dialog box (<i>Align</i>).
Tab Order	Enables you to change the order that the focus jumps from component to component when you press the tab key.
Creation Order	Enables you to change the order that data access components are created at start-up.
Revert to Inherited	Discards changes made to a module inherited from another module in the Object Repository, and reverts to the originally inherited module.
Add to Repository	Stores a link to the data module in the Object Repository.
View as Text	Displays the text representation of the data module's properties.
View DFM	Toggles between the formats (binary or text) in which this particular form file is saved.

For more information about data modules, see [About the Data Module Editor](#).

■ DevGuide: [Building applications with Delphi](#)

Naming a data module and its unit file

[Topic groups](#)

[See also](#)

The title bar of a data module displays the module's name. The default name for a data module is "DataModuleN" where N is a number representing the lowest unused unit number in a project. For example, if you start a new project, and add a module to it before doing any other application building, the name of the module defaults to "DataModule2." The corresponding unit file for *DataModule2* defaults to "Unit2."

You should rename your data modules and their corresponding unit files at design time to make them more descriptive. You should especially rename data modules you add to the Object Repository to avoid name conflicts with other data modules in the Repository or in applications that use your modules.

To rename a data module:

- 1 Select the module.
- 2 Edit the *Name* property for the module in the Object Inspector.

The new name for the module appears in the title bar when the *Name* property in the Object Inspector no longer has focus.

Changing the name of a data module at design time changes its variable name in the interface section of code. It also changes any use of the type name in procedure declarations. You must manually change any references to the data module in code you write.

To rename a unit file for a data module:

- 1 Select the unit file.

- DevGuide: [Building applications with Delphi](#)

Placing and naming components

[Topic groups](#)

[See also](#)

You place nonvisual components in a data module just as you place visual components on a form. Click the desired component on the appropriate page of the Component palette, then click in the data module to place the component. You cannot place visual controls, such as grids, on a data module. If you attempt it, you receive an error message.

For ease of use, components are displayed with their names in a data module. When you first place a component, Delphi assigns it a generic name that identifies what kind of component it is, followed by a *1*. For example, the *TDataSource* component adopts the name *DataSource1*. This makes it easy to select specific components whose properties and methods you want to work with.

You may still want to name a component a different name that reflects the type of component and what it is used for.

To change the name of a component in a data module:

- 1 Select the component.
- 2 Edit the component's *Name* property in the Object Inspector.

The new name for the component appears under its icon in the data module as soon as the *Name* property in the Object Inspector no longer has focus.

For example, suppose your database application uses the CUSTOMER table. To access the table, you need a minimum of two data access components: a data source component (*TDataSource*) and a table component (*TClientDataSet*). When you place these components in your data module, Delphi assigns them the names *DataSource1* and *ClientDataSet1*. To reflect the type of component and the database they access, CUSTOMER, you could change these names to *CustomerSource* and *CustomerTable*.

- DevGuide: [Building applications with Delphi](#)

Using component properties and events in a data module

[Topic groups](#)

[See also](#)

Placing components in a data module centralizes their behavior for your entire application. For example, you can use the properties of dataset components, such as *TClientDataSet*, to control the data available to the data source components that use those datasets. Setting the *ReadOnly* property to *True* for a dataset prevents users from editing the data they see in a data-aware visual control on a form. You can also invoke the Fields editor for a dataset, by double-clicking on *ClientDataSet1*, to restrict the fields within a table or query that are available to a data source and therefore to the data-aware controls on forms. The properties you set for components in a data module apply consistently to all forms in your application that use the module.

In addition to properties, you can write event handlers for components. For example, a *TDataSource* component has three possible events: *OnDataChange*, *OnStateChange*, and *OnUpdateData*. A *TClientDataSet* component has over 20 potential events. You can use these events to create a consistent set of business rules that govern data manipulation throughout your application.

- DevGuide: [Building applications with Delphi](#)

Creating business rules in a data module

[Topic groups](#)

[See also](#)

Besides writing event handlers for the components in a data module, you can code methods directly in the unit file for a data module. These methods can be applied to the forms that use the data module as business rules. For example, you might write a procedure to perform month-, quarter-, or year-end bookkeeping. You might call the procedure from an event handler for a component in the data module. The prototypes for the procedures and functions you write for a data module should appear in the module's **type** declaration:

```
type
  TCustomerData = class(TDataModule)
    Customers: TClientDataSet;
    Orders: TClientDataSet;
    ...
  private
    { Private declarations }
  public
    { Public declarations }
    procedure LineItemsCalcFields(DataSet: TDataSet); { A procedure you add }
  end;
var
  CustomerData: TCustomerData;
```

The procedures and functions you write should follow in the implementation section of the code for the module.

- DevGuide: [Building applications with Delphi](#)

Accessing a data module from a form

[Topic groups](#)

[See also](#)

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.
- Click the form's unit file, choose File|Use Unit, and enter the name of the module or pick it from the list box in the Use Unit dialog.
- For database components, in the data module click a dataset or query component to open the [Fields editor](#) and drag any existing fields from the editor onto the form. Delphi 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.

For example, if you've added the *TClientDataSet* component to your data module, double-click it to open the Fields editor. Select a field and drag it to the form. An edit box component appears.

Because the data source is not yet defined, Delphi adds a new data source component, *DataSource1*, to the form and sets the edit box's *DataSource* property to *DataSource1*. The data source automatically sets its *DataSet* property to the dataset component, *ClientDataSet1*, in the data module.

You can define the data source *before* you drag a field to the form by adding a *TDataSource* component to the data module. Set the data source's *DataSet* property to *ClientDataSet1*. After you drag a field to the form, the edit box appears with its *TDataSource* property already set to *DataSource1*. This method keeps your data access model cleaner.

- DevGuide: [Building applications with Delphi](#)

Adding a remote data module to an application server project

[Topic groups](#)

[See also](#)

Some editions of Delphi allow you to add *remote data modules* to application server projects. A remote data module has an interface that clients in a multi-tiered application can access across networks.

To add a remote data module to a project:

- 1 Choose File|New|Other.
- 2 Select the Multitier page in the New Items dialog box.
- 3 Double-click the desired type of module (CORBA Data Module, Remote Data Module, or Transactional Data Module) to open the Remote Data Module wizard.

Once you add a remote data module to a project, you use it just like a standard data module.

For more information about multi-tiered database applications, see [Creating multi-tiered applications](#).

- DevGuide: [Building applications with Delphi](#)

Using the Object Repository

[Topic groups](#)

[See also](#)

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 DELPHI32.DRO (by default in the BIN directory), a text file that contains references to the items that appear in the Repository and New Items dialogs.

- DevGuide: [Building applications with Delphi](#)

Sharing items within a project

[Topic groups](#)

[See also](#)

You can share items *within* a project without adding them to the Object Repository. When you open the New Items dialog box (File|New|Other), 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.

- DevGuide: [Building applications with Delphi](#)

Adding items to the Object Repository

[Topic groups](#)

[See also](#)

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, Delphi creates a new page.
- 5 Choose Browse to select an icon to represent the object in the Object Repository.
- 6 Choose OK.

- DevGuide: [Building applications with Delphi](#)

Sharing objects in a team environment

[Topic groups](#)

[See also](#)

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, Delphi creates a DELPHI32.DRO file in the Shared Repository directory if one doesn't exist already.

- DevGuide: [Building applications with Delphi](#)

Using an Object Repository item in a project

[Topic groups](#)

[See also](#)

To access items in the Object Repository, choose File|New|Other. 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](#)

- DevGuide: [Building applications with Delphi](#)

Copying an item

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Inheriting an item

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Using an item

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Using project templates

[Topic groups](#)

[See also](#)

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|Other 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.

Delphi copies the template files to the specified directory, where you can modify them. The original project template is unaffected by your changes.

- DevGuide: [Building applications with Delphi](#)

Modifying shared items

[Topic groups](#)

[See also](#)

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, DLL, component template, or frame from the item. If you create a component or DLL, you can share it with other developers.

- DevGuide: [Building applications with Delphi](#)

Specifying a default project, new form, and main form

[Topic groups](#)

[See also](#)

By default, when you choose File|New|Application or File|New|Form, Delphi 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), then 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.

- DevGuide: [Building applications with Delphi](#)

Enabling Help in applications

[Topic groups](#)

[See also](#)

Both the VCL and CLX support displaying Help from applications using an object-based mechanism that allows Help requests to be passed on to one of multiple external Help viewers. To support this, an application must include a class that implements the *ICustomHelpViewer* interface (and, optionally, one of several interfaces descended from it), and registers itself with the global Help Manager.

The VCL provides to all applications an instance of *TWinHelpViewer*, which implements all of these interfaces and provides a link between applications and WinHelp; CLX requires that application developers provide their own implementation.

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).

- DevGuide: [Building applications with Delphi](#)

Help system interfaces

[Topic groups](#)

[See also](#)

The 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 Help system, but "String manipulation routines" could be the name of a topic).

ISpecialWinHelpViewer 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.

- DevGuide: [Building applications with Delphi](#)

Implementing ICustomHelpViewer

[Topic groups](#)

[See also](#)

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.

For information on ICustomHelpViewer methods, see

- [Communicating with the Help Manager](#)
- [Displaying keyword-based Help](#)
- [Asking the Help Manager for information](#)

- DevGuide: [Building applications with Delphi](#)

Communicating with the Help Manager

[Topic groups](#)

[See also](#)

ICustomHelpViewer provides four functions that can be used to communicate system information with the Help Manager:

- *GetViewerName*
- *NotifyID*
- *ShutDown*
- *SoftShutDown*

The Help Manager 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.
- *ICustomHelpViewer.SoftShutDown* is called by the Help Manager to ask the Help viewer to close any externally visible manifestations of the help system (for example, windows displaying help information) without unloading the viewer.

- DevGuide: [Building applications with Delphi](#)

Asking the Help Manager for information

[Topic groups](#)

[See also](#)

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.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.

- [DevGuide: Building applications with Delphi](#)

Displaying keyword-based Help

[Topic groups](#)

[See also](#)

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:

- *UnderstandsKeyword*
- *GetHelpStrings*
- *ShowHelp*

`ICustomHelpViewer.UnderstandsKeyword(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 IDE, the HyperHelp viewer maintains its own index and searches 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 *UnderstandsKeyword* is invoked first.

- DevGuide: [Building applications with Delphi](#)

Displaying tables of contents

[Topic groups](#)

[See also](#)

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*.

It is reasonable for a particular viewer to refuse to allow 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*.

- DevGuide: [Building applications with Delphi](#)

Implementing `IExtendedHelpViewer`

[Topic groups](#)

[See also](#)

`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.UnderstandsKeyword`, the Help Manager queries all registered Help viewers iteratively. Unlike the case with `ICustomHelpViewer.UnderstandsKeyword`, 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.

- DevGuide: [Building applications with Delphi](#)

Implementing IHelpSelector

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

RegisteringHelpSystemObjects

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Using Help in a VCL Application

[Topic groups](#)

[See also](#)

The following sections explain how to use Help within a VCL application.

- [How TApplication processes VCL Help](#)
- [How VCL controls process Help](#)
- [Calling a Help system directly](#)
- [Using IHelpSystem](#)

- DevGuide: [Building applications with Delphi](#)

How TApplication processes VCL Help

[Topic groups](#)

[See also](#)

TApplication in the VCL provides four methods that are accessible from application code:

HelpCommand	Takes a Windows Help style HELP_COMMAND and passes it off to WinHelp. Help requests forwarded through this mechanism are passed only to implementations of ISpecialWinHelpViewer.
HelpContext	Invokes the Help System with a request for context-based Help.
HelpKeyword	Invokes the HelpSystem with a request for keyword-based Help.
HelpJump	Requests the display of a particular topic.

All four functions take the data passed to them and forward it through a data member of *TApplication* which represents the Help System. That data member is directly accessible through the property *HelpSystem*.

- DevGuide: [Building applications with Delphi](#)

How VCL controls process Help

[Topic groups](#)

[See also](#)

All controls that derive from *TControl* expose three properties which are used by the Help system: *HelpSystem*, *HelpType*, *HelpContext*, and *HelpKeyword*.

The *HelpType* property contains an instance of an enumerated type that determines if the control's designer expects help to be provided via keyword-based Help or context-based Help. 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*, that 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 *TWinControl* calls *InvokeHelp*.

- DevGuide: [Building applications with Delphi](#)

Using Help in a CLX Application

[Topic groups](#)

[See also](#)

The following sections explain how to use Help within a CLX application.

- [How TApplication processes CLX Help](#)
- [How CLX controls process Help](#)
- [Calling a Help system directly](#)
- [Using IHelpSystem](#)

- DevGuide: [Building applications with Delphi](#)

How TApplication processes CLX Help

[Topic groups](#)

[See also](#)

TApplication in CLX 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*.

- DevGuide: [Building applications with Delphi](#)

How CLX controls process Help

[Topic groups](#)

[See also](#)

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*.

- DevGuide: [Building applications with Delphi](#)

Calling a Help system directly

[Topic groups](#)

[See also](#)

For additional Help system functionality not provided by the VCL or CLX, *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.

- DevGuide: [Building applications with Delphi](#)

Using IHelpSystem

[Topic groups](#)

[See also](#)

IHelpSystem allows a VCL or CLX application to do three things:

- Provides path information to the Help Manager
- Provides a new Help selector
- Asks the Help Manager to display help

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 topic [Implementing IHelpSelector](#).

IHelpSystem exports four procedures and one function to request the Help Manager to display help:

- *ShowHelp*
- *ShowContextHelp*
- *ShowTopicHelp*
- *ShowTableOfContents*
- *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.

■ DevGuide: [Building applications with Delphi](#)

Customizing the IDE Help system

[Topic groups](#)

[See also](#)

The Delphi IDE supports multiple Help viewers in exactly the same way that a VCL or CLX application does: it delegates Help requests to the Help Manager, which forwards them to registered Help viewers. The IDE makes use of the same WinHelpViewer that the VCL uses.

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 initialization section of the unit, make sure that the instance is passed to the *RegisterHelpViewer* function.

- DevGuide: [Building applications with Delphi](#)

Developing the application user interface: Overview

[Topic groups](#)

With Delphi, you design a user interface (UI) by selecting components from the component palette and dropping them onto forms. You get the components to do what you want by setting their properties and coding their event handlers. Select Topic Groups or browse (choose >>) to read topics on how to develop the user interface.

- DevGuide: [Building applications with Delphi](#)

Controlling application behavior

[Topic groups](#)

[See also](#)

TApplication, *TScreen*, and *TForm* are the classes that form the backbone of all Delphi 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 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 the *TForm* class are the building blocks of your application's user interface. The windows and dialog boxes in your application are based on *TForm*.

- DevGuide: [Building applications with Delphi](#)

Using the main form

[Topic groups](#)

[See also](#)

TForm is the key class for creating GUI applications. When you open Delphi displaying a default project or when you create a new project, a form is displayed on which you can 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.

- DevGuide: [Building applications with Delphi](#)

Adding forms

[Topic groups](#)

[See also](#)

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.

■ DevGuide: [Building applications with Delphi](#)

Hiding the main form

[Topic groups](#)

[See also](#)

You can prevent the main form from displaying when your application first starts up. To do so, you must use the global *Application* variable.

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.

- DevGuide: [Building applications with Delphi](#)

Working at the application level

[Topic groups](#)

[See also](#)

The global variable *Application*, of type *TApplication*, is in every VCL or 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 *OnMessage* event lets you intercept Windows messages (on Windows only), 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.

- DevGuide: [Building applications with Delphi](#)

Handling the screen

[Topic groups](#)

[See also](#)

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
- multiple screen behavior (not available for cross-platform)

If your Windows application runs on multiple monitors, *Screen* maintains a list of monitors and their dimensions so that you can effectively manage the layout of your user interface.

If using CLX for cross-platform programming, the default behavior is that applications create a screen component based on information about the current screen device and assign it to *Screen*.

■ DevGuide: [Building applications with Delphi](#)

Managing layout

[Topic groups](#)

[See also](#)

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.

If you want to keep a control positioned relative to a particular edge of its parent, but don't want it to necessarily touch that edge or be resized so that it always runs along the entire edge, you can use the *Anchors* property.

If you want to ensure that a control does not grow too big or too small, you can use the *Constraints* property. *Constraints* lets you specify the control's maximum height, minimum height, maximum width, and minimum width. Set these to limit the size (in pixels) of the control's height and width. For example, by setting the *MinWidth* and *MinHeight* of the constraints on a container object, you can ensure that child objects are always visible.

The value of *Constraints* propagates through the parent/child hierarchy so that an object's size can be constrained because it contains aligned children that have size constraints. *Constraints* can also prevent a control from being scaled in a particular dimension when its *ChangeScale* method is called.

TControl introduces a protected event, *OnConstrainedResize*, of type *TConstrainedResizeEvent*:

```
TConstrainedResizeEvent = procedure (Sender: TObject; var MinWidth, MinHeight,  
MaxWidth, MaxHeight: Integer) of object;
```

This event allows you to override the size constraints when an attempt is made to resize the control. The values of the constraints are passed as var parameters which can be changed inside the event handler.

OnConstrainedResize is published for container objects (*TForm*, *TScrollBar*, *TControlBar*, and *TPanel*). In addition, component writers can use or publish this event for any descendant of *TControl*.

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.

- DevGuide: [Building applications with Delphi](#)

Responding to event notification

[Topic groups](#)

[See also](#)

The operating system will notify your application when an event has occurred (such as a mouse click, keystrokes entered, and so on) while it is running. The underlying way that event notifications are handled by VCL and CLX objects is different, but the way you work with event notifications at the component level is typically the same. Components have events and methods built-in for the most commonly occurring events. You can use the methods provided with the component in most cases. If you need to write additional event handling, you can override an existing method to write your own. Unless you are writing your own components, you do not need to change the underlying event notification schema.

VCL: If developing applications for Windows only, you need to understand that Windows is a message-based operating system. System messages are handled by a message handler that translates the message to an event or event handler. The message itself is a record passed to a control by Windows. For instance, when you click a mouse button on a dialog box, Windows sends a message to the active control and the application containing that control reacts to this new event. If the click occurs over a button, the *OnClick* event could be activated upon receipt of the message. If the click occurs just in the form, the application can ignore the message.

The record type passed to the application by Windows is called a *TMsg*. Windows predefines a constant for each message, and these values are stored in the message field of the *TMsg* record. Each of these constants begin with the letters *wm*.

The VCL automatically handles messages unless you override the message handling system and create your own message handlers. For more information on messages and message handling, see [Understanding the message-handling system](#), [Changing message handling](#), and [Creating new message handlers](#).

CLX: For cross-platform programming: The operating system notification that an event occurred is sent to the underlying Qt widget layer where it is translated into an event and eventually into event objects by [HookEvents](#). *EventFilter* is called automatically when a CLX control needs to handle a Qt mouse or keyboard event.

EventFilter responds to event notifications by performing the default response. Typically, this involves dispatching the event to the appropriate virtual method (such as the *Click* method, which generates an *OnClick* event).

CLX Note: When overriding the *EventFilter* method, you need to call the inherited method so that the default event processing can occur.

- DevGuide: [Building applications with Delphi](#)

Using forms

[Topic groups](#)

[See also](#)

When you create a form in Delphi from the IDE, Delphi automatically creates the form in memory by including code in the main entry point of your application function. 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 Delphi 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.

- DevGuide: [Building applications with Delphi](#)

Controlling when forms reside in memory

[Topic groups](#)

[See also](#)

By default, Delphi automatically creates the application's main form in memory by including the following code in the application's main entry point:

```
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.

- DevGuide: [Building applications with Delphi](#)

Displaying an auto-created form

[Topic groups](#)

[See also](#)

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.

■ DevGuide: [Building applications with Delphi](#)

Creating forms dynamically

[Topic groups](#)

[See also](#)

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. As an alternative, you can manually remove the following line from program's main entry point:

```
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);
  try
    ResultsForm.ShowModal;
  finally
    ResultsForm.Free;
  end;
```

In the above example, note the use of **try..finally**. Putting in the line `ResultsForm.Free;` in the **finally** clause ensures that the memory for the form is freed even if the form raises an exception.

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, Delphi 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.

- DevGuide: [Building applications with Delphi](#)

Creating modeless forms such as windows

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Using a local variable to create a form instance

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Passing additional arguments to forms

[Topic groups](#)

[See also](#)

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
  inherited Create(Owner);
  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;
```

- DevGuide: [Building applications with Delphi](#)

Retrieving data from forms

[Topic groups](#)

[See also](#)

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.

■ DevGuide: [Building applications with Delphi](#)

Retrieving data from modeless forms

[Topic groups](#)

[See also](#)

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[Index];
```

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*.

■ DevGuide: [Building applications with Delphi](#)

Retrieving data from modal forms

[Topic groups](#)

[See also](#)

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](#).

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).

- DevGuide: [Building applications with Delphi](#)

Reusing components and groups of components

[Topic groups](#)

[See also](#)

Delphi offers several ways to save and reuse work you've done with components:

- [Component templates](#) provide a simple, quick way of configuring and saving groups of components.
- 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.
- 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.
- Creating a [custom component](#) is the most complicated way of reusing code, but it offers the greatest flexibility.

- DevGuide: [Building applications with Delphi](#)

Creating and using component templates

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Working with frames

[Topic groups](#)

[See also](#)

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](#)

[Using and modifying frames](#)

[Sharing frames](#)

- DevGuide: [Building applications with Delphi](#)

Creating frames

[Topic groups](#)

[See also](#)

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.

■ DevGuide: [Building applications with Delphi](#)

Using and modifying frames

[Topic groups](#)

[See also](#)

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.

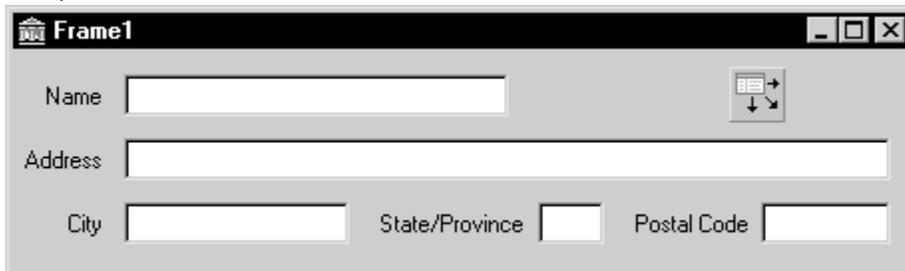
The Form Designer provides two ways to add a frame to an application:

- 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, Delphi declares a new class that descends from the frame you selected. (Similarly, when you add a new form to a project, Delphi 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.



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 *Image* 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.

- DevGuide: [Building applications with Delphi](#)

Sharing frames

[Topic groups](#)

[See also](#)

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 (.dfm or .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](#).

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, they will have to open it as part of a project.

- DevGuide: [Building applications with Delphi](#)

Organizing actions for toolbars and menus

[Topic groups](#)

[See also](#)

Delphi provides several features that simplify the work of creating, customizing, and maintaining menus and toolbars. These features allow you to organize lists of actions that users of your application can initiate by pressing a button on a toolbar, choosing a command on a menu, or pointing and clicking on an icon.

Often a set of actions is used in more than one user interface element. For example, the Cut, Copy, and Paste commands often appear on both an Edit menu and on a toolbar. You only need to add the action once to use it in multiple UI elements in your application.

On the Windows platform, tools are provided to make it easy to define and group actions, create different layouts, and customize menus at design time or runtime. These tools are known collectively as ActionBand tools, and the menus and toolbars you create with them are known as action bands. In general, you can create an ActionBand user interface as follows:

- Build the action list to create a set of actions that will be available for your application (use the Action Manager, [TActionManager](#))
- Add the user interface elements to the application (use ActionBand components such as [TActionMainMenuBar](#) and [TActionToolBar](#))
- Drag and drop actions from the Action Manager onto the user interface elements

The following table defines the terminology related to setting up menus and toolbars:

Term	Definition
Action	A response to something a user does, such as clicking a menu item. Many standard actions that are frequently required are provided for you to use in your applications as is. For example, file operations such as File Open, File SaveAs, File Run, and File Exit are included along with many others for editing, formatting, searches, help, dialogs, and window actions. You can also program custom actions and access them using action lists and the Action Manager.
Action band	A container for a set of actions associated with a customizable menu or toolbar. The ActionBand components for main menus and toolbars (TActionMainMenuBar and TActionToolBar) are examples of action bands.
Action category	Lets you group actions and drop them as a group onto a menu or toolbar. For example, one of the standard action categories is Search which includes Find, FindFirst, FindNext, and Replace actions all at once.
Action classes	Classes that perform the actions used in your application. All of the standard actions are defined in action classes such as TEditCopy , TEditCut , and TEditUndo . You can use these classes by dragging and dropping them from the Customize dialog onto an action band.
Action client	Most often represents a menu item or a button that receives a notification to initiate an action. When the client receives a user command (such as a mouse click), it initiates an associated action.
Action list	Maintains a list of actions that your application can take in response to something a user does.
Action Manager	Groups and organizes logical sets of actions that can be reused on ActionBand components. See TActionManager .
Menu	Lists commands that the user of the application can execute by clicking on them. You can create menus by using the ActionBand menu class TActionMainMenuBar , or by using cross-platform components such as TMainMenu or TPopupMenu .
Target	Represents the item an action does something to. The target is usually a control, such as a memo or a data control. Not all actions require a target. For example, the standard help actions ignore the target and simply launch the help system.
Toolbar	Displays a visible row of button icons which, when clicked, cause the program to perform some action, such as printing the current document. You can create toolbars by using the ActionBand toolbar component TActionToolBar , or by using the cross-platform component TToolBar .

If you are doing cross-platform development, refer to [Using action lists](#) for details.

- DevGuide: [Building applications with Delphi](#)

What is an action?

[Topic groups](#)

[See also](#)

As you are developing your application, you can create a set of actions that you can use on various UI elements. You can organize them into categories that can be dropped onto a menu as a set (for example, Cut, Copy, and Paste) or one at a time (for example, Tools|Customize).

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. You can create a repertoire of actions that are available to your application through menus, through buttons, through toolbars, context menus, and so on.

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, *TToolBar*, *TSpeedButton*, *TMenuItem*, *TButton*, *TCheckBox*, *TRadioButton*, and so on). Actions also reside on *ActionBand* components such as *TActionMainMenuBar* and *TActionToolBar*. 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 *OnExecute* 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 can add, delete, and rearrange actions using the Action Manager or the Action List editor (displayed by double-clicking an action list object, *TActionList*). These actions are later connected to client controls. See [Creating toolbars and menus](#) and, for cross-platform development, [Setting up action lists](#) for details.

- DevGuide: [Building applications with Delphi](#)

Setting up action bands

[Topic groups](#)

[See also](#)

Because actions do not maintain any “layout” (either appearance or positional) information, Delphi provides action bands which are capable of storing this data. Action bands provide a mechanism that allows you to specify layout information and a set of controls. You can render actions as UI elements such as toolbars and menus.

You organize sets of actions using the Action Manager ([TActionManager](#)). You can use standard actions provided or create new actions of your own.

You then create the action bands:

- Use [TActionMainMenuBar](#) to create a main menu.
- Use [TActionToolBar](#) to create a toolbar.

The action bands act as containers that hold and render sets of actions. You can drag and drop items from the Action Manager editor onto the action band at design time. At runtime, application users can also customize the application’s menus or toolbars using a dialog box similar to the Action Manager editor.

- DevGuide: [Building applications with Delphi](#)

Creating toolbars and menus

[Topic groups](#)

[See also](#)

Note: This section describes the recommended method for creating menus and toolbars in Windows applications. For cross-platform development, you need to use *TToolBar* and the menu components, such as *TMainMenu*, organizing them using action lists (*TActionList*). See [Setting up action lists](#) for details.

You use the Action Manager to automatically generate toolbars and main menus based on the actions contained in your application. The Action Manager manages standard actions and any custom actions that you have written. You then create UI elements based on these actions and use action bands to render the actions items as either menu items or as buttons on a toolbar.

The general procedure for creating menus, toolbars, and other action bands involves these steps:

- Drop an Action Manager onto a form.
- Add actions to the Action Manager, which organizes them into appropriate action lists.
- Create the action bands (that is, the menu or the toolbar) for the user interface.
- Drag and drop the actions into the application interface.

The following procedure explains these steps in more detail.

To create menus and toolbars using action bands:

- 1 From the Additional page of the component palette, drop an Action Manager component ([TActionManager](#)) onto the form where you want to create the toolbar or menu.
- 2 If you want images on the menu or toolbar, drop an ImageList component from the Win32 page of the component palette onto a form. (You need to add the images you want to use to the ImageList or use the one provided.)
- 3 From the Additional page of the component palette, drop one or more of the following action bands onto the form:
 - [TActionMainMenuBar](#) (for designing main menus)
 - [TActionToolBar](#) (for designing toolbars)
- 4 Connect the ImageList to the Action Manager: with focus on the Action Manager and in the Object Inspector, select the name of the ImageList from the Images property.
- 5 Add actions to the Action Manager editor's action pane:
 - Double-click the Action Manager to display the Action Manager editor.
 - Click the drop-down arrow next to the New Action button (the leftmost button at the top right corner of the Actions tab) and select New Action or New Standard Action. A tree view is displayed. Add one or more actions or categories of actions to the Action Manager's actions pane. The Action Manager adds the actions to its action lists.
- 6 Drag and drop single actions or categories of actions from the Action Manager editor onto the menu or toolbar you are designing.

To add user-defined actions, create a new *TAction* by pressing the New Action button and writing an event handler that defines how it will respond when fired. See [What happens when an action fires](#) for details. Once you've defined the actions, you can drag and drop them onto menus or toolbars like the standard actions.

- DevGuide: [Building applications with Delphi](#)

Adding icons to menus and toolbars

[Topic groups](#)

[See also](#)

You can use the *Background* and *BackgroundLayout* properties to specify a color, pattern, or bitmap to use on a menu item or button. These properties also let you set up a banner that runs up the left or right side of a menu.

You assign backgrounds and layouts to subitems from their action client objects. If you want to set the background of the items in a menu, in the form designer click on the menu item that contains the items. For example, selecting File lets you change the background of items appearing on the File menu. You can assign a color, pattern, or bitmap in the *Background* property in the Object Inspector.

Use the *BackgroundLayout* property to describe how to place the background on the element. Colors or images can be placed behind the caption normally, stretched to fit the item area, or tiled in small squares to cover the area.

Items with normal (blNormal), stretched (blStretch), or tiled (blTile) backgrounds are rendered with a transparent background. If you create a banner, the full image is placed on the left (blLeftBanner) or the right (blRightBanner) of the item. You need to make sure it is the correct size because it is not stretched or shrunk to fit.

To change the background of an action band (that is, on a main menu or toolbar), select the action band and choose the *TActionClientBar* through the action band collection editor. You can set *Background* and *BackgroundLayout* properties to specify a color, pattern, or bitmap to use on the entire toolbar or menu.

- DevGuide: [Building applications with Delphi](#)

Adding icons to menus and toolbars

[Topic groups](#)

[See also](#)

You can add icons next to menu items or replace captions on toolbars with icons. You organize bitmaps or icons using an ImageList.

- 1 Drop an ImageList component from the Win32 page of the component palette onto a form.
- 2 Add the images you want to use to the ImageList: Double-click the ImageList. Click Add and navigate to the images you want to use and click OK when done. Some sample images are included in Program Files\Common Files\Borland Shared\Images. (The buttons images include two views of each for active and inactive buttons.)
- 3 From the Additional page of the component palette, drop one or more of the following action bands onto the form:
 - [TActionMainMenuBar](#) (for designing main menus)
 - [TActionToolBar](#) (for designing toolbars)
- 4 Connect the ImageList to the Action Manager. First, set the focus on the Action Manager. Next, in the Object Inspector, select the name of the ImageList from the Images property.
- 5 Use the Action Manager editor to add actions to the Action Manager. You can associate an image with an action by setting its *ImageIndex* property to its number in the ImageList.
- 6 Drag and drop single actions or categories of actions from the Action Manager editor onto the menu or toolbar.
- 7 For toolbars where you only want to display the icon and no caption: select the Toolbar action band and double-click its Items property. In the collection editor, you can select one or more items and set their Caption properties.
- 8 The images automatically appear on the menu or toolbar.

- DevGuide: [Building applications with Delphi](#)

Creating toolbars and menus that users can customize

[Topic groups](#)

[See also](#)

You can use action bands with the Action Manager to create customizable toolbars and menus. At runtime, users of your application can customize the toolbars and menus (action bands) in the application user interface using a customization dialog similar to the Action Manager editor.

To allow the user of your application to customize an action band in your application:

- 1 Drop an Action Manager component onto a form.
- 2 Drop your action band components ([TActionMainMenuBar](#), [TActionToolBar](#)).
- 3 Double-click the Action Manager to display the Action Manager editor:
 - Add the actions you want to use in your application. Also add the Customize action, which appears at the bottom of the standard actions list.
 - Drop a [TCustomizeDlg](#) component from the Additional tab onto the form, and connect it to the Action Manager using its `ActionManager` property. You specify a filename for where to stream customizations made by users.
 - Drag and drop the actions onto the action band components. (Make sure you add the Customize action to the toolbar or menu.)
- 4 Complete your application.

When you compile and run the application, users can access a Customize command that displays a customization dialog box similar to the Action Manager editor. They can drag and drop menu items and create toolbars using the same actions you supplied in the Action Manager.

- DevGuide: [Building applications with Delphi](#)

Hiding unused items and categories in action bands

[Topic groups](#)

[See also](#)

One benefit of using ActionBands is that unused items and categories can be hidden from the user. Over time, the action bands become customized for the application users, showing only the items that they use and hiding the rest from view. Hidden items can become visible again when the user presses a drop-down button. Also, the user can restore the visibility of all action band items by resetting the usage statistics from the customization dialog. Item hiding is the default behavior of action bands, but that behavior can be changed to prevent hiding of individual items, all the items in a particular collection (like the File menu), or all of the items in a given action band.

The action manager keeps track of the number of times an action has been called by the user, which is stored in the associated *TActionClientItem*'s *UsageCount* field. The action manager also records the number of times the application has been run, which we shall call the session number, as well as the session number of the last time an action was used. The value of *UsageCount* is used to look up the maximum number of sessions the item can go unused before it becomes hidden, which is then compared with the difference between the current session number and the session number of the last use of the item. If that difference is greater than the number determined in *PrioritySchedule*, the item is hidden. The default values of *PrioritySchedule* are shown in the table below:

<u>Number of sessions in which an action band item was used</u>	<u>Number of sessions an item will remain unhidden after its last use</u>
0, 1	3
2	6
3	9
4, 5	12
6-8	17
9-13	23
14-24	29
25 or more	31

It is possible to disable item hiding at design time. To prevent a specific action (and all the collections containing it) from becoming hidden, find its *TActionClientItem* object and set its *UsageCount* to -1. To prevent hiding for an entire collection of items, such as the File menu or even the main menu bar, find the *TActionClients* object associated with the collection and set its *HideUnused* property to *False*.

- DevGuide: [Building applications with Delphi](#)

Using action lists

[Topic groups](#)

[See also](#)

Note: The contents of this section apply to setting up toolbars and menus for cross-platform development. For Windows development you can also use the methods described here. However, using action bands instead is simpler and offers more options. The action lists will be handled automatically by the Action Manager. See [Organizing actions for toolbars and menus](#) for details.

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.

- DevGuide: [Building applications with Delphi](#)

Setting up action lists

[Topic groups](#)

[See also](#)

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.
 - 1 Use one of the predefined actions listed in the editor: right-click and choose New Standard Action.
 - 2 The predefined actions are organized into categories (such as Dataset, Edit, Help, and Window) in the Standard Action Classes dialog box. Select all the standard actions you want to add to the action list and click OK.or
 - 3 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](#) 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. See [What happens when an action fires](#) for details.

■ DevGuide: [Building applications with Delphi](#)

What happens when an action fires

[Topic groups](#)

[See also](#)

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](#) “Working with events and event handlers” on page 25.

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, Delphi 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, Delphi 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](#) for details on how the application locates a target that can respond to a predefined action class.

■ DevGuide: [Building applications with Delphi](#)

How actions find their targets

[Topic groups](#)

[See also](#)

What happens when an action fires 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.

- DevGuide: [Building applications with Delphi](#)

Updating actions

[Topic groups](#)

[See also](#)

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.

■ DevGuide: [Building applications with Delphi](#)

Predefined action classes

[Topic groups](#)

[See also](#)

You can add predefined actions to your application by right-clicking on the Action Manager and choosing New Standard Action. The New Standard Action Classes dialog box is displayed listing the predefined action classes and the associated standard actions. These are actions that are included with Delphi and they are objects that automatically perform actions. The predefined actions are organized within the following classes:

<u>Class</u>	<u>Description</u>
Edit	Standard edit actions: Used with an edit control target. <i>TEditAction</i> is the base class for descendants that each override the <i>ExecuteTarget</i> method to implement copy, cut, and paste tasks by using the clipboard.
Format	Standard formatting actions: Used with rich text to apply text formatting options such as bold, italic, underline, strikethrough, and so on. <i>TRichEditAction</i> is the base class for descendants that each override the <i>ExecuteTarget</i> and <i>UpdateTarget</i> methods to implement formatting of the target.
Help	Standard Help actions: Used with any target. <i>THelpAction</i> is the base class for descendants that each override the <i>ExecuteTarget</i> method to pass the command onto a Help system.
Window	Standard window actions: Used with forms as targets in an MDI application. <i>TWindowAction</i> is the base class for descendants that each override the <i>ExecuteTarget</i> method to implement arranging, cascading, closing, tiling, and minimizing MDI child forms.
File	File actions: Used with operations on files such as File Open, File Run, or File Exit.
Search	Search actions: Used with search options. <i>TSearchAction</i> implements the common behavior for actions that display a modeless dialog where the user can enter a search string for searching an edit control.
Tab	Tab control actions: Used to move between tabs on a tab control such as the Prev and Next buttons on a wizard.
List	List control actions: Used for managing items in a list view.
Dialog	Dialog actions: Used with dialog components. <i>TDialogAction</i> implements the common behavior for actions that display a dialog when executed. Each descendant class represents a specific dialog.
Internet	Internet actions: Used for functions such as Internet browsing, downloading, and sending mail.
DataSet	DataSet actions: Used with a dataset component target. <i>TDataSetAction</i> is the base class for descendants that each override the <i>ExecuteTarget</i> and <i>UpdateTarget</i> methods to implement navigation and editing of the target. <i>TDataSetAction</i> introduces a <i>DataSource</i> property that ensures actions are performed on that dataset. If <i>DataSource</i> is nil, the currently focused data-aware control is used.
Tools	Tools: Additional tools such as <i>TCustomizeActionBars</i> for automatically displaying the customization dialog for action bands.

All of the action objects are described under the action object names in the online reference Help.

- DevGuide: [Building applications with Delphi](#)

Writing action components

[Topic groups](#)

[See also](#)

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:

<u>Method</u>	<u>Description</u>
<i>HandlesTarget</i>	Called automatically when the user invokes an object (such as a toolbutton or menu item) that is linked to the action. The <i>HandlesTarget</i> method lets the action object indicate whether it is appropriate to execute at this time with the object specified by the <i>Target</i> parameter as a "target". See How actions find their targets for details.
<i>UpdateTarget</i>	Called automatically when the application is idle so that actions can update themselves according to current conditions. Use in place of <i>OnUpdateAction</i> . See Updating actions for details.
<i>ExecuteTarget</i>	Called automatically when the action fires in response to a user action in place of <i>OnExecute</i> (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 for details.

When you write your own action classes, it is important to understand the following:

- [How actions find their targets](#)
- [Registering actions](#)

■ DevGuide: [Building applications with Delphi](#)

Registering actions

[Topic groups](#)

[See also](#)

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.

- DevGuide: [Building applications with Delphi](#)

Creating and managing menus

[Topic groups](#)

[See also](#)

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 section 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](#)

For information about hooking up menu items to the code that executes when they are selected, see [Associating menu events with event handlers](#).

■ DevGuide: [Building applications with Delphi](#)

Opening the Menu Designer

[Topic groups](#)

[See also](#)

You design menus for your application using the Menu Designer. Before you can start using the Menu Designer, first add either a MainMenu or PopupMenu component to your form. Both menu components are located on the Standard page of the component palette.



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 appears, with the first (blank) menu item highlighted in the Designer, and the *Caption* property selected in the Object Inspector.

Building menus

[Topic groups](#)

[See also](#)

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.

For more information about menu templates, see [Using menu templates](#).

For more information about creating a menu using the menu designer see



[Naming menus](#)



[Naming the menu items](#)



[Adding, inserting, and deleting menu items](#)



[Creating submenus](#)



[Adding images to menu items](#)



[Viewing the menu](#)

Naming menus

[Topic groups](#)

[See also](#)

As with all components, when you add a menu component to the form, Delphi gives it a default name; for example, *MainMenu1*. You can give the menu a more meaningful name that follows Object Pascal naming conventions.

Delphi adds the menu name to the form's type declaration, and the menu name then appears in the Component list.

Naming the menu items

[Topic groups](#)

[See also](#)

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 Delphi derive the *Name* property from the caption.

For example, if you give a menu item a *Caption* property value of *File*, Delphi assigns the menu item a *Name* property of *File1*. If you fill in the *Name* property before filling in the *Caption* property, Delphi 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, Delphi modifies the *Name* property accordingly. For example, if you want the caption to start with a number, Delphi 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.

<u>Component caption</u>	<u>Derived name</u>	<u>Explanation</u>
&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, Delphi 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

[Topic groups](#)

[See also](#)

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 Delphi has filled in the *Name* property based on the value you entered for the caption. (See [Naming the menu items](#).)

- 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.

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 add a separator bar

 Add a menu item as described above and type a hyphen (-) for the caption.

To add accelerators or shortcuts to menu items

 see [Specifying accelerator keys and keyboard shortcuts](#)

Specifying accelerator keys and keyboard shortcuts

[Topic groups](#)

[See also](#)

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.

Delphi 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. You can turn off this automatic checking by setting the *AutoHotkeys* property of a menu item to *maManual*.

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

[Topic groups](#)

[See also](#)

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. Delphi 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.)

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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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 an image to a menu item 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 in this topic group.
- 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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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](#).)

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.

<u>Menu command</u>	<u>Action</u>
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.
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

[Topic groups](#)

[See also](#)

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.

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

[Topic groups](#)

[See also](#)

Delphi 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 Delphi are stored in the BIN subdirectory in a default installation. These files have a .DMT (Delphi menu template) extension.

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.

- 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*.

Delphi deletes the template from the templates list and from your hard disk.

Saving a menu as a template

[Topic groups](#)

[See also](#)

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 your BIN subdirectory as .DMT files.

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.

- 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

[Topic groups](#)

[See also](#)

When you save a menu as a template, Delphi 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, Delphi 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, Delphi names it *File1*. If you insert it into a menu with an existing menu item named *File1*, Delphi names it *File2*.

Delphi 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, Delphi still generates the event handler name.

You can easily associate items in the menu template with existing *OnClick* event handlers in the form. For more information, see [Associating an event with an existing event handler](#).

Manipulating menu items at runtime

[Topic groups](#)

[See also](#)

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](#).

In multiple document interface (MDI) and Object Linking and Embedding (OLE) applications, you can also merge menu items into an existing menu bar. See [Merging menus](#) for more information.

Merging menus

[Topic groups](#)

[See also](#)

For MDI applications, such as the text editor sample application, and for OLE client applications, your application's main menu needs to be able to receive menu items either from another form or from the OLE server object. This is often called *merging menus*. Note that OLE technology is limited to Windows applications only and is not available for use in cross-platform programming.

You prepare menus for merging by specifying values for two properties:

 *Menu*, a property of the form

 *GroupIndex*, a property of menu items in the menu

Specifying the active menu: **Menu** property

[Topic groups](#)

[See also](#)

The *Menu* property specifies the active menu for the form. Menu-merging operations apply only to the active menu. If the form contains more than one menu component, you can change the active menu at runtime by setting the *Menu* property in code. For example,

```
Form1.Menu := SecondMenu;
```

Determining the order of merged menu items: *GroupIndex* property

[Topic groups](#)

[See also](#)

The *GroupIndex* property determines the order in which the merging menu items appear in the shared menu bar. Merging menu items can replace those on the main menu bar, or can be inserted.

The default value for *GroupIndex* is 0. Several rules apply when specifying a value for *GroupIndex*:



Lower numbers appear first (farther left) in the menu.

For instance, set the *GroupIndex* property to 0 (zero) for a menu that you always want to appear leftmost, such as a File menu. Similarly, specify a high number (it needn't be in sequence) for a menu that you always want to appear rightmost, such as a Help menu.



To replace items in the main menu, give items on the child menu the same *GroupIndex* value.

This can apply to groupings or to single items. For example, if your main form has an Edit menu item with a *GroupIndex* value of 1, you can replace it with one or more items from the child form's menu by giving them a *GroupIndex* value of 1 as well.

Giving multiple items in the child menu the same *GroupIndex* value keeps their order intact when they merge into the main menu.



To insert items without replacing items in the main menu, leave room in the numeric range of the main menu's items and "plug in" numbers from the child form.

For example, number the items in the main menu 0 and 5, and insert items from the child menu by numbering them 1, 2, 3, and 4.

Importing resource files

[Topic groups](#)

[See also](#)

Delphi supports use of menus built with other applications, so long as they are in the standard Windows resource (.RC) file format. You can import such menus directly into your Delphi project, saving you the time and effort of rebuilding menus that you created elsewhere.

To load existing .RC menu files,

- 1 In the Menu Designer, place your cursor where you want the menu to appear.
The imported menu can be part of a menu you are designing, or an entire menu in itself.
- 2 Right-click and choose Insert From Resource.
The Insert Menu From Resource dialog box appears.
- 3 In the dialog box, select the resource file you want to load, and choose OK.
The menu appears in the Menu Designer window.

Note: If your resource file contains more than one menu, you first need to save each menu as a separate resource file before importing it.

Designing toolbars and cool bars

[Topic groups](#)

[See also](#)

A *toolbar* is a panel, usually across the top of a form (under the menu bar), that holds buttons and other controls. A *cool bar* (also called a rebar) is a kind of toolbar that displays controls on movable, resizable bands. If you have multiple panels aligned to the top of the form, they stack vertically in the order added.

Note: Cool bars are not available in CLX for cross-platform applications.

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.

You can add a toolbar to a form in several ways:



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.



Use a cool bar (*TCoolBar*) component and add controls to it. The cool bar displays controls on independently movable and resizable bands.

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 and cool bar components, you are ensuring that your application has the look and feel of a Windows application because you are using the native Windows controls. If these operating system controls change in the future, your application could change as well. Also, since the toolbar and cool bar rely on common components in Windows, your application requires the COMCTL32.DLL. Toolbars and cool bars are not supported in WinNT 3.51 applications.

The following sections describe how to



[Adding a toolbar using a panel component](#)



[Adding a toolbar using the toolbar component](#)



[Adding a cool bar component](#)



[Responding to clicks](#)



[Adding hidden toolbars](#)



[Hiding and showing toolbars](#)

Adding a toolbar using a panel component

[Topic groups](#)

[See also](#)

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:

-  [Adding a speed button to a panel](#)
-  [Assigning a speed button's glyph](#)
-  [Setting the initial condition of a speed button](#)
-  [Creating a group of speed buttons](#)
-  [Allowing toggle buttons](#)

Adding a speed button to a panel

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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.

The table below lists some actions you can set to change a speed button's appearance:

To make a speed button:	Set the toolbar's:
Appear pressed	<i>GroupIndex</i> property to a value other than zero and its <i>Down</i> property to <i>True</i> .
Appear disabled	<i>Enabled</i> property to <i>False</i> .
Have a left margin	<i>Indent</i> 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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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 (from the Win32 page of the component palette). 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:



[Adding a tool button](#)



[Assigning images to tool buttons](#)



[Setting tool button appearance and initial conditions](#)



[Creating groups of tool buttons](#)



[Allowing toggled tool buttons](#)

Adding a tool button

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

The table below lists some actions you can set to change a tool button's appearance:

<u>To make a tool button:</u>	<u>Set the toolbar's:</u>
Appear pressed	(on tool button) <i>Style</i> property to <i>tbsCheck</i> and <i>Down</i> property to <i>True</i> .
Appear disabled	<i>Enabled</i> property to <i>False</i> .
Have a left margin	<i>Indent</i> property to a value greater than 0.
Appear to have "pop-up" borders, thus making the toolbar appear transparent	<i>Flat</i> property to <i>True</i> .

Note: Using the *Flat* property of *TToolBar* requires version 4.70 or later of COMCTL32.DLL.

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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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.

Adding a cool bar component

[Topic groups](#)

[See also](#)

The cool bar component (*TCoolBar*)—also called a *rebar*—displays windowed controls on independently movable, resizable bands. The user can position the bands by dragging the resizing grips on the left side of each band.

To add a cool bar to a form in a Windows application,

- 1 Add a cool bar component to the form (from the Win32 page of the component palette). The cool bar automatically aligns to the top of the form.
- 2 Add windowed controls from the component palette to the bar.

Only VCL components that descend from *TWinControl* are windowed controls. You can add graphic controls—such as labels or speed buttons—to a cool bar, but they will not appear on separate bands.

Setting the appearance of the cool bar

[Topic groups](#)

The cool bar component offers several useful configuration options. The table below lists some actions you can set to change a tool button's appearance:

To make the cool bar:

Resize automatically to accommodate the bands it contains

Bands maintain a uniform height

Reorient to vertical rather than horizontal

Prevent the *Text* properties of the bands from displaying at runtime

Remove the border around the bar

Keep users from changing the bands' order at runtime. (The user can still move and resize the bands.)

Create a background image for the cool bar

Choose a list of images to appear on the left of any band

To assign images to individual bands, select the cool bar and double-click on the *Bands* property in the Object Inspector. Then select a band and assign a value to its *ImageIndex* property.

Set the toolbar's:

AutoSize property to *True*.

FixedSize property to *True*.

Vertical property to *True*. This changes the effect of the *FixedSize* property.

ShowText property to *False*. Each band in a cool bar has its own *Text* property.

BandBorderStyle to *bsNone*.

FixedOrder to *True*.

Bitmap property to *TBitmap* object.

Images property to *TImageList* object.

Responding to clicks

[Topic groups](#)

[See also](#)

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](#) and [Generating a handler for a component's default event](#).

Assigning a menu to a tool button

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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, cool bar, 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

[Topic groups](#)

[See also](#)

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;
```

 DevGuide: Building applications with Delphi

Demo programs

[Topic groups](#)

[See also](#)

For examples of Windows applications that use actions and action lists, refer to Demos\RichEdit. In addition, the Application wizard (File|New Project page), MDI Application, SDI Application, and Winx Logo Applications can use the action and action list objects. For examples of cross-platform applications, refer to Demos\CLX.

Implementing drag-and-drop in controls

[Topic groups](#)

[See also](#)

Drag-and-drop is often a convenient way for users to manipulate objects. You can let users drag an entire control, or let them drag items from one control—such as a list box or tree view—into another.



[Starting a drag operation](#)



[Accepting dragged items](#)



[Dropping items](#)



[Ending a drag operation](#)



[Customizing drag and drop with a drag object](#)



[Changing the drag mouse pointer](#)

Starting a drag operation

[See also](#)

Every control has a property called *DragMode* that determines how drag operations are initiated. If *DragMode* is *dmAutomatic*, dragging begins automatically when the user presses a mouse button with the cursor on the control. Because *dmAutomatic* can interfere with normal mouse activity, you may want to set *DragMode* to *dmManual* (the default) and start the dragging by handling mouse-down events.

To start dragging a control manually, call the control's *BeginDrag* method. *BeginDrag* takes a Boolean parameter called *Immediate* and, optionally, an integer parameter called *Threshold*. If you pass *True* for *Immediate*, dragging begins immediately. If you pass *False*, dragging does not begin until the user moves the mouse the number of pixels specified by *Threshold*. Calling

```
BeginDrag False)
```

allows the control to accept mouse clicks without beginning a drag operation.

You can place other conditions on whether to begin dragging, such as checking which mouse button the user pressed, by testing the parameters of the mouse-down event before calling *BeginDrag*. The following code, for example, handles a mouse-down event in a file list box by initiating a drag operation only if the left mouse button was pressed.

```
procedure TFMForm.FileListBox1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then { drag only if left button pressed }
    with Sender as TFileListBox do { treat Sender as TFileListBox }
      begin
        if ItemAtPos(Point(X, Y), True) >= 0 then { is there an item here? }
          BeginDrag(False); { if so, drag it }
        end;
      end;
end;
```

Accepting dragged items

[See also](#)

When the user drags something over a control, that control receives an *OnDragOver* event, at which time it must indicate whether it can accept the item if the user drops it there. The drag cursor changes to indicate whether the control can accept the dragged item. To accept items dragged over a control, attach an event handler to the control's *OnDragOver* event.

The drag-over event has a parameter called *Accept* that the event handler can set to *True* if it will accept the item. If *Accept* is *True*, the application sends a drag-and-drop event to the control.

The drag-over event has other parameters, including the source of the dragging and the current location of the mouse cursor, that the event handler can use to determine whether to accept the drop. In the following example, a directory tree view accepts dragged items only if they come from a file list box.

```
procedure TFMForm.DirectoryOutline1DragOver(Sender, Source: TObject; X,  
    Y: Integer; State: TDragState; var Accept: Boolean);  
begin  
    if Source is TFileListBox then  
        Accept := True  
    else  
        Accept := False;  
end;
```

Dropping items

[See also](#)

If a control indicates that it can accept a dragged item, it needs to handle the item should it be dropped. To handle dropped items, attach an event handler to the *OnDragDrop* event of the control accepting the drop. Like the drag-over event, the drag-and-drop event indicates the source of the dragged item and the coordinates of the mouse cursor over the accepting control. The latter parameter allows you to monitor the path an item takes while being dragged; you might, for example, want to use this information to change the color of components as they are passed over.

In the following example, a directory tree view, accepting items dragged from a file list box, responds by moving files to the directory on which they are dropped.

```
procedure TFMForm.DirectoryOutline1DragDrop(Sender, Source: TObject; X,  
    Y: Integer);  
begin  
    if Source is TFileListBox then  
        with DirectoryOutline1 do  
            ConfirmChange('Move', FileListBox1.FileName, Items[GetItem(X, Y)].FullPath);  
end;
```

Ending a drag operation

[Topic groups](#)

[See also](#)

A drag operation ends when the item is either successfully dropped or released over a control that cannot accept it. At this point an end-drag event is sent to the control from which the item was dragged. To enable a control to respond when items have been dragged from it, attach an event handler to the control's *OnEndDrag* event.

The most important parameter in an *OnEndDrag* event is called *Target*, which indicates which control, if any, accepts the drop. If *Target* is *nil*, it means no control accepts the dragged item. The *OnEndDrag* event also includes the coordinates on the receiving control.

In this example, a file list box handles an end-drag event by refreshing its file list.

```
procedure TFMForm.FileListBox1EndDrag(Sender, Target: TObject; X, Y: Integer);  
begin  
    if Target <> nil then FileListBox1.Update;  
end;
```

Customizing drag and drop with a drag object

[See also](#)

You can use a *TDragObject* descendant to customize an object's drag-and-drop behavior. The standard drag-over and drag-and-drop events indicate the source of the dragged item and the coordinates of the mouse cursor over the accepting control. To get additional state information, derive a custom drag object from *TDragObject* or *TDragObjectEx* and override its virtual methods. Create the custom drag object in the *OnStartDrag* event.

Normally, the source parameter of the drag-over and drag-and-drop events is the control that starts the drag operation. If different kinds of control can start an operation involving the same kind of data, the source needs to support each kind of control. When you use a descendant of *TDragObject*, however, the source is the drag object itself; if each control creates the same kind of drag object in its *OnStartDrag* event, the target needs to handle only one kind of object. The drag-over and drag-and-drop events can tell if the source is a drag object, as opposed to the control, by calling the *IsDragObject* function.

TDragObjectEx descendants are freed automatically whereas descendants of *TDragObject* are not. If you have *TDragObject* descendants that you are not explicitly freeing, you can change them so they descend from *TDragObjectEx* instead to prevent memory loss.

Drag objects let you drag items between a form implemented in the application's main executable file and a form implemented using a DLL, or between forms that are implemented using different DLLs.

 DevGuide: [Building applications with Delphi](#)

Changing the drag mouse pointer

[See also](#)

You can customize the appearance of the mouse pointer during drag operations by setting the source component's *DragCursor* property (VCL only).

Implementing drag-and-dock in controls

[Topic groups](#)

[See also](#)

Note: Drag and dock properties are available in the VCL but not CLX.

Descendants of *TWinControl* can act as docking sites and descendants of *TControl* can act as child windows that are docked into docking sites. For example, to provide a docking site at the left edge of a form window, align a panel to the left edge of the form and make the panel a docking site. When dockable controls are dragged to the panel and released, they become child controls of the panel.



[Making a windowed control a docking site](#)



[Making a control a dockable child](#)



[Controlling how child controls are docked](#)



[Controlling how child controls are undocked](#)



[Controlling how child controls respond to drag-and-dock operations](#)

Making a windowed control a docking site

[Topic groups](#)

[See also](#)

Note: Drag and dock properties are available in the VCL but not CLX.

To make a windowed control a docking site,

- 1 Set the *DockSite* property to *True*.
- 2 If the dock site object should not appear except when it contains a docked client, set its *AutoSize* property to *True*. When *AutoSize* is *True*, the dock site is sized to 0 until it accepts a child control for docking. Then it resizes to fit around the child control.

Making a control a dockable child

[Topic groups](#)

[See also](#)

Note: Drag and dock properties are available in the VCL but not CLX.

To make a control a dockable child,

- 1 Set its *DragKind* property to *dkDock*. When *DragKind* is *dkDock*, dragging the control moves the control to a new docking site or undocks the control so that it becomes a floating window. When *DragKind* is *dkDrag* (the default), dragging the control starts a drag-and-drop operation which must be implemented using the *OnDragOver*, *OnEndDrag*, and *OnDragDrop* events.
- 2 Set its *DragMode* to *dmAutomatic*. When *DragMode* is *dmAutomatic*, dragging (for drag-and-drop or docking, depending on *DragKind*) is initiated automatically when the user starts dragging the control with the mouse. When *DragMode* is *dmManual*, you can still begin a drag-and-dock (or drag-and-drop) operation by calling the *BeginDrag* method.
- 3 Set its *FloatingDockSiteClass* property to indicate the *TWinControl* descendant that should host the control when it is undocked and left as a floating window. When the control is released and not over a docking site, a windowed control of this class is created dynamically, and becomes the parent of the dockable child. If the dockable child control is a descendant of *TWinControl*, it is not necessary to create a separate floating dock site to host the control, although you may want to specify a form in order to get a border and title bar. To omit a dynamic container window, set *FloatingDockSiteClass* to the same class as the control, and it will become a floating window with no parent.

Controlling how child controls are docked

[Topic groups](#)

[See also](#)

Note: Drag and dock properties are available in the VCL but not CLX.

A docking site automatically accepts child controls when they are released over the docking site. For most controls, the first child is docked to fill the client area, the second splits that into separate regions, and so on. Page controls dock children into new tab sheets (or merge in the tab sheets if the child is another page control).

Three events allow docking sites to further constrain how child controls are docked:

```
property OnGetSiteInfo: TGetSiteInfoEvent;  
TGetSiteInfoEvent = procedure (Sender: TObject; DockClient: TControl; var  
InfluenceRect: TRect; var CanDock: Boolean) of object;
```

OnGetSiteInfo occurs on the docking site when the user drags a dockable child over the control. It allows the site to indicate whether it will accept the control specified by the *DockClient* parameter as a child, and if so, where the child must be to be considered for docking. When *OnGetSiteInfo* occurs, *InfluenceRect* is initialized to the screen coordinates of the docking site, and *CanDock* is initialized to *True*. A more limited docking region can be created by changing *InfluenceRect* and the child can be rejected by setting *CanDock* to *False*.

```
property OnDockOver: TDockOverEvent;  
TDockOverEvent = procedure (Sender: TObject; Source: TDragDockObject; X, Y: Integer;  
State: TDragState; var Accept: Boolean) of object;
```

OnDockOver occurs on the docking site when the user drags a dockable child over the control. It is analogous to the *OnDragOver* event in a drag-and-drop operation. Use it to signal that the child can be released for docking, by setting the *Accept* parameter. If the dockable control is rejected by the *OnGetSiteInfo* event handler (perhaps because it is the wrong type of control), *OnDockOver* does not occur.

```
property OnDockDrop: TDockDropEvent;  
TDockDropEvent = procedure (Sender: TObject; Source: TDragDockObject; X, Y: Integer)  
of object;
```

OnDockDrop occurs on the docking site when the user releases the dockable child over the control. It is analogous to the *OnDragDrop* event in a normal drag-and-drop operation. Use this event to perform any necessary accommodations to accepting the control as a child control. Access to the child control can be obtained using the *Control* property of the *TDockObject* specified by the *Source* parameter.

Controlling how child controls are undocked

[Topic groups](#)

[See also](#)

Note: Drag and dock properties are available in the VCL but not CLX.

A docking site automatically allows child controls to be undocked when they are dragged and have a *DragMode* property of *dmAutomatic*. Docking sites can respond when child controls are dragged off, and even prevent the undocking, in an *OnUnDock* event handler:

```
property OnUnDock: TUnDockEvent;  
TUnDockEvent = procedure(Sender: TObject; Client: TControl; var Allow: Boolean) of  
object;
```

The *Client* parameter indicates the child control that is trying to undock, and the *Allow* parameter lets the docking site (*Sender*) reject the undocking. When implementing an *OnUnDock* event handler, it can be useful to know what other children (if any) are currently docked. This information is available in the read-only *DockClients* property, which is an indexed array of *TControl*. The number of dock clients is given by the read-only *DockClientCount* property.

Controlling how child controls respond to drag-and-dock operations

[Topic groups](#)

[See also](#)

Note: Drag and dock properties are available in the VCL but not CLX.

Dockable child controls have two events that occur during drag-and-dock operations: *OnStartDock*, analogous to the *OnStartDrag* event of a drag-and-drop operation, allows the dockable child control to create a custom drag object. *OnEndDock*, like *OnEndDrag*, occurs when the dragging terminates.

Working with text in controls

[Topic groups](#)

The following topics how to use various features of rich 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

[Topic groups](#)

[See also](#)

In a rich edit or 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.

For example, the following code attaches an *OnClick* event handler to the *Character|Left* menu item, then attaches the same event handler to both the *Right* and *Center* menu items on the *Character* menu.

```
procedure TEditForm.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

[Topic groups](#)

[See also](#)

Rich edit 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 rich edit or memo component's *ScrollBars* property to include or exclude scroll bars.

The following example attaches an *OnClick* event handler to a Character|WordWrap menu item.

```
procedure TEditForm.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;
```

The rich edit and memo components handle their scroll bars in a slightly different way. The rich edit component can hide its scroll bars if the text fits inside the bounds of the component. The memo always shows scroll bars if they are enabled.

Adding the clipboard object

[Topic groups](#)

[See also](#)

Most text-handling applications provide users with a way to move selected text between documents, including documents in different applications. The *Clipboard* object in Delphi encapsulates a clipboard (such as the Windows Clipboard) and includes methods for cutting, copying, and pasting text (and other formats, including graphics). The *Clipboard* object is declared in the *Clipbrd* unit.

To add the *Clipboard* object to an application,

- 1 Select the unit that will use the clipboard.
- 2 Search for the **implementation** reserved word.
- 3 Add *Clipbrd* to the **uses** clause below **implementation**.



If there is already a **uses** clause in the **implementation** part, add *Clipbrd* to the end of it.



If there is not already a **uses** clause, add one that says

```
uses Clipbrd;
```

For example, in an application with a child window, the **uses** clause in the unit's implementation part might look like this:

```
uses  
  MDIframe, Clipbrd;
```

Selecting text

[Topic groups](#)

[See also](#)

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.

The table below lists properties commonly used to handle selected text.

Property	Description
<i>SelText</i>	Contains a string representing the selected text in the component.
<i>SelLength</i>	Contains the length of a selected string.
<i>SelStart</i>	Contains the starting position of a string.

Selecting all text

[Topic groups](#)

The *SelectAll* method selects the entire contents of the rich edit or 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 rich edit or memo control, call the *RichEdit1* control's *SelectAll* method.

For example,

```
procedure TMainForm.SelectAll(Sender: TObject);  
begin  
    RichEdit1.SelectAll; { select all text in RichEdit }  
end;
```

Cutting, copying, and pasting text

[Topic groups](#)

[See also](#)

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.

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 TEditForm.CutToClipboard(Sender: TObject);
begin
    Editor.CutToClipboard;
end;
procedure TEditForm.CopyToClipboard(Sender: TObject);
begin
    Editor.CopyToClipboard;
end;
procedure TEditForm.PasteFromClipboard(Sender: TObject);
begin
    Editor.PasteFromClipboard;
end;
```

Deleting selected text

[Topic groups](#)

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 TEditForm.Delete(Sender: TObject);  
begin  
    RichEdit1.ClearSelection;  
end;
```

Disabling menu items

[Topic groups](#)

[See also](#)

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 `RichEdit1` has selected text. The Paste command is enabled or disabled based on whether any text exists on the clipboard.

```
procedure TEditForm.Edit1Click(Sender: TObject);  
var  
    HasSelection: Boolean; { declare a temporary variable }  
begin  
    Paste1.Enabled := Clipboard.HasFormat(CF_TEXT); {enable or disable the Paste  
menu item}  
    HasSelection := Editor.SelLength > 0; { True if text is selected }  
    Cut1.Enabled := HasSelection; { enable menu items if HasSelection is True }  
    Copy1.Enabled := HasSelection;  
    Delete1.Enabled := HasSelection;  
end;
```

The `HasFormat` method of the clipboard returns a Boolean value based on whether the clipboard contains objects, text, or images of a particular format. By calling `HasFormat` with the parameter `CF_TEXT`, you can determine whether the clipboard contains any text, and enable or disable the Paste item as appropriate.

Providing a pop-up menu

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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.

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](#) 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 TEditForm.Edit1Click(Sender: TObject);  
var  
    HasSelection: Boolean;  
begin  
    Paste1.Enabled := Clipboard.HasFormat(CF_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

[Topic groups](#)

[See also](#)

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 tool bars. Instead of using the standard method of drawing a 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.](#)

All owner-draw controls 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 a list to make it easy to use that object when drawing items.

In general, creating an owner-draw control in Delphi involves these steps:

- 1 [Indicating that a control is owner-drawn](#)
- 2 [Adding graphical objects to a string list](#)
- 3 [Drawing owner-drawn items](#)

Indicating that a control is owner-drawn

[Topic groups](#)

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. List views and 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 the following table 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.

<u>Owner-draw style</u>	<u>Meaning</u>	<u>Examples</u>
Fixed	Each item is the same height, with that height determined by the <i>ItemHeight</i> property.	<i>lbOwnerDrawFixed</i> , <i>csOwnerDrawFixed</i>
Variable	Each item might have a different height, determined by the data at runtime.	<i>lbOwnerDrawVariable</i> , <i>csOwnerDrawVariable</i>

Adding graphical objects to a string list

[Topic groups](#)

[See also](#)

Every string list has the ability to hold a list of objects in addition to its list of strings.

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.

Adding images to an application

[Topic groups](#)

[See also](#)

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 to the desired bitmap using the Picture editor from the Object Inspector.

The image controls are invisible when you run the application.

Adding images to a string list

[Topic groups](#)

[See also](#)

Once you have graphical images in an application, you can associate them with the strings in a string list. You can either add the objects at the same time as the strings, or associate objects with existing strings. The preferred method is to add objects and strings at the same time, if all the needed data is available.

The following example shows how you might want to add images to a string list. This is part of a file manager application where, along with a letter for each valid drive, it adds a bitmap indicating each drive's type. The *OnCreate* event handler looks like this:

```
procedure TFMForm.FormCreate(Sender: TObject);
var
  Drive: Char;
  AddedIndex: Integer;
begin
  for Drive := 'A' to 'Z' do { iterate through all possible drives }
  begin
    case GetDriveType(Drive + ':/') of { positive values mean valid drives }
      DRIVE_REMOVABLE: { add a tab }
        AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Floppy.Picture.Graphic);
      DRIVE_FIXED: { add a tab }
        AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Fixed.Picture.Graphic);
      DRIVE_REMOTE: { add a tab }
        AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Network.Picture.Graphic);
    end;
    if UpCase(Drive) = UpCase(DirectoryOutline.Drive) then { current drive? }
      DriveTabSet.TabIndex := AddedIndex; { then make that current tab }
    end;
  end;
end;
```

Drawing owner-drawn items

Topic groups

When you indicate that a control is owner-drawn, either by setting a property or supplying a custom draw event handler, the control is no longer drawn on the screen. Instead, the operating system generates events for each visible item in the control. Your application handles the events to draw the items.

To draw the items in an owner-draw control, do the following for each visible item in the control. Use a single event handler for all items.

- 1 Size the item, if needed.
Items of the same size (for example, with a list box style of *IsOwnerDrawFixed*), do not require sizing.
- 2 Draw the item.

Sizing owner-draw items

[Topic groups](#)

Before giving your application the chance to draw each item in a variable owner-draw control, the operating system generates a measure-item event. The measure-item event tells the application where the item appears on the control.

Delphi determines the size of 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 be the size of the bitmap. If you want a bitmap and text, adjust the rectangle to be big 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. Depending on the control, the name of the event can vary. List boxes and combo boxes use *OnMeasureItem*. Grids have no measure-item event.

The sizing event has two important parameters: the index number of the item and the size of that item. The size 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.

Owner-draw grids cannot change the sizes of their cells as they draw. The size of each row and column is set before drawing by the *ColWidths* and *RowHeights* properties.

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

Topic groups

When an application needs to draw or redraw an owner-draw control, the operating system 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 subitems.

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*



OnAdvancedCustomDraw, such as *OnAdvancedCustomDrawItem*

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;
```

 DevGuide: [Building applications with Delphi](#)

Working with graphics and multimedia

[Topic groups](#)

Graphics and multimedia elements can add polish to your applications. Delphi offers a variety of ways to introduce these features 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. To add multimedia capabilities, Delphi includes special components that can play audio and video clips. Note that multimedia components are not available for cross-platform programming.

The following topics describe how to enhance your applications by introducing graphics or multimedia elements:

 [Overview of graphics programming](#)

 [Working with multimedia](#)

Overview of graphics programming

[Topic groups](#)

[See also](#)

The VCL graphics components defined in the Graphics unit encapsulate the Windows Graphics Device Interface (GDI), making it easy to add graphics to your Windows applications. CLX graphics components defined in the QGraphics unit encapsulate the Qt graphics widgets for adding graphics to cross-platform applications.

To draw graphics in a Delphi 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 it takes care of device context, so your programs can use the same methods regardless of whether you are drawing on the screen, to a printer, or on bitmaps or metafiles (drawings in CLX). Canvases are available only at runtime, so you do all your work with canvases by writing code.

VCL Note: Since *TCanvas* is a wrapper resource manager around the Windows device context, you can also use all Windows GDI functions on the canvas. The *Handle* property of the canvas *is* the device context Handle.

CLX 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* message (VCL) or event (CLX).

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 following topics describe how to use graphics components to simplify your coding.

-  [Refreshing the screen](#)
-  [Types of graphic objects](#)
-  [Common properties and methods of canvases](#)
-  [Handling multiple drawing objects in an application](#)
-  [Drawing on a bitmap](#)
-  [Loading and saving graphics files](#)
-  [Using the clipboard with graphics](#)
-  [Rubber banding example](#)

Refreshing the screen

[Topic groups](#)

[See also](#)

At certain times, the operating system determines that objects onscreen need to refresh their appearance, so it generates WM_PAINT messages on Windows, which the VCL routes to *OnPaint* events. (If you are using CLX for cross-platform development, a paint event is generated, which CLX routes to *OnPaint* events.) If you have written an *OnPaint* event handler for that object, it is called 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 or if using the VCL, you want to paint a background on a form.

While some operating systems automatically handle the redrawing of the client area of a window that has been invalidated, Windows does not. In the Windows operating system anything drawn on the screen is permanent. 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. For more information about the WM_PAINT message, see the Windows online Help.

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. The *Picture* property specifies the actual bitmap, drawing, or other graphic object that *TImage* displays. You can also set the *Proportional* property to ensure that the image can be fully displayed in the image control without any distortion. 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 screen device (VCL) or the painter (CLX), 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

[Topic groups](#)

[See also](#)

The VCL/CLX provides the following graphic objects. These objects have methods to draw on the canvas, which are described in [Using Canvas methods to draw graphic objects](#) and to load and save to graphics files, as described in [Loading and saving graphics files](#)

<u>Object</u>	<u>Description</u>
Picture	Used to hold any graphic image. To add additional graphic file formats, use the Picture <i>Register</i> 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 <i>handle</i> 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 (:ICO file).
Metafile (VCL only)	Contains a file that records the operations required to construct an image, rather than contain the actual bitmap pixels of the image.
Drawing (CLX only)	Metafiles or drawings are extremely scalable without the loss of image detail and often require much less memory than bitmaps, particularly for high-resolution devices, such as printers. However, metafiles and drawings do not display as fast as bitmaps. Use a metafile or drawing when versatility or precision is more important than performance.

Common Properties and Methods of Canvas

[Topic groups](#)

[See also](#)

The following table lists the commonly used 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.
Pixels	Specifies the color of the area of pixels within the current ClipRect.

These properties are described in more detail in [Using the properties of the Canvas object](#).

Here is a list of several methods you can use:

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).
Ellipse	Draws the ellipse defined by a bounding rectangle on the canvas.
FillRect	Fills the specified rectangle on the canvas using the current brush.
FloodFill (VCL only)	Fills an area of the canvas using the current brush.
FrameRect	Draws a rectangle using the Brush of the canvas to draw the border.
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.
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 <i>Rectangle</i> 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.

These methods are described in more detail in [Using Canvas methods to draw graphic objects](#).

 DevGuide: [Building applications with Delphi](#)

Using the properties of the Canvas object

[Topic groups](#)

[See also](#)

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](#)



[Reading and setting pixels](#)

Using pens

[Topic groups](#)

[See also](#)

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.

You can use TPenRecall for quick saving off and restoring the properties of pens.

Changing the pen color

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

A pen's width determines the thickness, in pixels, of the lines it draws.

Note: When the thickness is greater than 1, Windows 95/98 always draw 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

[Topic groups](#)

[See also](#)

A pen's *Style* property allows you to set solid lines, dashed lines, dotted lines, and so on.

VCL Note: If developing a cross-platform application for deployment under Windows, 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*.

Delphi generates an empty click-event handler called *SetPenStyle* and attaches it to the *OnClick* events of all six buttons.

- 2 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

[Topic groups](#)

[See also](#)

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.

Getting the pen position

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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.

You can use [TBrushRecall](#) for quick saving off and restoring the properties of brushes.

Changing the brush color

[Topic groups](#)

[See also](#)

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 :

```
procedure TForm1.BrushColorClick(Sender: TObject);  
begin  
    Canvas.Brush.Color := BrushColor.ForegroundColor;  
end;
```

Changing the brush style

[Topic groups](#)

[See also](#)

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*, or *bsDiagCross*.

This example sets brush styles by sharing a click-event handler for a set of eight brush-style buttons. All eight 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

[Topic groups](#)

[See also](#)

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.

Reading and setting pixels

[Topic groups](#)

[See also](#)

You will notice that every canvas has an indexed [Pixels property](#) that represents the individual colored points that make up the image on the canvas. You rarely need to access *Pixels* directly, it is available only for convenience to perform small actions such as finding or setting a pixel's color.

Note: Setting and getting individual pixels is thousands of times slower than performing graphics operations on regions. Do not use the Pixel array property to access the image pixels of a general array. For high-performance access to image pixels, see the [TBitmap.ScanLine property](#).

 DevGuide: [Building applications with Delphi](#)

Using Canvas methods to draw graphic objects

[Topic groups](#)

[See also](#)

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](#)

 DevGuide: [Building applications with Delphi](#)

Drawing lines and polylines

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

[Example](#)

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.

Example: Drawing polylines

The following method 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 Delphi'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.

Drawing shapes

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

[Example](#)

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.

Example: Drawing rectangles and ellipses

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

[Topic groups](#)

[See also](#)

[Example](#)

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.

Example: Drawing rounded rectangles

The following method 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

[Topic groups](#)

[See also](#)

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.

Example: Drawing polygons

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;
```

 DevGuide: [Building applications with Delphi](#)

Handling multiple drawing objects in your application

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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 word `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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

[Example](#)

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.

Example: Sharing code among event handlers

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.FormMouseDown(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

[Topic groups](#)

[See also](#)

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 an 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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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](#) shows how to use graphics in controls.

Placing the control

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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 .

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

[Topic groups](#)

[See also](#)

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
    Bitmap.LoadFromFile('C:\Program Files\Borland\Delphi
4\Images\Splash\256color\factory.bmp');
    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;
    canvas.draw(0,0,Bitmap);
  finally
    Bitmap.free;
  end;
end;
```

Loading and saving graphics files

[Topic groups](#)

[See also](#)

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.

The components you use to load, save, and replace graphic images support many graphic formats including bitmap files, metafiles, glyphs, 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 these topics:

 [Loading a picture from a file](#)

 [Saving a picture to a file](#)

 [Replacing the picture](#)

Loading a picture from a file

[Topic groups](#)

[See also](#)

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 OpenPictureDialog1.Execute then
  begin
    CurrentFile := OpenPictureDialog1.FileName;
    Image.Picture.LoadFromFile(CurrentFile);
  end;
end;
```

Saving a picture to a file

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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, 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 such 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

[Topic groups](#)

[See also](#)

You can use the Windows clipboard to copy and paste graphics within your applications or to exchange graphics with other applications. The VCL's 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 Clipbrd (QClipbrd in CLX) unit to the **uses** clause of any unit that needs to access clipboard data.

For cross-platform applications, data that is stored on the clipboard when using CLX 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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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 [HasFormat method](#) (if using the VCL) or [Provides method](#) (if using CLX) to see whether the clipboard contains a graphic.
HasFormat (or *Provides* in CLX) is a Boolean function. It returns *True* if the clipboard contains an item of the type specified in the parameter. To test for graphics on the Windows platform, you pass `CF_BITMAP`. In cross-platform applications, you pass *SDelphiBitmap*.
- 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.HasFormat(CF_BITMAP) then { is there a bitmap on the Windows
clipboard? }
  begin
    Image1.Picture.Bitmap.Assign(Clipboard);
  end;
end;
```

The same example in CLX for cross-platform development would look as follows:

```
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, such as Microsoft Paint. 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

[Topic groups](#)

[See also](#)

This example describes 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 example code in this section is taken from a sample application located in the Demos\DOC\Graphexdirectory. 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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

The VCL has three mouse events: [OnMouseDown event](#), [OnMouseMove event](#), and [OnMouseUp event](#).

When an 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:

<u>Parameter</u>	<u>Meaning</u>
<i>Sender</i>	The object that detected the mouse action
<i>Button</i>	Indicates which mouse button was involved: <i>mbLeft</i> , <i>mbMiddle</i> , or <i>mbRight</i>
<i>Shift</i>	Indicates the state of the <i>Alt</i> , <i>Ctrl</i> , and <i>Shift</i> keys at the time of the mouse action
<i>X, Y</i>	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.

Note: Delphi uses the same criteria as Microsoft Windows in determining which mouse button has been pressed. Thus, if you have switched the default "primary" and "secondary" mouse buttons (so that the right mouse button is now the primary button), clicking the primary (right) button will record *mbLeft* as the value of the *Button* parameter.

Responding to a mouse-down action

[Topic groups](#)

[See also](#)

[Example](#)

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.

The VCL 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;
```

Example: Responding to a mouse-down action

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

[Topic groups](#)

[See also](#)

[Example](#)

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.

Example: Responding to a mouse-up action

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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

[Example](#)

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, Delphi 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.

Delphi "owns" any declarations before the **public** directive: that's where it puts the fields that represent controls and the methods that respond to events.

Example: Adding a field to a form object to track mouse actions

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 type TPoint.

```
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.

 DevGuide: [Building applications with Delphi](#)

Refining line drawing

[Topic groups](#)

[See also](#)

With fields in place to track various points, you can refine an application's line drawing.

Tracking the origin point

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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.

Working with multimedia

[Topic groups](#)

[See also](#)

Delphi allows you to add multimedia components to your applications. To do this, you can use either the [TAnimate](#) component on the Win32 page or the [TMediaPlayer](#) component on the System page of the Component palette. Use the animate component when you want to add silent video clips to your application. Use the media player component when you want to add audio and/or video clips to an application.

The following topics are discussed in this section:

-  [Adding silent video clips to an application](#)
-  [Adding audio and/or video clips to an application](#)

Adding silent video clips to an application

[Topic groups](#)

[See also](#)

The animation control in Delphi allows you to add silent video clips to your application.

To add a silent video clip to an application:

- 1 Double-click the animate icon on the Win32 page of the Component palette. This automatically puts an animation control on the form window in which you want to display the video clip.
- 2 Using the Object Inspector, select the *Name* property and enter a new *name* for your animation control. You will use this name when you call the animation control. (Follow the standard rules for naming Delphi identifiers).

Always work directly with the Object Inspector when setting design time properties and creating event handlers.

- 3 Do one of the following:



Select the *Common AVI* property and choose one of the AVIs available from the drop down list; or



Select the *FileName* property and click the ellipsis (...) button, choose an AVI file from any available local or network directories and click Open in the Open AVI dialog; or



Select the resource of an AVI using the *ResName* or *ResID* properties. Use *ResHandle* to indicate the module that contains the resource identified by *ResName* or *ResID*.

This loads the AVI file into memory. If you want to display the first frame of the AVI clip on-screen until it is played using the *Active* property or the *Play* method, then set the *Open* property to *True*.

- 4 Set the *Repetitions* property to the number of times you want to the AVI clip to play. If this value is 0, then the sequence is repeated until the *Stop* method is called.
- 5 Make any other changes to the *animation* control settings. For example, if you want to change the first frame displayed when animation control opens, then set the *StartFrame* property to the desired frame value.
- 6 Set the *Active* property to *True* using the drop down list or write an event handler to run the AVI clip when a specific event takes place at runtime. For example, to activate the AVI clip when a button object is clicked, write the button's *OnClick* event specifying that. You may also call the *Play* method to specify when to play the AVI.

Note: If you make any changes to the form or any of the components on the form after setting *Active* to *True*, the *Active* property becomes *False* and you have to reset it to *True*. Do this either just before runtime or at runtime.

For more information on using the animation control, see the topic called [Example of adding silent video clips](#).

Example of adding silent video clips

[Topic groups](#)

[See also](#)

Suppose you want to display an animated logo as the first screen that appears when your application starts. After the logo finishes playing the screen disappears.

To run this example, create a new project and save the Unit1.pas file as Frmlogo.pas and save the Project1.dpr file as Logo.dpr. Then:

- 1 Double-click the animate icon from the Win32 page of the Component palette.
- 2 Using the Object Inspector, set its Name property to *Logo1*.
- 3 Select its FileName property, click the ellipsis (...) button, choose the cool.avi file from your ..\Demos\Coolstuf directory. Then click Open in the Open AVI dialog.

This loads the cool.avi file into memory.

- 4 Position the animation control box on the form by clicking and dragging it to the top right hand side of the form.
- 5 Set its Repetitions property to 5.
- 6 Click the form to bring focus to it and set its Name property to *LogoForm1* and its Caption property to *Logo Window*. Now decrease the height of the form to right-center the animation control on it.
- 7 Double-click the form's *OnActivate* event and write the following code to run the AVI clip when the form is in focus at runtime:

```
Logo1.Active := True;
```

- 8 Double-click the Label icon on the Standard page of the Component palette. Select its Caption property and enter *Welcome to Cool Images 4.0*. Now select its Font property, click the ellipsis (...) button and choose Font Style: Bold, Size: 18, Color: Navy from the Font dialog and click OK. Click and drag the label control to center it on the form.
- 9 Click the animation control to bring focus back to it. Double-click its *OnStop* event and write the following code to close the form when the AVI file stops:

```
LogoForm1.Close;
```

- 10 Select Run|Run to execute the animated logo window.

Adding audio and/or video clips to an application

[Topic groups](#)

[See also](#)

The media player component in Delphi allows you to add audio and/or video clips to your application. It opens a media device and plays, stops, pauses, records, etc., the audio and/or video clips used by the media device. The media device may be hardware or software.

Note: Audio and video clip support is not provided for cross-platform programming.

To add an audio and/or video clip to an application:

- 1 Double-click the media player icon on the System page of the Component palette. This automatically put a media player control on the form window in which you want the media feature.
- 2 Using the Object Inspector, select the Name property and enter a new name for your media player control. You will use this when you call the media player control. (Follow the standard rules for naming Delphi identifiers.)

Always work directly with the Object Inspector when setting design time properties and creating event handlers.

- 3 Select the DeviceType property and choose the appropriate device type to open using the AutoOpen property or the Open method. (If DeviceType is dtAutoSelect the device type is selected based on the file extension of the media file specified by the FileName property.) For more information on device types and their functions, see the table below.
- 4 If the device stores its media in a file, specify the name of the media file using the FileName property. Select the FileName property, click the ellipsis (...) button, and choose a media file from any available local or network directories and click Open in the Open dialog. Otherwise, insert the hardware the media is stored in (disk, cassette, and so on) for the selected media device, at runtime.
- 5 Set the AutoOpen property to True. This way the media player automatically opens the specified device when the form containing the media player control is created at runtime. If AutoOpen is False, the device must be opened with a call to the Open method.
- 6 Set the AutoEnable property to True to automatically enable or disable the media player buttons as required at runtime; or, double-click the EnabledButtons property to set each button to True or False depending on which ones you want to enable or disable.

The multimedia device is played, paused, stopped, and so on when the user clicks the corresponding button on the media player component. The device can also be controlled by the methods that correspond to the buttons (Play, Pause, Stop, Next, Previous, and so on).

- 7 Position the media player control bar on the form by either clicking and dragging it to the appropriate place on the form or by selecting the Align property and choosing the appropriate align position from the drop down list. If you want the media player to be invisible at runtime, set the Visible property to False and control the device by calling the appropriate methods (Play, Pause, Stop, Next, Previous, Step, Back, Start Recording, Eject).
- 8 Make any other changes to the media player control settings. For example, if the media requires a display window, set the Display property to the control that displays the media. If the device uses multiple tracks, set the Tracks property to the desired track.

<u>Device Type</u>	<u>Software/Hardware used</u>	<u>Plays</u>	<u>Uses Tracks</u>
dtAVIVideo	AVI Video Player for Windows	AVI Video files	No
dtCDAudio	CD Audio Player for Windows or a CD Audio Player	CD Audio Disks	Yes
dtDAT	Digital Audio Tape Player	Digital Audio Tapes	Yes
dtDigitalVideo	Digital Video Player for Windows	AVI, MPG, MOV files	No
dtMMMovie	MM Movie Player	MM film	No
dtOverlay	Overlay device	Analog Video	No
dtScanner	Image Scanner	N/A for Play (scans images on Record)	No
dtSequencer	MIDI Sequencer for Windows	MIDI files	Yes
dtVCR	Video Cassette Recorder	Video Cassettes	No
dtWaveAudio	Wave Audio Player for Windows	WAV files	No

For more information on using the media player control, see the topic called [Example of adding audio and/or video clips](#).

Example of adding audio and/or video clips (VCL only)

[Topic groups](#)

[See also](#)

This example runs an AVI video clip of a multimedia advertisement for Delphi. To run this example, create a new project and save the Unit1.pas file to FrmAd.pas and save the Project1.dpr file to DelphiAd.dpr. Then:

- 1 Double-click the media player icon on the System page of the Component palette.
- 2 Using the Object Inspector, set the Name property of the media player to *VideoPlayer1*.
- 3 Select its DeviceType property and choose dtAVIVideo from the drop down list.
- 4 Select its FileName property, click the ellipsis (...) button, choose the speedis.avi file from your .. \Demos\Coolstuf directory. Click Open in the Open dialog.
- 5 Set its AutoOpen property to *True* and its Visible property to *False*.
- 6 Double-click the Animate icon from the Win32 page of the Component palette. Set its AutoSize property to *False*, its Height property to *175* and Width property to *200*. Click and drag the animation control to the top left corner of the form.
- 7 Click the media player to bring back focus to it. Select its Display property and choose Animate1 from the drop down list.
- 8 Click the form to bring focus to it and select its Name property and enter *Delphi_Ad*. Now resize the form to the size of the animation control.
- 9 Double-click the form's *OnActivate* event and write the following code to run the AVI video when the form is in focus:

```
VideoPlayer1.Play;
```

- 10 Choose Run|Run to execute the AVI video.

Writing multi-threaded applications

[Topic groups](#)

[See also](#)

Delphi 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: Not all operating systems implement true multi-processing, even when it is supported by the underlying hardware. For example, Windows 9x only simulates multiprocessing, even if the underlying hardware supports it.

The following topics discuss support for threads in Delphi:

 [Defining thread objects](#)

 [Coordinating threads](#)

 [Executing thread objects](#)

 [Debugging multi-threaded applications](#)

Defining thread objects

[Topic groups](#)

[See also](#)

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.

Note: 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](#).

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, Delphi 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.
```

In the automatically generated unit file, you

-  Optionally, [Initialize the thread](#).
-  [Write the thread function](#) by filling in the Execute method.
-  Optionally, [Write clean-up code](#).

Initializing the thread

[Topic groups](#)

[See also](#)

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.

If writing a Windows application, *Priority* values fall along a seven-point scale, as described in the following table:

<u>Value</u>	<u>Priority</u>
<code>tpIdle</code>	The thread executes only when the system is idle. Windows won't interrupt other threads to execute a thread with <i>tpIdle</i> priority.
<code>tpLowest</code>	The thread's priority is two points below normal.
<code>tpLower</code>	The thread's priority is one point below normal.
<code>tpNormal</code>	The thread has normal priority.
<code>tpHigher</code>	The thread's priority is one point above normal.
<code>tpHighest</code>	The thread's priority is two points above normal.
<code>tpTimeCritical</code>	The thread gets highest priority.

Note: If writing a cross-platform application, you must use separate code for assigning priorities on Windows and Linux. On Linux, *Priority* is a numeric value that depends on the threading policy which can only be changed by root. See the CLX version of [TThread](#) and `~JMPPriority ~!`
`Alink(tthread_priority,1,TopicNotFound,main) JMP~> online Help for details.`

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

[Topic groups](#)

[See also](#)

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.

When implementing the Execute method, you can manage these issues by

-  [Using the main VCL/CLX thread](#)
-  [Using thread-local variables](#)
-  [Avoiding simultaneous access](#)
-  [Waiting for other threads](#)
-  [Checking for termination by other threads](#)
-  [Handling exceptions in the thread function](#)

Using the main VCL/CLX thread

[Topic groups](#)

[See also](#)

When you use objects from the VCL or CLX object hierarchies, 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 thread is set aside for access of VCL and CLX objects. This is the thread that handles all Windows messages received by components in your application.

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 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 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 VCL or CLX objects in console applications.

You do not always need to use the main 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 VCL or 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 BDE-enabled datasets, each thread must have its own database session component. The one exception to this is when you are using Access drivers, which are built using a Microsoft library that is not thread-safe. For dbDirect, as long as the vendor client library is thread-safe, the dbDirect components will be thread-safe. ADO and InterbaseExpress components are thread-safe.

When using data access components, you must still 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.

For more information about using database sessions with threads in BDE-enabled applications, see [Managing multiple sessions](#).

 VisualCLX objects are not thread-safe.

 DataCLX objects are thread-safe.

 Graphics objects are thread-safe. You do not need to use the main VCL or CLX thread to access [TFont](#), [TPen](#), [TBrush](#), [TBitmap](#), [TMetafile](#) (VCL only), [TDrawing](#) (CLX only), or [TIcon](#). Canvas objects can be used outside the *Synchronize* method by [locking them](#).

 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

[Topic groups](#)

[See also](#)

The thread function 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

[Topic groups](#)

[See also](#)

Your thread object begins running when the *Execute* method is called (see [Executing thread objects](#)) 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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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 VCL or 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 VCL or CLX thread values).

 You can safely access any components and VCL or CLX objects from the *OnTerminate* event handler without worrying about clashing with other threads.

Coordinating threads

[Topic groups](#)

[See also](#)

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.

Whether using thread objects or generating threads using `BeginThread`, the following topics describe techniques for coordinating threads:



[Avoiding simultaneous access](#)



[Waiting for other threads](#)



[Using the main VCL/CLX thread](#)

When 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.

Avoiding simultaneous access

[Topic groups](#)

[See also](#)

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.

The VCL and CLX include support for three techniques that prevent other threads from accessing the same memory as your thread:



[Locking objects](#)



[Using critical sections](#)



[Using a multi-read exclusive-write synchronizer](#)

Locking objects

[Topic groups](#)

[See also](#)

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.

The VCL and CLX also both include 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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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.

Waiting for other threads

[Topic groups](#)

[See also](#)

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



Wait for a task to be completed.

Waiting for a thread to finish executing

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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 the following table:

<u>Value</u>	<u>Meaning</u>
<code>wrSignaled</code>	The signal of the event was set.
<code>wrTimeout</code>	The specified time elapsed without the signal being set.
<code>wrAbandoned</code>	The event object was destroyed before the timeout period elapsed.
<code>wrError</code>	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

[Topic groups](#)

[See also](#)

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.

The following topics discuss how to use the threads in your application:



[Overriding the default priority.](#)



[Starting and stopping threads](#)

Overriding the default priority

[Topic groups](#)

[See also](#)

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](#). 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 }
```

Note: If writing a cross-platform application, you must use separate code for assigning priorities on Windows and Linux. On Linux, *Priority* is a numeric value that depends on the threading policy which can only be changed by root. See the CLX version of [TThread](#) and [~JMPPriority~!](#)
Alink(tthread_priority,1,TopicNotFound,main) JMP~> online Help for details.

Starting and stopping threads

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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*.

Using CLX for cross-platform development

[Topic groups](#)

[See also](#)

You can use Delphi to develop cross-platform 32-bit applications that run on both the Windows and Linux 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. Kylix is Borland's Delphi for Linux software that allows you to compile and develop applications on Linux. If you want to develop and deploy applications on Linux and Windows, you'll need to use Kylix as well as Delphi.

Note: Most applications developed using CLX (with no operating system specific API calls) will run on both Linux and Windows platforms. The application must be compiled on the platform on which you want it to run.

The following topics provide additional information:

-  [Creating cross-platform applications](#)
-  [Porting VCL applications to CLX](#)
-  [Porting your application](#)
-  [CLX versus VCL](#)
-  [What CLX does differently](#)
-  [Features that will not port](#)
-  [Writing portable code](#)
-  [Messages and system events](#)

Creating cross-platform applications

[Topic groups](#)

[See also](#)

You create cross-platform applications much as you create any Delphi application. You need to use CLX visual components, and you should not use operating system specific APIs if you want the application to be completely cross-platform. (See [Writing portable code](#) for tips on writing cross-platform applications.)

To create a cross-platform application:

- 1 In the IDE, choose File|New|CLX application.
The Component palette shows components that can be used in CLX applications.
- 2 Develop your application within the IDE. Remember to use only CLX components in your application.
- 3 Compile and test the application on each platform on which you want to run the application. Review any error messages to see where additional changes need to be made.

When moving an application to Kylix, you need to reset your project options. That's because the .dof file which stores the project options is recreated on Kylix and called .kof (with the default options set). You can also store many of the compiler options with the application by typing Ctrl+O+O. The options are placed at the beginning of the currently open file.

The form file in cross-platform applications has an extension of xfm instead of dfm. This is to distinguish cross-platform forms that use CLX components from forms that use VCL components. An xfm form file works on both Windows or Linux but a dfm form only works on Windows.

You could also begin development of a cross-platform application by starting on Kylix instead of Delphi:

- 1 Develop, compile and test the application on Linux using Kylix.
- 2 Move the application source files over to Windows.
- 3 Reset your project options.
- 4 Recompile the application on Windows using Delphi.

For information on writing platform-independent database or internet applications, see



[Cross-platform database applications](#)



[Cross-platform Internet applications](#)

Porting VCL applications to CLX

Topic groups

If you have Delphi applications that were written for the Windows environment, you can make them cross platform. How easy it will be depends on the nature and complexity of the application and how many Windows dependencies there are.

The following topics 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
-  Porting your application
-  CLX versus VCL
-  Features that will not port
-  Environmental differences between Windows and Linux
-  Writing portable code
-  Programming differences on Linux
-  Cross-platform database applications

Porting techniques

[Topic groups](#)

[See also](#)

The following are different approaches you can take to port an application from one platform to another:

<u>Technique</u>	<u>Description</u>
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. Delphi’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 solution is very hard to maintain.

Where you want to emulate Windows APIs, you can include two distinct sections using **\$IFDEFs** to indicate sections of the code that apply specifically to Windows or Linux.

Porting your application

[Topic groups](#)

[See also](#)

If you are porting an application that you want to run on both Windows and Linux, you need to modify your code or use **\$IFDEFs** to indicate sections of the code that apply specifically to Windows or Linux.

Follow these general steps to port your VCL application to CLX:

- 1 Open the project containing the application you want to change in Delphi.
- 2 Copy .dfm files to .xpm files of the same name (for example, rename unit1.dfm to unit1.xpm). Rename (or **\$IFDEF**) the reference to the .dfm file in the unit file(s) from {\$R *.dfm} to {\$R *.xpm}. (The .xpm file will work in both Kylix and Delphi.)

For example, change the form reference in the **implementation** section from

```
{ $R *.dfm }  
to  
{ $R *.xpm }
```

- 3 Change (or **\$IFDEF**) all **uses** clauses so they refer to the correct units in CLX. (See [CLX and VCL unit comparison](#) for information.)

For example, change the following **uses** clause in a Windows application

```
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
Dialogs, StdCtrls;
```

to the following for a CLX application:

```
uses Windows, Messages, SysUtils, Variants, Classes, QForms, QControls, QStdCtrls;
```

- 4 Save the project and reopen it. Now the Component palette shows components that can be used in CLX applications.

Note: Some Windows only nonvisual components can be used in CLX applications but they contain functionality that will only work in Windows CLX applications. If you plan to compile your application on Linux as well, do not use the nonvisual VCL components in your applications or use **\$IFDEFs** to mark these sections of the code as Windows only. You cannot use the visual part of VCL with VisualCLX in the same application.

- 5 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](#) for information.)
- 6 Find equivalent functionality for features that are different on Linux. Use **\$IFDEFs** (sparingly) to delimit Windows-specific information. (See [Using conditional directives](#) 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 ]
```

- 7 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.

- 8 Compile, test and debug your application.

To transfer the application to Linux:

- 1 Move your Delphi 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 (.dpc files).

Project-related files include form files (.xfrm files), resource files (.res files), and project options files (.dof files—in Kylix these change to .kof 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—in Kylix this changes to .conf).

- 2 Open the project in Kylix. You will receive warnings on Windows-specific features that are in use.
- 3 Compile the project using Kylix. Review any error messages to see where additional changes need to be made.

CLX versus VCL

[Topic groups](#)

[See also](#)

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. Delphi includes both CLX and the VCL.

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:

<u>Part</u>	<u>Description</u>
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 Web Broker. These are the same on Linux and Windows.
RTL	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 in CLX and to change the names to CLX 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

[Topic groups](#)

[See also](#)

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 to be aware of when writing cross-platform applications.

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 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 converting a VCL application that included Windows calls to a CLX application, 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 *VarClear* 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.

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 containing 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 on Linux, 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 in cross-platform applications. However, in CLX, you need to use *TMemIniFile* instead of *TRegIniFile*.

Other differences

CLX 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.

The key value in a *OnKeyDown* event or *KeyUp* event for the Enter key on Windows is 13. On Linux, this value is 4100. If you check for a hardcoded numeric value for a key, such as checking for a value of 13 for the Enter key, you need to change this when porting a Delphi application to Kylix.

Additional differences exist. Refer to the CLX online documentation for details on all of the CLX objects or in versions of Delphi that include the source code you can refer to the code, it is located in ..
\Delphi6\Source\VCL\CLX.

Features that will not port

[Topic groups](#)

[See also](#)

When using CLX instead of the VCL, many of the objects are the same. However, the objects may be missing some features (such as properties, methods, or events). 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)

See [Features that will not port](#) for information on Windows-specific features that are missing in Linux.

Features that will not port

[Topic groups](#)

[See also](#)

Some Windows-specific features supported on Delphi 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.

<u>Delphi/Windows feature</u>	<u>Kylix/Linux feature</u>
ADO components	Regular database components
Automation Servers	Not available
BDE	dbExpress and regular database components
COM+ components (including ActiveX)	Not available
DataSnap	Not yet available
FastNet	Not available
Internet Express	Not yet available
Legacy components (such as items on the Win 3.1 Component palette tab)	Not available
Messaging Application Programming Interface (MAPI) includes a standard library of Windows messaging functions.	SMTP/POP3 let you send, receive, and save email messages
Quick Reports	Not available
Web Services (SOAP)	Not yet available
WebSnap	Not yet available
Windows API calls	CLX methods, Qt calls, libc calls, or calls to other system libraries
Windows messaging	Qt events
Winsock	BSD sockets

The Linux 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 or Delphi generates the correct code. (For information, see [Including inline assembler code.](#))

Kylix library modules and packages are implemented using .so files.

CLX and VCL unit comparison

[Topic groups](#)

[See also](#)

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 topic provides tables that list the CLX units and the comparable VCL unit, list the units that are for CLX only, and list the units that are for VCL only.

The following table lists VCL units and the comparable CLX units:

<u>VCL units</u>	<u>CLX units</u>
ActnList	QActnList
Buttons	QButtons
CheckLst	QCheckLst
Classes	Classes
Clipbrd	QClipbrd
ComCtrls	QComCtrls
Consts	Consts, QConsts, and RTLConsts
Contnrs	Contnrs
Controls	QControls
DateUtils	DateUtils
DB	DB
DBActns	QDBActns
DBCClient	DBCClient
DBCommon	DBCommon
DBConnAdmin	DBConnAdmin
DBConsts	DBConsts
DBCtrls	QDBCtrls
DBGrids	QDBGrids
DBLocal	DBLocal
DBLocalS	DBLocalS
DBLogDlg	DBLogDlg
DBXpress	DBXpress
Dialogs	QDialogs
DSIntf	DSIntf
ExtCtrls	QExtCtrls
FMTBCD	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	Qt
Search	QSearch

Sockets	Sockets
StdActns	QStdActns
StdCtrls	QStdCtrls
SqlConst	SqlConst
SqlExpr	SqlExpr
SqlTimSt	SqlTimSt
SyncObjs	SyncObjs
SysConst	SysConst
SysInit	SysInit
System	System
SysUtils	SysUtils
Types	Types and QTypes
TypeInfo	TypeInfo
Variants	Variants
VarUtils	VarUtils

The following units are in CLX but not VCL:

<u>Unit</u>	<u>Description</u>
DirSel	Directory selection
QStyle	GUI look and feel

The following Windows VCL units are not included in CLX mostly because they concern Windows-specific features that are not available on Linux such as ADO, COM, and the BDE. The reason for the unit's exclusion is listed.

<u>Unit</u>	<u>Reason for exclusion</u>
ADOCnst	No ADO feature
ADODB	No ADO feature
AppEvnts	No TApplicationEvent object
AxCtrls	No COM feature
BdeConst	No BDE feature
ComStrs	No COM feature
ConvUtils	New feature for Delphi 6
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
ExtActns	New feature to Delphi 6
ExtDlgs	No picture dialogs
FileCtrl	Obsolete
ListActns	New feature to Delphi 6
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

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
StdConvs	New feature to Delphi 6
SvcMgr	NT Services support
Tabnotbk	Obsolete
Tabs	Obsolete
ToolWin	No docking feature
VarCmplx	New feature to Delphi 6
VarConv	New feature to Delphi 6
VCLCom	No COM feature
WebConst	Windows-specific constants
Windows	Windows-specific (API)

Features that will not port

[Topic groups](#)

[See also](#)

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 calling into the Qt interface library such as *QWidget_Create()*, you are creating an instance of a Qt widget that is not owned by a CLX object. This passes the instance of an existing Qt widget to the CLX object to use during its construction. 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

[Topic groups](#)

[See also](#)

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 create a new CLX application in Delphi or Kylix, an .xfm is created instead of a .dfm. If you plan to write cross-platform applications, the .xfm will work both on Delphi and Kylix.

Environmental differences between Windows and Linux

[Topic groups](#)

[See also](#)

Currently, cross-platform means an application that can run virtually unchanged on both the Windows and Linux operating systems. The following table lists some of the differences between Linux and the Windows operating environments.

<u>Difference</u>	<u>Description</u>
File name case sensitivity	In Linux, a capital letter is <i>not</i> the same as a lowercase letter. The file Test.txt is <i>not</i> 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, <code>chmod +x <scriptfile></code> . 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 redisplay the command prompt without a status message.
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.

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

[Topic groups](#)

[See also](#)

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.

<u>Directory</u>	<u>Contents</u>
/	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 UNIX/Linux hierarchical file system and to read the *Filesystem Hierarchy Standard*.

Writing portable code

[Topic groups](#)

[See also](#)

If you are writing cross-platform applications that are meant to run on Windows and Linux, 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](#).)

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. For more information, see [Using conditional directives](#).

Using conditional directives

[Topic groups](#)

[See also](#)

Using **\$IFDEF** compiler directives is a reasonable way to conditionalize your code for the Windows and Linux platforms. However, because **\$IFDEFs** make source code harder to understand and maintain, you need to understand when it is reasonable to use **\$IFDEFs**. When considering the use of **\$IFDEFs**, 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 **\$IFDEFs** within cross-platform applications:

 Try not to use **\$IFDEFs** unless absolutely necessary. **\$IFDEFs** in a source file are only evaluated when source code is compiled. Unlike C/C++, Delphi does not require unit sources (header files) to compile a project. Full rebuilds of all source code is an uncommon event for most Delphi projects.

 Do not use **\$IFDEFs** in package (.dpk) files. Limit their use to source files. Component writers need to create two design-time packages when doing cross-platform development, not one package using **\$IFDEFs**.

 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 **\$IFDEFs**. 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 **\$IFDEFs**, 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 **\$IFDEFs** 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 **\$IFDEFs** to hide the new **\$IF** syntax from previous compilers.

For recommendations concerning ending these directives, see [Terminating conditional directives](#). For an overview of using conditional directives, see [Conditional compilation](#).

Terminating conditional directives

[Topic groups](#)

[See also](#)

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 **\$IFDEFs**, 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 or Delphi 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.

For more information, see [Emitting messages](#).

Emitting messages

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

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

[Topic groups](#)

[See also](#)

Message loops and events work differently on Linux and in CLX, 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 in a cross-platform application, 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 take place.

Method	Description
<i>ChangeBounds</i>	Used when a <i>TWidgetControl</i> 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.
<i>ChangeScale</i>	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 <i>ChangeScale</i> 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.
<i>ColorChanged</i>	Called when the color of the control has been changed.
<i>CursorChanged</i>	Called when the cursor changes shape. The mouse cursor assumes this shape when it's over this widget.
<i>EnabledChanged</i>	Called when an application changes the enabled state of a window or control.
<i>FontChanged</i>	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.
<i>PaletteChanged</i>	Called when the system palette has been changed. .
<i>ShowHintChanged</i>	Called when Help hints are displayed or hidden on a control.
<i>StyleChanged</i>	Called when the window or control's GUI styles have changed.
<i>TabStopChanged</i>	Called when the tab order on the form has been changed.
<i>VisibleChanged</i>	Called when a control is hidden or shown.
<i>WidgetDestroyed</i>	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 Delphi 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 in Delphi CLX 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 Kylix go through *EventHandler*, which is basically a replacement for *WndProc*.

Programming differences on Linux

[Topic groups](#)

[See also](#)

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 is not allowed in Delphi.

Cross-platform database applications

[Topic groups](#)

[See also](#)

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. See [dbExpress differences](#) for details on how DBExpress differs from other low-level data access implementations.

 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. See [Component-level differences](#) for details on how DBExpress applications differ at the component level.

 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. See [Updating data in dbExpress applications](#) for details.

For information on porting existing database applications to *dbExpress*, see [Porting database applications to Linux](#). For information on designing new *dbExpress* applications, see [Designing database applications](#).

dbExpress differences

[Topic groups](#)

[See also](#)

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 (ldapi32.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

[Topic groups](#)

[See also](#)

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.

The following table 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.

<u>InterBase Express components</u>	<u>BDE components</u>	<u>ADO components</u>	<u>dbExpress components</u>
<i>TIBDatabase</i>	<i>TDatabase</i>	<i>TADOConnection</i>	TSQLConnection
<i>TIBTable</i>	<i>TTable</i>	<i>TADOTable</i>	TSQLTable
<i>TIBQuery</i>	<i>TQuery</i>	<i>TADOQuery</i>	TSQLQuery
<i>TIBStoredProc</i>	<i>TStoredProc</i>	<i>TADOStoredProc</i>	TSQLStoredProc
<i>TIBDataSet</i>		<i>TADODataSet</i>	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 [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](#).

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 than 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

[Topic groups](#)

[See also](#)

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](#).

Porting database applications to Linux

[Topic groups](#)

[See also](#)

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/VCL database application to Kylix/CLX:

- 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.

Some versions of Delphi include the Data Pump utility which you can use 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](#).

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 described in [Component-level differences](#) 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 [Controlling connections](#).

Updating data in dbExpress applications

[Topic groups](#)

[See also](#)

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*:

<u>On BDE datasets (or TDatabase)</u>	<u>On ADO datasets</u>	<u>On TClientDataSet</u>	<u>Purpose</u>
<i>CachedUpdates</i>	<i>LockType</i>	Not needed, client datasets always cache updates.	Determines whether cached updates have any effect.
Not supported.	<i>CursorType</i>	Not supported.	Specifies how isolated the dataset is from changes on the server.
<i>UpdatesPending</i>	Not supported.	<u><i>ChangeCount</i></u>	Indicates whether the local cache contains updated records that need to be applied to the database.
<i>UpdateRecordTypes</i>	<i>FilterGroup</i>	<u><i>StatusFilter</i></u>	Indicates the kind of updated records that are visible when applying cached updates.
<i>UpdateStatus</i>	<i>RecordStatus</i>	<u><i>UpdateStatus</i></u>	Indicates if a record is unchanged, inserted, or deleted.
<i>OnUpdateError</i>	Not supported.	<u><i>OnReconcileError</i></u>	An event for handling update errors on a record-by-record basis.
<i>ApplyUpdates</i> (on dataset or database)	UpdateBatch	<u><i>ApplyUpdates</i></u>	Applies records in the local cache to the database.
<i>CancelUpdates</i>	CancelUpdates or CancelBatch	<u><i>CancelUpdates</i></u>	Removes pending updates from the local cache without applying them.
<i>CommitUpdates</i>	Handled automatically	<u><i>Reconcile</i></u>	Clears the update cache following the application of updates.
<i>FetchAll</i>	Not supported	<u><i>GetNextPacket</i></u> (and <u><i>PacketRecords</i></u>)	Copies database records to the local cache for editing and updating.
<i>RevertRecord</i>	CancelBatch	<u><i>RevertRecord</i></u>	Undoes updates to the current record if they are not yet applied.

Cross-platform Internet applications

[Topic groups](#)

[See also](#)

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.

You can use Delphi or Kylix 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 Web Broker will run on both Windows and Linux.

Porting Internet applications to Linux

[Topic groups](#)

[See also](#)

If you have existing Internet applications that you want to make cross-platform, you should consider whether you want to port your Web server application or if you want to create a new application on Linux. See [Creating Internet server applications](#) for information on writing Web servers. If your application uses Web Broker and writes to the Web Broker 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 VCL applications to CLX](#).

About packages

[Topic groups](#)

[See also](#)

A *package* is a special dynamic-link library used by Delphi 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. To distinguish them from other DLLs, package libraries are stored in files that end with the .bpl (Borland package library) extension.

Like other runtime libraries, packages contain code that can be shared among applications. For example, the most frequently used Delphi components reside in a package called vcl. Each time you create an application, it automatically uses vcl. 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 the runtime package called vcl60.bpl. A computer with several package-enabled applications installed on it needs only a single copy of vcl60.bpl, which is shared by all the applications and the Delphi IDE itself.

Delphi ships with several precompiled runtime packages that encapsulate VCL and CLX components. Delphi 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 Delphi components, you can compile your components into design-time packages before installing them.

Package topics:

-  [Why use packages?](#)
-  [Packages and standard DLLs](#)
-  [Runtime packages](#)
-  [Using runtime packages in an application](#)
-  [Deciding which runtime packages to use](#)
-  [Custom packages](#)
-  [Design-time packages](#)
-  [Installing component packages](#)
-  [Creating and editing packages](#)
-  [Creating a package](#)
-  [Editing an existing package](#)
-  [Editing package source files manually](#)
-  [Understanding the structure of a package](#)
 -  [Naming packages](#)
 -  [The Requires clause](#)
 -  [Avoiding circular package references](#)
 -  [Handling duplicate package references](#)
 -  [The Contains clause](#)
 -  [Avoiding redundant source code uses](#)
-  [Compiling packages](#)
 -  [Package-specific compiler directives](#)
 -  [Weak packaging](#)
 -  [Using the command-line compiler and linker](#)
 -  [Package files created by a successful compilation](#)
-  [Deploying packages](#)
-  [Deploying applications that use packages](#)
-  [Distributing packages to other developers](#)
-  [Package collection files](#)

Why use packages?

[Topic groups](#)

[See also](#)

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 Delphi 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.

For more information, see [Packages and standard DLLs](#).

Packages and standard DLLs

[Topic groups](#)

[See also](#)

Create a package when you want to make a custom component that's available through the IDE. Create a standard DLL 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:

<u>File extension</u>	<u>Contents</u>
dpk	The source file listing the units contained in the package.
dcp	A binary image containing a package header and the concatenation of all dcu 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.
dcu	A binary image for a unit file contained in a package. One dcu is created, when necessary, for each unit file.
bpl	The runtime package. This file is a Windows DLL with special Delphi-specific features. The base name for the bpl is the base name of the dpk source file.

You can include VCL or CLX or both types of components in a package. Packages meant to be cross-platform should include CLX components only.

Note: Packages share their global data with other modules in an application.

Runtime packages

[Topic groups](#)

[See also](#)

Runtime packages are deployed with Delphi 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 (.bpl files) that the application uses. The .bpl 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 .bpls.

 [Using runtime packages in an application](#)

 [Deciding which runtime packages to use](#)

 [Custom packages](#)

Using runtime packages in an application

[Topic groups](#)

[See also](#)

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. 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 Delphi’s global Library Path.

You do not need to include file extensions with package names (or the number representing the Delphi release); that is, `vcl60.bpl` is written as `vcl`. If you type directly into the Runtime Packages edit box, be sure to separate multiple names with semicolons. For example:

```
rtl;vcl;vclldb;vclado;vclx;Vclbde;
```

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 Delphi 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  
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
  Dialogs;
```

The units referenced in this example are contained in the `vcl` and `rtl` packages. Nonetheless, you must keep these references in the **uses** clause, even if you use `vcl` and `rtl` in your application, or you will get compiler errors. In generated source files, Delphi adds these units to the **uses** clause automatically.

 [Deciding which runtime packages to use](#)

Dynamically loading packages

[Topic groups](#)

[See also](#)

To load a package at runtime, call the *LoadPackage* function. *LoadPackage* loads the package, checks for duplicate units, and calls the initialization blocks of all units contained in the package. 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

[Topic groups](#)

[See also](#)

Delphi ships with several precompiled runtime packages, including rtl and vcl, which supply basic language and component support.

The vcl package contains the most commonly used components; the rtl package includes all the non-component system functions and Windows interface elements. It does not include database or other special components, which are available in separate packages.

To create a client/server database application that uses packages, you need at least three runtime packages: vcl and vcldb. If you want to use Outline components in your application, you also need vclx. To use these packages, choose Project|Options, select the Packages tab, and enter the following list in the Runtime Packages edit box.

```
rtl;vcl;Vcldb;vclx;
```

Actually, you don't have to include vcl and rtl, because they are referenced in the [Requires clause](#) of vcldb. Your application will compile just the same whether or not vcl and rtl are included in the Runtime Packages edit box.

Custom packages

[Topic groups](#)

[See also](#)

A custom package is either a bpl 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 stats.bpl. To use it in an application, the line you enter in the Runtime Packages edit box might look like this:

```
rtl;vcl;vcldb;stats
```

If you create your own packages, you can add them to the list as needed.

Design-time packages

[Topic groups](#)

[See also](#)

Design-time packages are used to install components on the IDE's Component palette and to create special property editors for custom components.

Delphi ships with many design-time component packages preinstalled in the IDE. Which ones are installed depends on which version of Delphi you are using and whether or not you have customized it. You can view a list of what packages are installed on your system by choosing Component|Install Packages.

The design-time packages work by calling runtime packages, which they reference in their [Requires clause](#). For example, dclstd references vcl. Dclstd itself contains additional functionality that makes most 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](#)

Installing component packages

[Topic groups](#)

[See also](#)

All components are installed in the IDE as packages. If you've written your own components, [create and compile](#) a package that contains them. Your component source code must follow the model described in [Overview of component creation](#).

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 .bpl, .dcp, and .dcu files be sure to copy all of them.

The directory where you store the .dcp file—and the .dcu files, if they are included with the distribution—must be in the Delphi Library Path.

If the package is shipped as a .dpc (package collection) file, only the one file needs to be copied; the .dpc file contains the other files. (For more information about package collection files, see [Package collection files](#).)

- 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 .bpl or .dpc file resides (see step 1). Select the .bpl or .dpc file and click Open. If you select a .dpc file, a new dialog box appears to handle the extraction of the .bpl and other files from the package collection.



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 Packages tab of the Project Options 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

[Topic groups](#)

[See also](#)

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 .bpl. 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 .dpc extension, are generated by the Package editor.



[Creating a package](#)



[Editing an existing package](#)



[Editing package source files manually](#)



[Understanding the structure of a package](#)



[Compiling packages](#)

Creating a package

[Topic groups](#)

[See also](#)

To create a package, follow the procedure below. Refer to [Understanding the structure of a package](#) 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

[Topic groups](#)

[See also](#)

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 defproj.dof file.

Editing Package source files manually

[Topic groups](#)

[See also](#)

Package source files, like project files, are generated by Delphi from information you supply. Like project files, they can also be edited manually. A package source file should be saved with the .dpk (Delphi 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 vcldb package (in the source file vcldb60.bpl):

```
package vcldb;  
  requires vcldb;  
  contains rtl, vcl, Db, DBActns, DBOleCtl, Dbcgrids, dbCommon, dbConsts, Dbctrls,  
    Dbgrids, Dblogdlg, SQLTimSt, FmtBcd;  
end.
```

Understanding the structure of a package

[Topic groups](#)

[See also](#)

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 STATS.bpl 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.

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 rtl. If using VCL components, you'll also need to include the vcl package. If using CLX components for cross-platform programming, you need to include VisualCLX.

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.

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 Delphi IDE*. This means that if you create a package that contains one of the units in vcl 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

[Topic groups](#)

[See also](#)

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 Delphi 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 11-12.



[Package-specific compiler directives](#)



[Weak packaging](#)



[Using the command-line compiler and linker](#)



[Package files created by a successful compilation](#)

Package-specific compiler directives

[Topic groups](#)

[See also](#)

The following table lists package-specific compiler directives that you can insert into your source code.

<u>Directive</u>	<u>Purpose</u>
<code>{\$IMPLICITBUILD OFF}</code>	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.
<code>{\$G-}</code> or <code>{IMPORTEDDATA OFF}</code>	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.
<code>{\$WEAKPACKAGEUNIT ON}</code>	Packages unit “weakly.” See Weak packaging .
<code>{\$DENYPACKAGEUNIT ON}</code>	Prevents unit from being placed in a package.
<code>{\$DESIGNONLY ON}</code>	Compiles the package for installation in the IDE. (Put in .dpk file.)
<code>{\$RUNONLY ON}</code>	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](#) for information on compiler directives not discussed here.

Refer to [Creating packages and DLLs](#) for additional directives that can be used in all libraries.

Weak packaging

[Topic groups](#)

[See also](#)

The **\$WEAKPACKAGEUNIT** directive affects the way a .dcp file is stored in a package's .dcp and .bpl files. (For information about files generated by the compiler, see [Package files created by a successful compilation](#) .) If **{\$WEAKPACKAGEUNIT ON}** appears in a unit file, the compiler omits the unit from bpls 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.dll. If you put **{\$WEAKPACKAGEUNIT ON}** in UNIT1.pas when you compile your package, UNIT1 will not be included in PACK.bpl; you will not have to distribute copies of RARE.dll 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 PACK.bpl. 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 Delphi programmers. It can help you to avoid distribution of infrequently used DLLs, and to eliminate conflicts among packages that may depend on the same external library.

For example, Delphi's PenWin unit references PenWin.dll. Most projects don't use PenWin, and most computers don't have PenWin.dll installed on them. For this reason, the PenWin unit is weakly packaged in vcl. When you compile a project that uses PenWin and the vcl package, PenWin is copied from VCL60.dcp and bound directly into your project; the resulting executable is statically linked to PenWin.dll.

If PenWin were not weakly packaged, two problems would arise. First, vcl itself would be statically linked to PenWin.dll, and so you could not load it on any computer which didn't have PenWin.dll installed. Second, if you tried to create a package that contained PenWin, a compiler error would result because the PenWin unit would be contained in both vcl and your package. Thus, without weak packaging, PenWin could not be included in standard distributions of vcl.

Using the command-line compiler and linker

[Topic groups](#)

[See also](#)

When you compile from the command line, you can use the package-specific switches listed in the following table.

Switch	Purpose
<code>-\$G-</code>	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.
<code>-LEpath</code>	Specifies the directory where the package bpl file will be placed.
<code>-LNpath</code>	Specifies the directory where the package dcp file will be placed.
<code>-LUpackage</code>	Use packages.
<code>-Z</code>	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](#) for information on command-line options not discussed here.

Package files created by a successful compilation

[Topic groups](#)

[See also](#)

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 traypak.bpl.

The following table lists the files produced by the successful compilation of a package.

<u>File extension</u>	<u>Contents</u>
dcp	A binary image containing a package header and the concatenation of all dcu 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.
dcu	A binary image for a unit file contained in a package. One dcu is created, when necessary, for each unit file.
bpl	The runtime package. This file is a Windows DLL with special Delphi-specific features. The base name for the bpl is the base name of the dpk source file.

When compiled, the bpi, bpl, and lib files are generated by default in the directories specified in 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

[Topic groups](#)

[See also](#)

You deploy packages much like you deploy other applications. For general deployment information, refer to [Deploying applications](#).

Deploying applications that use packages

When distributing an application that uses runtime packages, make sure that your users have the application's .exe file as well as all the library (.bpl or .dll) files that the application calls. If the library files are in a different directory from the .exe file, they must be accessible through the user's Path. You may want to follow the convention of putting library files in the Windows\System directory. If you use InstallShield Express, your installation script can check the user's system for any packages it requires before blindly reinstalling them.

Distributing packages to other developers

If you distribute runtime or design-time packages to other Delphi developers, be sure to supply both .dcp and .bpl files. You will probably want to include .dcu files as well.

Package collection files

[Topic groups](#)

[See also](#)

Package collections (.dpc files) offer a convenient way to distribute packages to other developers. Each package collection contains one or more packages, including bpls and any additional files you want to distribute with them. When a package collection is selected for IDE installation, its constituent files are automatically extracted from their .pce container; the Installation dialog box offers a choice of installing all packages in the collection or installing packages selectively.

To create a package collection,

- 1 Choose Tools|Package Collection Editor to open the Package Collection editor.
- 2 Click the Add a Package speed button, then select a bpl in the Select Package dialog and click Open. To add more bpls to the collection, click the Add a Package speed button again. A tree diagram on the left side of the Package editor displays the bpls as you add them. To remove a package, select it and click the Remove Package speed button.
- 3 Select the Collection node at the top of the tree diagram. On the right side of the Package Collection editor, two fields will appear:

 In the Author/Vendor Name edit box, you can enter optional information about your package collection that will appear in the Installation dialog when users install packages.

 Under Directory List, list the default directories where you want the files in your package collection to be installed. Use the Add, Edit, and Delete buttons to edit this list. For example, suppose you want all source code files to be copied to the same directory. In this case, you might enter Source as a Directory Name with C:\MyPackage\Source as the Suggested Path. The Installation dialog box will display C:\MyPackage\Source as the suggested path for the directory.

- 4 In addition to bpls, your package collection can contain .dcp, .dcl, and .pas (unit) files, documentation, and any other files you want to include with the distribution. Ancillary files are placed in file groups associated with specific packages (bpls); the files in a group are installed only when their associated bpl is installed. To place ancillary files in your package collection, select a bpl in the tree diagram and click the Add File Group speed button; type a name for the file group. Add more file groups, if desired, in the same way. When you select a file group, new fields will appear on the right in the Package Collection editor,

 In the Install Directory list box, select the directory where you want files in this group to be installed. The drop-down list includes the directories you entered under Directory List in step 3, above.

 Check the Optional Group check box if you want installation of the files in this group to be optional.

 Under Include Files, list the files you want to include in this group. Use the Add, Delete, and Auto buttons to edit the list. The Auto button allows you to select all files with specified extensions that are listed in the **contains** clause of the package; the Package Collection editor uses Delphi's global Library Path to search for these files.

- 5 You can select installation directories for the packages listed in the **requires** clause of any package in your collection. When you select a bpl in the tree diagram, four new fields appear on the right side of the Package Collection editor:

 In the Required Executables list box, select the directory where you want the .bpl files for packages listed in the **requires** clause to be installed. (The drop-down list includes the directories you entered under Directory List in step 3, above.) The Package Collection Editor searches for these files using Delphi's global Library Path and lists them under Required Executable Files.

 In the Required Libraries list box, select the directory where you want the .dcp files for packages listed in the **requires** clause to be installed. (The drop-down list includes the directories you entered under Directory List in step 3, above.) The Package Collection Editor searches for these files using Delphi's global Library Path and lists them under Required Library Files.

- 6 To save your package collection source file, choose File|Save. Package collection source files should be saved with the .pce extension.
- 7 To build your package collection, click the Compile speed button. The Package Collection editor generates a .dpc file with the same name as your source (.pce) file. If you have not yet saved the source file, the editor queries you for a file name before compiling.

To edit or recompile an existing .pce file, select File|Open in the Package Collection editor and locate the file you want to work with.

 DevGuide: [Building applications with Delphi](#)

Creating international applications

[Topic groups](#)

[See also](#)

This topic 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.

The following topics are discussed in this section:

 [Internationalization and localization](#)

 [Internationalizing applications](#)

 [Localizing applications](#)

 DevGuide: [Building applications with Delphi](#)

Internationalization and localization

[Topic groups](#)

[See also](#)

To create an application that you can distribute to foreign markets, there are two major steps that need to be performed:

 [Internationalization](#)

 [Localization](#)

If your version of Delphi includes the Translation Tools, you can use them to manage localization. For more information, see the online Help for the Translation Tools (ETM.hlp).

Internationalization

[Topic groups](#)

[See also](#)

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. Windows supports a large set of locales, each of which is described by a language and country pair.

Localization

[Topic groups](#)

[See also](#)

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.

 DevGuide: [Building applications with Delphi](#)

Internationalizing applications

[Topic groups](#)

[See also](#)

You need to complete the following steps to create internationalized applications:



You must [enable your code](#) to handle strings from international character sets.



You need to [design your user interface](#) so that it can accommodate the changes that result from localization.



You need to [isolate all resources](#) that need to be localized.

 DevGuide: [Building applications with Delphi](#)

Enabling application code

[Topic groups](#)

[See also](#)

You must make sure that the code in your application can handle the strings it will encounter in the various target locales. To do this, you must consider the following:



[Character sets](#)



[OEM and ANSI character sets](#)



[Double byte character sets](#)



[Wide characters](#)



[Locale-specific features](#)

Character sets

[Topic groups](#)

[See also](#)

The United States edition of Windows uses the ANSI Latin-1 (1252) character set. However, other editions of Windows use different character sets. For example, the Japanese version of Windows uses the Shift-JIS character set (code page 932), which represents Japanese characters as multibyte character codes.

There are generally three types of characters sets:

-  Single-byte
-  Multibyte
-  Fixed-width multibyte

Windows and Linux both support single-byte and multibyte character sets as well as Unicode. With a single-byte character set, each byte in a string represents one character. The ANSI character set used by many Western operating systems is a single-byte character set.

In a multibyte character set, some characters are represented by one byte and others by more than one byte. The first byte of a multibyte character is called the lead byte. In general, the lower 128 characters of a multibyte character set map to the 7-bit ASCII characters, and any byte whose ordinal value is greater than 127 is the lead byte of a multibyte character. Only single-byte characters can contain the null value (#0). Multibyte character sets—especially double-byte character sets (DBCS)—are widely used for Asian languages, while the UTF-8 character set used by Linux is a multibyte encoding of Unicode.

 DevGuide: [Building applications with Delphi](#)

OEM and ANSI character sets

[Topic groups](#)

[See also](#)

It is sometimes necessary to convert between the Windows character set (ANSI) and the character set specified by the code page of the user's machine (called the OEM character set).

Double byte character sets

[Topic groups](#)

[See also](#)

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, a multibyte string can contain one or more bytes per character. AnsiStrings can contain a mix of single-byte and multibyte characters.

The lead 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. Delphi provides you with many runtime library functions that allow you to do this, many of which are listed here:

AdjustLineBreaks	AnsiStrLower	ExtractFileDir
AnsiCompareFileName	AnsiStrPos	ExtractFileExt
AnsiExtractQuotedStr	AnsiStrRScan	ExtractFileName
AnsiLastChar	AnsiStrScan	ExtractFilePath
AnsiLowerCase	AnsiStrUpper	ExtractRelativePath
AnsiLowerCaseFileName	AnsiUpperCase	FileSearch
AnsiPos	AnsiUpperCaseFileName	IsDelimiter
AnsiQuotedStr	ByteToCharIndex	IsPathDelimiter
AnsiStrComp	ByteToCharLen	LastDelimiter
AnsiStrlComp	ByteType	StrByteType
AnsiStrLastChar	ChangeFileExt	StringReplace
AnsiStrLComp	CharToByteIndex	WrapText
AnsiStrlComp	CharToByteLen	

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

[Topic groups](#)

[See also](#)

One approach to working with ideographic character sets is to convert all characters to a wide character encoding scheme such as Unicode. Unicode characters and strings are also called wide characters and wide character strings. In the Unicode character set, each character is represented by two bytes. Thus a Unicode string is a sequence not of individual bytes but of two-byte words.

The first 256 Unicode characters map to the ANSI character set. The Windows operating system supports Unicode (UCS-2). The Linux operating system supports UCS-4, a superset of UCS-2. Delphi/Kylix supports UCS-2 on both platforms. Because wide characters are two bytes instead of one, the character set can represent many more different characters.

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.

The biggest disadvantage of working with wide characters is that Windows 9x only supports a few wide character API function calls. Because of this, the VCL components represent all string values as single byte or MBCS strings. Translating between the wide character system and the MBCS system every time you set a string property or read its value would require additional code and slow your application down. However, you may want to translate into wide characters for some special string processing algorithms that need to take advantage of the 1:1 mapping between characters and *WideChars*.

Including bi-directional functionality in applications

[Topic groups](#)

[See also](#)

Some languages do not follow the left to right reading order commonly found in western languages, but rather read words right to left and numbers left to right. These languages are termed bi-directional (BiDi) because of this separation. The most common bi-directional languages are Arabic and Hebrew, although other Middle East languages are also bi-directional.

TApplication has two properties, *BiDiKeyboard* and *NonBiDiKeyboard*, that allow you to specify the keyboard layout. In addition, the VCL supports bi-directional localization through the *BiDiMode* and *ParentBiDiMode* properties. The following table lists VCL objects that have these properties:

<u>Component palette page</u>	<u>VCL object</u>	
Standard	TButton	
	TCheckBox	
	TComboBox	
	TEdit	
	TGroupBox	
	TLabel	
	TListBox	
	TMainMenu	
	TMemo	
	TPanel	
	TPopupMenu	
	TRadioButton	
	TRadioGroup	
	TScrollBar	
	Additional	TActionMainMenuBar
		TActionToolBar
TCheckListBox		
TDrawGrid		
TMaskEdit		
TScrollBar		
TSpeedButton		
TStaticLabel		
Win32		TComboBoxEx
		THeaderControl
	TListView	
	TMonthCalendar	
	TPageControl	
	TRichEdit	
	TStatusBar	
	TTreeView	
	Data Controls	TDBCheckBox
		TDBComboBox
TDBEdit		
TDBGrid		
TDBListBox		
TDBLookupComboBox		
TDBLookupListBox		
TDBMemo		
TDBRadioGroup		
TDBRichEdit		
TDBText		
QReport		TQRDBText
	TQRExpr	
	TQRLabel	

Other classes

- TQRMemo
- TQRSysData
- TApplication (has no *ParentBiDiMode*)
- TForm
- THintWindow (has no *ParentBiDiMode*)
- TValueListEditor

Notes: *THintWindow* picks up the *BiDiMode* of the control that activated the hint.

Bi-directional properties

[Topic groups](#)

[See also](#)

TApplication's *BiDiKeyboard* and *NonBiDiKeyboard*, support bi-directional localization.

Note: Bi-directional properties are not available in CLX for cross-platform programming.

The property *BiDiMode* is a new enumerated type, *TBiDiMode*, with four states: *bdLeftToRight*, *bdRightToLeft*, *bdRightToLeftNoAlign*, and *bdRightToLeftReadingOnly*.

bdLeftToRight

bdLeftToRight draws text using left to right reading order, and the alignment and scroll bars are not changed. For instance, when entering right to left text, such as Arabic or Hebrew, the cursor goes into push mode and the text is entered right to left. Latin text, such as English or French, is entered left to right. *bdLeftToRight* is the default value.

bdRightToLeft

bdRightToLeft draws text using right to left reading order, the alignment is changed and the scroll bar is moved. Text is entered as normal for right-to-left languages such as Arabic or Hebrew. When the keyboard is changed to a Latin language, the cursor goes into push mode and the text is entered left-to-right.

bdRightToLeftNoAlign

bdRightToLeftNoAlign draws text using right to left reading order, the alignment is not changed, and the scroll bar is moved.

bdRightToLeftReadingOnly

bdRightToLeftReadingOnly draws text using right to left reading order, and the alignment and scroll bars are not changed.

ParentBiDiMode property

[Topic groups](#)

[See also](#)

ParentBiDiMode is a Boolean property. When *True* (the default) the control looks to its parent to determine what *BiDiMode* to use. If the control is a *TForm* object, the form uses the *BiDiMode* setting from *Application*. If all the *ParentBiDiMode* properties are *True*, when you change *Application's BiDiMode* property, all forms and controls in the project are updated with the new setting.

FlipChildren method

[Topic groups](#)

[See also](#)

The *FlipChildren* method allows you to flip the position of a container control's children. Container controls are controls that can accept other controls, such as *TForm*, *TPanel*, and *TGroupBox*. *FlipChildren* has a single boolean parameter, *AllLevels*. When *False*, only the immediate children of the container control are flipped. When *True*, all the levels of children in the container control are flipped.

Delphi flips the controls by changing the *Left* property and the alignment of the control. If a control's left side is five pixels from the left edge of its parent control, after it is flipped the control's right side is five pixels from the right edge of the parent control. If the control is left aligned, calling *FlipChildren* will make the control right aligned.

To flip a control at design-time select Edit|Flip Children and select either All or Selected, depending on whether you want to flip all the controls, or just the children of the selected control. You can also flip a control by selecting the control on the form, right-clicking, and selecting Flip Children from the context menu.

Note: Selecting an edit control and issuing a Flip Children|Selected command does nothing. This is because edit controls are not containers.

Additional methods

[Topic groups](#)

[See also](#)

There are several other methods useful for developing applications for bi-directional users.

Method	Description
OkToChangeFieldAlignment	Used with database controls. Checks to see if the alignment of a control can be changed.
DBUseRightToLeftAlignment	A wrapper for database controls for checking alignment.
ChangeBiDiModeAlignment	Changes the alignment parameter passed to it. No check is done for <i>BiDiMode</i> setting, it just converts left alignment into right alignment and vice versa, leaving center-aligned controls alone.
IsRightToLeft	Returns <i>True</i> if any of the right to left options are selected. If it returns <i>False</i> the control is in left to right mode.
UseRightToLeftReading	Returns <i>True</i> if the control is using right to left reading.
UseRightToLeftAlignment	Returns <i>True</i> if the control is using right to left alignment. It can be overridden for customization.
UseRightToLeftScrollBar	Returns <i>True</i> if the control is using a left scroll bar.
DrawTextBiDiModeFlags	Returns the correct draw text flags for the <i>BiDiMode</i> of the control.
DrawTextBiDiModeFlagsReadingOnly	Returns the correct draw text flags for the <i>BiDiMode</i> of the control, limiting the flag to its reading order.
AddBiDiModeExStyle	Adds the appropriate <i>ExStyle</i> flags to the control that is being created.

Locale-specific features

[Topic groups](#)

[See also](#)

You can add extra features to your application for specific locales. In particular, for Asian language environments, you may want your application to control the input method editor (IME) that is used to convert the keystrokes typed by the user into character strings.

VCL components offer support in programming the IME. Most windowed controls that work directly with text input have an [ImeName](#) property that allows you to specify a particular IME that should be used when the control has input focus. They also provide an [ImeMode](#) property that specifies how the IME should convert keyboard input. [ImeName](#) introduces several protected methods that you can use to control the IME from classes you define. In addition, the global [Screen](#) variable provides information about the IMEs available on the user's system.

The global *Screen* variable (available in VCL and CLX) also provides information about the keyboard mapping installed on the user's system. You can use this to obtain locale-specific information about the environment in which your application is running.

Designing the user interface

[Topic groups](#)

[See also](#)

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.

The following topics are discussed in this section:



[Text](#)



[Graphic images](#)



[Formats and sort order](#)



[Keyboard mappings](#)

Text

[Topic groups](#)

[See also](#)

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. The following table provides a rough estimate of how much expansion you should plan for given the length of your English strings:

<u>Length of English string (in characters)</u>	<u>Expected increase</u>
1-5	100%
6-12	80%
13-20	60%
21-30	40%
31-50	20%
over 50	10%

Graphic images

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

The date, time, number, and currency formats used in your application should be localized for the target locale. If you use only the Windows formats, there is no need to translate formats, as these are taken from the user's Windows Registry. However, 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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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. Delphi automatically creates a .dfm (.xfm in CLX applications) 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.

For information on using resource DLLs in your applications see "[Creating resource DLLs](#)" and "[Using resource DLLs](#)."

Creating resource DLLs

[Topic groups](#)

[See also](#)

Isolating resources simplifies the translation process. The next level of resource separation is the creation of a resource DLL. A resource DLL contains all the resources and only the resources for a program. Resource DLLs allow you to create a program that supports many translations simply by swapping the resource DLL.

Use the Resource DLL wizard to create a resource DLL for your program. The Resource DLL wizard requires an open, saved, compiled project. It will create an RC file that contains the string tables from used RC files and **resourcestring** strings of the project, and generate a project for a resource only DLL that contains the relevant forms and the created RES file. The RES file is compiled from the new RC file.

You should create a resource DLL for each translation you want to support. Each resource DLL should have a file name extension specific to the target locale. The first two characters indicate the target language, and the third character indicates the country of the locale. If you use the Resource DLL wizard, this is handled for you. Otherwise, use the following code to obtain the locale code for the target translation:

```
unit locales;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
type
  TForm1 = class(TForm)
    Button1: TButton;
    LocaleList: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
function GetLocaleData(ID: LCID; Flag: DWORD): string;
var
  BufSize: Integer;
begin
  BufSize := GetLocaleInfo(ID, Flag, nil, 0);
  SetLength(Result, BufSize);
  GetLocaleInfo(ID, Flag, PChar(Result), BufSize);
  SetLength(Result, BufSize - 1);
end;
{ Called for each supported locale. }
function LocalesCallback(Name: PChar): Bool; stdcall;
var
  LCID: Integer;
begin
  LCID := StrToInt('$' + Copy(Name, 5, 4));
  Form1.LocaleList.Items.Add(GetLocaleData(LCID, LOCALE_SLANGUAGE));
  Result := Bool(1);
end;
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  with Languages do
  begin
    for I := 0 to Count - 1 do
    begin
```

```
        ListBox1.Items.Add(Name[I]);  
    end;  
end;  
end;
```

Using resource DLLs

[Topic groups](#)

[See also](#)

The executable, DLLs, and packages that make up your application contain all the necessary resources. However, to replace those resources by localized versions, you need only ship your application with localized resource DLLs that have the same name as your EXE, DLL, or BPL files.

When your application starts up, it checks the locale of the local system. If it finds any resource DLLs with the same name as the EXE, DLL, or BPL files it is using, it checks the extension on those DLLs. 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, DLL, or package. If there is not a resource module that matches both the language and the country, your application will try to locate a resource module that matches just the language. If there is no resource module that matches the language, your application will use the resources compiled with the executable, DLL, or package.

If you want your application to use a different resource module than the one that matches the locale of the local system, you can set a locale override entry in the Windows registry. Under the HKEY_CURRENT_USER\Software\Borland\Locales key, add your application's path and file name as a string value and set the data value to the extension of your resource DLLs. At startup, the application will look for resource DLLs with this extension before trying the system locale. Setting this registry entry allows you to test localized versions of your application without changing the locale on your system.

For example, the following procedure can be used in an install or setup program to set the registry key value that indicates the locale to use when loading Delphi applications:

```
procedure SetLocalOverrides(FileName: string, LocaleOverride: string);  
var  
    Reg: TRegistry;  
begin  
    Reg := TRegistry.Create;  
    try  
        if Reg.OpenKey('Software\Borland\Locales', True) then  
            Reg.WriteString(LocaleOverride, FileName);  
    finally  
        Reg.Free;  
end;
```

Within your application, use the global *FindResourceHInstance* function to obtain the handle of the current resource module. For example:

```
LoadStr(FindResourceHInstance(HInstance), IDS_AmountDueName, szQuery,  
        SizeOf(szQuery));
```

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 DLLs.

Dynamic switching of resource DLLs

[Topic groups](#)

[See also](#)

In addition to locating a resource DLL at application startup, it is possible to switch resource DLLs dynamically at runtime. To add this functionality to your own applications, you need to include the `ReInit` unit in your **uses** statement. (`ReInit` is located in the `Richedit` sample in the `Demos` directory.) To switch languages, you should call `LoadResourceModule`, passing the LCID for the new language, and then call `ReinitializeForms`.

For example, the following code switches the interface language to French:

```
const
  FRENCH = (SUBLANG_FRENCH shl 10) or LANG_FRENCH;
if LoadNewResourceModule(FRENCH) <> 0 then
  ReinitializeForms;
```

The advantage of this technique is that the current instance of the application and all of its forms are used. It is not necessary to update the registry settings and restart the application or reacquire resources required by the application, such as logging in to database servers.

When you switch resource DLLs the properties specified in the new DLL overwrite the properties in the running instances of the forms.

Note: Any changes made to the form properties at runtime will be lost. Once the new DLL is loaded, default values are not reset. Avoid code that assumes that the form objects are reinitialized to their startup state, apart from differences due to localization.

Localizing applications

[Topic groups](#)

[See also](#)

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 DLL that contains form files (.dfm or .xfm) and a resource file. You can open your forms in the IDE and translate the relevant properties.

Note: In a resource DLL project, you cannot add or delete components. It is possible, however, to change properties in ways that could cause runtime errors, so be careful to modify only those properties that require translation. To avoid mistakes, you can configure the Object Inspector to display only localizable properties; to do so, right-click in the Object Inspector and use the View menu to filter out unwanted property categories.

You can open the RC file and translate relevant strings. Use the StringTable editor by opening the RC file from the Project Manager.

Deploying applications

[Topic groups](#)

[See also](#)

Once your Delphi 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. What is required by a given application varies, depending on the type of application. The following sections describe considerations when deploying different types of applications:



[Deploying general applications](#)



[Deploying CLX applications](#)



[Deploying database applications](#)



[Deploying Web applications](#)



[Programming for varying host environments](#)



[Software license requirements](#)

Note: Information included in these sections is for deploying applications on Windows. If writing cross-platform applications for deployment on Linux, you need to refer to deployment information provided in your Kylix documentation.

Deploying general applications

[Topic groups](#)

[See also](#)

Beyond the executable file, an application may require a number of supporting files, such as DLLs, package files, and helper applications. In addition, the Windows registry may need to contain entries for an application, from specifying the location of supporting files to simple program settings. The process of copying an application's files to a computer and making any needed registry settings can be automated by an installation program, such as InstallShield Express. These are the main deployment concerns common to nearly all types of applications:



[Using installation programs](#)



[Identifying application files](#)



[Helper applications](#)



[DLL locations](#)

Delphi 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](#). For more information on installing Web applications, see [Deploying Web applications](#). For more information on installing ActiveX controls, see [Deploying an ActiveX control on the Web](#).

Using installation programs

[Topic groups](#)

[See also](#)

Simple Delphi 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.

Setup toolkits automate the process of creating installation programs, often without needing to write any code. Installation programs created with Setup toolkits perform various tasks inherent to installing Delphi applications, including: copying the executable and supporting files to the host computer, making Windows registry entries, and installing the Borland Database Engine for BDE database applications.

InstallShield Express is a setup toolkit that is bundled with Delphi. InstallShield Express is certified for use with Delphi and the Borland Database Engine. It is based on Windows Installer (MSI) technology.

InstallShield Express is not automatically installed when Delphi is installed, so it must be manually installed if you want to use it to create installation programs. Run the installation program from the Delphi CD to install InstallShield Express. For more information on using InstallShield Express to create installation programs, see the InstallShield Express online help.

Other setup toolkits are available. However, if deploying BDE database applications, you should only use toolkits based on MSI technology and those which are certified to deploy the Borland Database Engine.

 DevGuide: [Building applications with Delphi](#)

Identifying application files

[Topic groups](#)

[See also](#)

Besides the executable file, a number of other files may need to be distributed with an application.



[Application files, listed by file name extension](#)



[Package files](#)



[Merge modules](#)



[ActiveX controls](#)

Application files, listed by file name extension

[Topic groups](#)

[See also](#)

The following types of files may need to be distributed with an application.

Type	File name extension
Program files	.exe and .dll
Package files	.bpl and .dcp
Help files	.hlp, .cnt, and .toc (if used) or any other help files your application supports
ActiveX files	.ocx (sometimes supported by a DLL)
Local table files	.dbf, .mdx, .dbt, .ndx, .db, .px, .y*, .x*, .mb, .val, .qbe, .gd*

Package files

[Topic groups](#)

[See also](#)

If the application uses runtime packages, those package files need to be distributed with the application. InstallShield Express handles the installation of package files the same as DLLs, copying the files and making necessary entries in the Windows registry. You can also use merge modules for deploying runtime packages with MSI-based setup tools including InstallShield Express. See [Merge modules](#) for details.

Borland recommends installing the runtime package files supplied by Borland in the Windows\System directory. This serves as a common location so that multiple applications would have access to a single instance of the files. For packages you created, it is recommended that you install them in the same directory as the application. Only the .BPL files need to be distributed.

Note: If deploying packages with CLX applications, you need to include clx60.bpl rather than vcl60.bpl. If you are distributing packages to other developers, supply both the .BPL and the .DCP files.

Merge modules

[Topic groups](#)

[See also](#)

InstallShield Express 3.0 is based on Windows Installer (MSI) technology. That is why Delphi includes merge modules. Merge modules provide a standard method that you can use to deliver shared code, files, resources, Registry entries, and setup logic to applications as a single compound file. You can use merge modules for deploying runtime packages with MSI-based setup tools including InstallShield Express.

The runtime libraries have some interdependencies because of the way they are grouped together. The result of this is that when one package is added to an install project, the install tool automatically adds or reports a dependency on one or more other packages. For example, if you add the VCLInternet merge module to an install project, the install tool should also automatically add or report a dependency on the VCLDatabase and StandardVCL modules.

The dependencies for each merge module are listed in the table below. The various install tools may react to these dependencies differently. The InstallShield for Windows Installer automatically adds the required modules if it can find them. Other tools may simply report a dependency or may generate a build failure if all required modules are not included in the project.

<u>Merge module</u>	<u>BPLs included</u>	<u>Dependencies</u>
ADORTL	adortl60.bpl	DatabaseRTL, BaseRTL
BaseClientDataSet	cds60.bpl	DatabaseRTL, BaseRTL, DataSnap, dbExpress
BaseRTL	rtl60.bpl	No dependencies
BaseVCL	vcl60.bpl, vclx60.bpl	BaseRTL
BDEClientDataSet	bdecds60.bpl	BaseClientDataSet, DatabaseRTL, BaseRTL, DataSnap, dbExpress, BDERTL
BDEInternet	inetdbbde60.bpl	Internet, DatabaseRTL, BaseRTL, BDERTL
BDERTL	bdertl60.bpl	DatabaseRTL, BaseRTL
DatabaseRTL	dbrtl60.bpl	BaseRTL
DatabaseVCL	vcldb60.bpl	BaseVCL, DatabaseRTL, BaseRTL
DataSnap	dsn60.bpl	DatabaseRTL, BaseRTL
DataSnapConnection	dsncon60.bpl	DataSnap, DatabaseRTL, BaseRTL
DataSnapCorba	dsnacrba60.bpl	DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL, BaseVCL
DataSnapEntera	dsnapent60.bpl	DataSnap, DatabaseRTL, BaseRTL, BaseVCL
DBCompatVCL	vcldbx60.bpl	DatabaseVCL, BaseVCL, BaseRTL
dbExpress	dbexpress60.bpl	DatabaseRTL, BaseRTL
dbExpressClientDataSet	dbxcds60.bpl	BaseClientDataSet, DatabaseRTL, BaseRTL, DataSnap, dbExpress
DBXInternet	inetdbxpress60.bpl	Internet, DatabaseRTL, BaseRTL, dbExpress, DatabaseVCL, BaseVCL
DecisionCube	dss60.bpl	TeeChart, BaseVCL, BaseRTL, DatabaseVCL, DatabaseRTL, BDERTL
FastNet	nmfast60.bpl	BaseVCL, BaseRTL
InterbaseVCL	vclib60.bpl	BaseClientDataSet, DatabaseRTL, BaseRTL, DataSnap, dbExpress, BaseVCL
Internet	inet60.bpl, inetdb60.bpl	DatabaseRTL, BaseRTL
InternetDirect	indy60.bpl	BaseVCL, BaseRTL
Office2000Components	dcloffice2k60.bpl	DatabaseVCL, BaseVCL, DatabaseRTL, BaseRTL
QuickReport	qrpt60.bpl	BaseVCL, BaseRTL, BDERTL, DatabaseRTL
SampleVCL	vclsmp60.bpl	BaseVCL, BaseRTL
TeeChart	tee60.bpl, teedb60.bpl, teeqr60.bpl, teeui60.bpl	BaseVCL, BaseRTL
VCLIE	vclie60.bpl	BaseVCL, BaseRTL
VisualCLX	visualclx60.bpl	BaseRTL
WebDataSnap	webdsn60.bpl	XMLRTL, Internet, DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL
WebSnap	websnap60.bpl, vcljpg60.bpl	WebDataSnap, XMLRTL, Internet, DataSnapConnection, DataSnap, DatabaseRTL,

XMLRTL

xmIrtI60.bpl

BaseRTL, BaseVCL
Internet, DatabaseRTL, BaseRTL

ActiveX controls

[Topic groups](#)

[See also](#)

Certain components bundled with Delphi are ActiveX controls. The component wrapper is linked into the application's executable file (or a runtime package), but the .OCX file for the component also needs to be deployed with the application. These components include

-  Chart FX, copyright SoftwareFX Inc.
-  VisualSpeller Control, copyright Visual Components, Inc.
-  Formula One (spreadsheet), copyright Visual Components, Inc.
-  First Impression (VtChart), copyright Visual Components, Inc.
-  Graph Custom Control, copyright Bits Per Second Ltd.

ActiveX controls of your own creation need to be registered on the deployment computer before use. Installation programs such as InstallShield Express automate this registration process. To manually register an ActiveX control, use the TRegSvr demo application or the Microsoft utility REGSRV32.EXE (not included with all Windows versions).

DLLs that support an ActiveX control also need to be distributed with an application.

Helper applications

[Topic groups](#)

[See also](#)

Helper applications are separate programs without which your Delphi application would be partially or completely unable to function. Helper applications may be those supplied with the operating system, by Borland, or they might be third-party products. An example of a helper application is the InterBase utility program Server Manager, which administers InterBase databases, users, and security.

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 for the helper for specific information.

DLL locations

[Topic groups](#)

[See also](#)

You can install .dll files used only by a single application in the same directory as the application. DLLs 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 DLLs is to place them either in the Windows or the Windows\System directory. A better way is to create a dedicated directory for the common .dll files, similar to the way the Borland Database Engine is installed.

Deploying CLX applications

[Topic groups](#)

[See also](#)

If you are writing cross-platform applications that will be deployed on both Windows and Linux, you need to compile and deploy the applications on both platforms. The steps for deploying CLX applications are the same as those for general applications. For information on deploying general applications, see [Deploying general applications](#). For information on installing database CLX applications, see [Deploying database applications](#).

Note: When deploying CLX applications on Windows, you need to include qtintf.dll with the application to include the CLX runtime. If deploying packages with CLX applications, you need to include clx60.bpl rather than vcl60.bpl.

See [Developing cross-platform applications](#) for information on writing CLX applications.

Deploying database applications

[Topic groups](#)

[See also](#)

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 even more specialized handling on installation, because the files that make up the application are typically located on multiple computers.

Since several different database technologies (ADO, BDE, dbExpress, and InterBase Express) are supported, deployment requirements differ for each. Regardless of which you are using, you need to make sure that the client side software is installed on the system where you plan to run the database application. BDE, ADO, and dbExpress also require drivers to interact with the client-side software of the database. InterBase does not require drivers because the IBX components communicate directly with the database.

Specific information on how to deploy dbExpress, BDE, and multi-tiered database applications is described in the following topics:



[Deploying dbExpress database applications](#)



[Deploying BDE applications](#)



[Deploying multi-tiered database applications \(DataSnap\)](#)

Database applications that use client datasets such as *TClientDataSet* or *TSQLClientDataSet* or dataset providers require you to include `libmidas.dcu` and `crtl.dcu` (for static linking when providing a standalone executable); if you are packaging your application (with the executable and any needed DLLs), you need to include `Midas.dll`.

If deploying database applications that use ADO, you need to be sure that MDAC version 2.1 or later is installed on the system where you plan to run the application. MDAC is automatically installed with software such as Windows 2000 and Internet Explorer version 5 or later. You also need to be sure the drivers for the database server you want to connect to are installed on the client. No other deployment steps are required.

If deploying database applications that use InterBase Express, you need to be sure that the InterBase client is installed on the system where you plan to run the application. InterBase requires `gd32.dll` and `interbase.msg` to be located in an accessible directory. No other deployment steps are required. InterBase Express components communicate directly with the database and do not require additional drivers. For more information, refer to the Embedded Installation Guide posted on the Borland Web site.

In addition to the technologies described here, you can also use third-party database engines to provide database access for Delphi applications. Consult the documentation or vendor for the database engine regarding redistribution rights, installation, and configuration.

Deploying dbExpress database applications

[Topic groups](#)

[See also](#)

dbExpress is a set of drivers that provide fast access to database information. dbExpress components support cross-platform development because they are also available on Linux. Refer to [Using unidirectional datasets](#) for more information about using the dbExpress components.

You can deploy dbExpress applications either as a stand-alone executable file or as an executable file that includes associated dbExpress driver DLLs.

To deploy dbExpress applications as standalone executable files, the dbExpress object files must be statically linked into your executable. You do this by including the following DCUs, located in the lib directory:

<u>Database unit</u>	<u>When to include</u>
dbExpInt	Applications connecting to InterBase databases
dbExpOra	Applications connecting to Oracle databases
dbExpDb2	Applications connecting to DB2 databases
dbExpMy	Applications connecting to MySQL databases
CrtI, MidasLib	Required by dbExpress executables that use client datasets such as <i>TSQLClientDataSet</i>

If you are not deploying a standalone executable, you can deploy associated dbExpress drivers and DataSnap DLLs with your executable. The following table lists the appropriate DLLs and when to include them:

<u>Database DLL</u>	<u>When to deploy</u>
dbexpint.dll	Applications connecting to InterBase databases
dbexpora.dll	Applications connecting to Oracle databases
dbexpdb2.dll	Applications connecting to DB2 databases
dbexpmy.dll	Applications connecting to MySQL databases
Midas.dll	Required by database applications that use client datasets

Deploying BDE applications

[Topic groups](#)

[See also](#)

The Borland Database Engine (BDE) defines a large API for interacting with databases. Of all the data access mechanisms, the BDE supports the broadest range of functions and comes with the most supporting utilities. It is the best way to work with data in Paradox or dBASE tables.

Database access for an application is provided by various database engines. An application can use the BDE or a third-party database engine. SQL Links is provided (not available in all versions) to enable native access to SQL database systems. The following topics describe installation of the database access elements of an application:



[Borland Database Engine](#)



[SQL Links](#)

Borland Database Engine

[Topic groups](#)

[See also](#)

For standard Delphi data components to have database access, the Borland Database Engine (BDE) must be present and accessible. See the BDEDEPLOY document for specific rights and limitations on redistributing the BDE.

Borland recommends use of InstallShield Express (or other certified installation program) for installing the BDE. InstallShield Express will create the necessary registry entries and define any aliases the application may require. Using a certified installation program to deploy the BDE files and subsets is important because:

 Improper installation of the BDE or BDE subsets can cause other applications using the BDE to fail. Such applications include not only Borland products, but many third-party programs that use the BDE.

 Under Windows 9x and Windows NT, BDE configuration information is stored in the Windows registry instead of .INI files, as was the case under 16-bit Windows. Making the correct entries and deletions for install and uninstall is a complex task.

It is possible to install only as much of the BDE as an application actually needs. For instance, if an application only uses Paradox tables, it is only necessary to install that portion of the BDE required to access Paradox tables. This reduces the disk space needed for an application. Certified installation programs, like InstallShield Express, are capable of performing partial BDE installations. Be sure to leave BDE system files that are not used by the deployed application, but that are needed by other programs.

SQL Links

[Topic groups](#)

[See also](#)

SQL Links provides the drivers that connect an application (through the Borland Database Engine) with the client software for an SQL database. See the DEPLOY document for specific rights and limitations on redistributing SQL Links. As is the case with the Borland Database Engine (BDE), SQL Links must be deployed using InstallShield Express (or other certified installation program).

Note: SQL Links only connects the BDE to the client software, not to the SQL database itself. It is still necessary to install the client software for the SQL database system used. See the documentation for the SQL database system or consult the vendor that supplies it for more information on installing and configuring client software.

The following table shows the names of the driver and configuration files SQL Links uses to connect to the different SQL database systems. These files come with SQL Links and are redistributable in accordance with the Delphi license agreement.

<u>Vendor</u>	<u>Redistributable files</u>
Oracle 7	SQLORA32.DLL and SQL_ORA.CNF
Oracle8	SQLORA8.DLL and SQL_ORA8.CNF
Sybase Db-Lib	SQLSYB32.DLL and SQL_SYB.CNF
Sybase Ct-Lib	SQLSSC32.DLL and SQL_SSC.CNF
Microsoft SQL Server	SQLMSS32.DLL and SQL_MSS.CNF
Informix 7	SQLINF32.DLL and SQL_INF.CNF
Informix 9	SQLINF9.DLL and SQL_INF9.CNF
DB/2	SQLDB232.DLL and SQL_DB2.CNF
InterBase	SQLINT32.DLL and SQL_INT.CNF

Install SQL Links using InstallShield Express or other certified installation program. For specific information concerning the installation and configuration of SQL Links, see the help file SQLLNK32.HLP, by default installed into the main BDE directory.

 DevGuide: [Building applications with Delphi](#)

Deploying multi-tiered database applications (DataSnap)

[Topic groups](#)

[See also](#)

DataSnap provides multi-tier database capability to Delphi applications by allowing client applications to connect to providers in an application server.

Install DataSnap along with a multi-tier application using InstallShield Express (or other Borland-certified installation scripting utility). See the DEPLOY document (found in the main Delphi directory) for details on the files that need to be redistributed with an application. Also see the REMOTE document for related information on what DataSnap files can be redistributed and how.

Deploying Web applications

[Topic groups](#)

[See also](#)

Some Delphi applications are designed to be run over the World Wide Web, such as those in the form of Server-side Extension DLLs (ISAPI and Apache), CGI applications, and ActiveForms.

The steps for deploying 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](#). For information on deploying database Web applications, see [Deploying database applications](#).

Here are some special considerations for deploying Web applications:

-  For BDE database applications, the Borland Database Engine (or alternate database engine) is installed with the application files on the Web server.
 -  For dbExpress applications, the dbExpress DLLs must be included in the path. If included, the *dbExpress* driver is installed with the application files on the Web server.
 -  Security for the directories should be set so that the application can access all needed database files.
 -  The directory containing an application must have read and execute attributes.
 -  The application should not use hard-coded paths for accessing database or other files.
 -  The location of an ActiveX control is indicated by the CODEBASE parameter of the <OBJECT> HTML tag.
- For information on deploying applications on Apache servers, see [Deployment on Apache](#).

Deployment on Apache

[Topic groups](#)

[See also](#)

WebBroker supports Apache version 1.3.9 and later for DLLs and CGI applications. Apache is configured by files in the conf directory.

If creating Apache DLLs, you need to be sure to set appropriate directives in the Apache server configuration file, called httpd.conf. The DLL should be physically located in the Modules subdirectory of the Apache software.

If creating CGI applications, the physical directory (specified in the Directory directive of the httpd.conf file) must have the ExecCGI option set to allow execution of programs so the CGI script can be executed. To ensure that permissions are set up properly, you need to either use the ScriptAlias directive or set Options ExecCGI to on.

The ScriptAlias directive creates a virtual directory on your server and marks the target directory as containing CGI scripts. For example, you could add the following line to your httpd.conf file:

```
ScriptAlias /cgi-bin "c:\inetpub\cgi-bin"
```

This would cause requests such as /cgi-bin/mycgi to be satisfied by running the script c:\inetpub\cgi-bin\mycgi.

You can also set Options to All or to ExecCGI using the Directory directive in httpd.conf. The Options directive controls which server features are available in a particular directory. Directory directives are used to enclose a set of directives that apply to the named directory and its subdirectories. An example of the Directory directive is shown below:

```
<Directory <apache-root-dir>\cgi-bin>
    AllowOverride None
    Options ExecCGI
    Order allow,deny
    Allow from all
</Directory>
```

In this example, Options is set to ExecCGI permitting execution of CGI scripts in the directory cgi-bin.

Note: Apache executes locally on the server within the account specified in the User directive in the httpd.conf file. Make sure that the user has the appropriate rights to access the resources needed by the application.

Information concerning the deployment of Apache software can be found in the Apache LICENSE file, which is included in the Apache distribution. You can also find configuration information on the Apache Web site at www.apache.org.

 DevGuide: [Building applications with Delphi](#)

Programming for varying host environments

[Topic groups](#)

[See also](#)

Due to the nature of various operating system environments, 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](#)



[Windows versions](#)

Screen resolutions and color depths

[Topic groups](#)

[See also](#)

The size of the desktop and number of available colors on a computer is configurable and dependent on the hardware 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.

The following topics are discussed in this section:

 [Considerations when not dynamically resizing](#)

 [Considerations when dynamically resizing forms and controls](#)

 [Accommodating varying color depths](#)

Considerations when not dynamically resizing

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

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 is 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* Screen object's [Height](#) or *TScreen.Width* [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* form's [Scaled](#) property to *True* and calling *TWinControl.ScaleBy* its [ScaleBy](#) method (*TWidgetControl.ScaleBy* for cross-platform applications). 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 *TWinControl.ScaleBy* method (*TWidgetControl.ScaleBy* for cross-platform applications), 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

[Topic groups](#)

[See also](#)

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

[Topic groups](#)

[See also](#)

The Windows and Linux operating systems come with a standard sets of fonts. When designing an application to be deployed on other computers, realize that not all computers will have fonts outside the standard sets.

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.

Windows has a safety measure to account for attempts to use a font that does not exist on the computer. It substitutes another, existing font that it considers the closest match. While this may circumvent errors concerning missing fonts, the end result may be a degradation of the visual appearance of the application. It is better to prepare for this eventuality at design time.

To make a nonstandard font available to a Windows application, use the Windows API functions *AddFontResource* and *DeleteFontResource*. Deploy the .fot file for the nonstandard font with the application.

Windows versions

[Topic groups](#)

[See also](#)

When using operating system APIs or accessing areas of the operating system from an application, there is the possibility that the function, operation, or area may not be available on computers with different operating system versions.

To account for this possibility, you have a few options:

 Specify in the application's system requirements the versions of the operating system on which the application can run. It is the user's responsibility to install and use the application only under compatible operating system versions.

 Check the version of the operating system as the application is installed. If an incompatible version of the operating system is present, either halt the installation process or at least warn the installer of the problem.

 Check the operating system version at runtime, just prior to executing an operation not applicable to all versions. If an incompatible version of the operating system is present, abort the process and alert the user. Alternately, provide different code to run dependent on different operating system versions. For example, some operations are performed differently on Windows 95/98 than on Windows NT/2000. Use the Windows API function *GetVersionEx* to determine the Windows version.

Software license requirements

[Topic groups](#)

[See also](#)

The distribution of some files associated with Delphi 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

DEPLOY covers the some of the legal aspects of distributing of various components and utilities, and other product areas that can be part of or associated with your application. DEPLOY is a document installed in the main directory. The topics covered include, but are not limited to



.EXE, .DLL, and .BPL files



Components and design-time packages



Borland Database Engine (BDE)



ActiveX controls



Sample Images



SQL Links



README

README contains last minute information about Delphi possibly including information that could affect the redistribution rights for components, or utilities, or other product areas. README is a document installed into the main Delphi directory.



No-nonsense license agreement

The Delphi no-nonsense license agreement, a printed document, covers other legal rights and obligations concerning Delphi.



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 Delphi applications prior to distribution.

Building applications with Delphi

- [The Delphi Development Environment](#)
- [Building applications, components, and libraries](#)
- [Working with controls](#)
- [Deploying applications](#)
- [Developing the application user interface](#)
- [Working with graphics](#)
- [Creating international applications](#)
- [Working with multimedia](#)
- [Working with packages and components](#)
- [Common programming tasks](#)
- [Writing multi-threaded applications](#)
- [Using Object Pascal with the VCL](#)
- [Developing cross-platform applications](#)

The Delphi Development Environment

[Related topic groups](#)

- [Integrated development environment](#)
- [Using the component libraries](#)
- [Designing applications](#)
- [Properties, methods, and events](#)
- [Types of events](#)
- [Object Pascal and the class libraries](#)
- [Using the object model](#)
- [What is an object?](#)
- [Examining a Delphi object](#)
- [Changing the name of a component](#)
- [Inheriting data and code from an object](#)
- [Scope and qualifiers](#)
- [Private, protected, public, and published declarations](#)
- [Using object variables](#)
- [Creating, instantiating, and destroying objects](#)
- [Components and ownership](#)
- [Objects, components, and controls](#)
- [TObject Branch](#)
- [TPersistent Branch](#)
- [TComponent Branch](#)
- [TControl Branch](#)
- [TWinControl Branch](#)
- [Properties common to TControl](#)
- [Action properties](#)
- [Position, size, and alignment properties](#)
- [TControl display properties](#)
- [Parent properties](#)
- [A navigation property](#)
- [Drag-and-drop properties](#)
- [TControl drag-and-dock properties \(VCL only\)](#)
- [Standard events common to TControl](#)
- [Properties common to TWinControl and TWidgetControl](#)
- [TWinControl General Information Properties](#)
- [Border Style Display Properties](#)
- [Navigation properties](#)
- [TWinControl drag-and-dock properties \(VCL only\)](#)
- [Standard events common to TWinControl and TWidgetControl](#)
- [Using components](#)
- [Setting component properties](#)
- [Using the Object Inspector](#)
- [Using property editors](#)
- [Setting properties at runtime](#)
- [Calling methods](#)
- [Working with events and event handlers](#)
- [Generating a new event handler](#)
- [Generating a handler for a component's default event](#)
- [Locating event handlers](#)
- [Associating an event with an existing event handler](#)
- [Using the Sender parameter](#)

- Displaying and coding shared events
- Associating menu events with event handlers
- Deleting event handlers
- VCL and CLX components
- Adding custom components to the component palette
- Text controls
- Text control properties
- Properties of memo and rich text controls
- Rich text controls (VCL only)
- Specialized input controls
- Scroll bars
- Track bars
- Up-down controls (VCL only)
- Spin edit controls (CLX only)
- Hot key controls (VCL only)
- Splitter control
- Buttons and similar controls
- Button controls
- Bitmap buttons
- Speed buttons
- Check boxes
- Radio buttons
- Toolbars
- Cool bars (VCL only)
- Handling lists
- List boxes and check-list boxes
- Combo boxes
- Tree views
- List views
- Date-time pickers and month calendars (VCL only)
- Grouping components
- Group boxes and radio groups
- Panels
- Scroll boxes
- Tab controls
- Page controls
- Header controls
- Providing visual feedback
- Labels and static text components
- Status bars
- Progress bars
- Help and hint properties
- Grids
- Draw grids
- String grids
- Value list editors (VCL only)
- Displaying graphics
- Images
- Shapes
- Bevels
- Paint boxes

- [Animation control \(VCL only\)](#)
- [Developing dialog boxes](#)
- [Using Windows Common Dialog Boxes](#)
- [Using helper objects](#)
- [Working with lists](#)
- [Working with string lists](#)
- [Loading and saving string lists](#)
- [Creating a new string list](#)
- [Manipulating strings in a list](#)
- [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](#)
- [Deleting a string from a list](#)
- [Copying a complete string list](#)
- [Associating objects with a string list](#)
- [Windows registry and INI files \(VCL only\)](#)
- [Using TIniFile \(VCL only\)](#)
- [Using TRegistry \(VCL only\)](#)
- [Using TRegIniFile \(VCL only\)](#)
- [Creating drawing spaces](#)
- [Printing](#)
- [Using streams](#)

Building applications, components, and libraries

[Related topic groups](#)

- [Creating applications](#)
- [GUI applications](#)
- [User interface models](#)
- [SDI applications](#)
- [MDI applications](#)
- [Setting IDE, project, and compilation options](#)
- [Programming templates](#)
- [Console applications](#)
- [Service applications](#)
- [Service threads](#)
- [Service name properties \(Windows only\)](#)
- [Debugging service applications](#)
- [Creating packages and DLLs](#)
- [When to use packages and DLLs](#)
- [Writing database applications](#)
- [Distributing database applications](#)
- [Creating Web server applications](#)
- [Using Web Broker](#)
- [Creating WebSnap applications](#)
- [Using InternetExpress](#)
- [Creating Web Services applications](#)
- [Writing applications using COM](#)
- [Using data modules](#)
- [Creating and editing standard data modules](#)
- [Naming a data module and its unit file](#)
- [Placing and naming components](#)
- [Using component properties and events in a data module](#)
- [Creating business rules in a data module](#)
- [Accessing a data module from a form](#)
- [Adding a remote data module to an application server project](#)

Working with controls

[Related topic groups](#)

- [Implementing drag-and-drop in controls](#)
- [Ending a drag operation](#)
- [Implementing drag-and-dock in controls](#)
- [Making a windowed control a docking site](#)
- [Making a control a dockable child](#)
- [Controlling how child controls are docked](#)
- [Controlling how child controls are undocked](#)
- [Controlling how child controls respond to drag-and-dock operations](#)
- [Working with text in controls](#)
- [Setting text alignment](#)
- [Adding scroll bars 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](#)
- [Adding graphics to controls](#)
- [Indicating that a control is owner-drawn](#)
- [Adding graphical objects to a string list](#)
- [Adding images to an application](#)
- [Adding images to a string list](#)
- [Drawing owner-drawn items](#)
- [Sizing owner-draw items](#)
- [Drawing owner-draw items](#)

Deploying applications

[Related topic groups](#)

- [Deploying applications: Overview](#)
- [Deploying general applications](#)
- [Using installation programs](#)
- [Identifying application files](#)
- [Application files, listed by file name extension](#)
- [Package files](#)
- [Merge modules](#)
- [ActiveX controls](#)
- [Helper applications](#)
- [DLL locations](#)
- [Deploying CLX applications](#)
- [Deploying database applications](#)
- [Deploying dbExpress database applications](#)
- [Deploying BDE applications](#)
- [Borland Database Engine](#)
- [SQL Links](#)
- [Deploying multi-tiered database applications \(DataSnap\)](#)
- [Deploying Web applications](#)
- [Deployment on Apache](#)
- [Programming for varying host environments](#)
- [Screen resolutions and color depths](#)
- [Considerations when not dynamically resizing](#)
- [Considerations when dynamically resizing forms and controls](#)
- [Accommodating varying color depths](#)
- [Fonts](#)
- [Windows versions](#)
- [Software license requirements](#)

Developing the application user interface

[Related topic groups](#)

- [Developing the application user interface: Overview](#)
- [Controlling application behavior](#)
- [Using the main form](#)
- [Adding forms](#)
- [Hiding the main form](#)
- [Working at the application level](#)
- [Handling the screen](#)
- [Managing layout](#)
- [Responding to event notification](#)
- [Using forms](#)
- [Controlling when forms reside in memory](#)
- [Displaying an auto-created form](#)
- [Creating forms dynamically](#)
- [Creating modeless forms such as windows](#)
- [Using a local variable to create a form instance](#)
- [Passing additional arguments to forms](#)
- [Retrieving data from forms](#)
- [Retrieving data from modeless forms](#)
- [Retrieving data from modal forms](#)
- [Reusing components and groups of components](#)
- [Creating and using component templates](#)
- [Working with frames](#)
- [Creating frames](#)
- [Using and modifying frames](#)
- [Creating frames](#)
- [Organizing actions for toolbars and menus](#)
- [What is an action?](#)
- [Setting up action bands](#)
- [Creating toolbars and menus](#)
- [Adding icons to menus and toolbars](#)
- [Adding icons to menus and toolbars](#)
- [Creating toolbars and menus that users can customize](#)
- [Hiding unused items and categories in action bands](#)
- [Using action lists](#)
- [Setting up action lists](#)
- [What happens when an action fires](#)
- [How actions find their targets](#)
- [Updating actions](#)
- [Predefined action classes](#)
- [Writing action components](#)
- [Registering actions](#)
- [Creating and managing menus](#)
- [Opening the Menu Designer](#)
- [Building menus](#)
- [Naming menus](#)
- [Naming the menu items](#)
- [Adding, inserting, and deleting menu items](#)
- [Specifying accelerator keys and keyboard shortcuts](#)
- [Creating submenus](#)

- [Moving menu items](#)
- [Adding images to menu items](#)
- [Viewing the menu](#)
- [Editing menu items in the Object Inspector](#)
- [Using the Menu Designer context menu](#)
- [Switching between menus at design time](#)
- [Using menu templates](#)
- [Saving a menu as a template](#)
- [Naming conventions for template menu items and event handlers](#)
- [Manipulating menu items at runtime](#)
- [Merging menus](#)
- [Specifying the active menu: Menu property](#)
- [Determining the order of merged menu items: GroupIndex property](#)
- [Importing resource files](#)
- [Designing toolbars and cool bars](#)
- [Adding a toolbar using a panel component](#)
- [Adding a speed button to a panel](#)
- [Assigning a speed button's glyph](#)
- [Setting the initial condition of a speed button](#)
- [Creating a group of speed buttons](#)
- [Allowing toggle buttons](#)
- [Adding a toolbar using the toolbar component](#)
- [Adding a tool button](#)
- [Assigning images to tool buttons](#)
- [Setting tool button appearance and initial conditions](#)
- [Creating groups of tool buttons](#)
- [Allowing toggled tool buttons](#)
- [Adding a cool bar component](#)
- [Setting the appearance of the cool bar](#)
- [Responding to clicks](#)
- [Assigning a menu to a tool button](#)
- [Adding hidden toolbars](#)
- [Hiding and showing toolbars](#)
- [Demo programs](#)

Working with graphics

[Related topic groups](#)

- [Working with graphics and multimedia](#)
- [Overview of graphics programming](#)
- [Refreshing the screen](#)
- [Types of graphic objects](#)
- [Common properties and methods of Canvas](#)
- [Using the properties of the Canvas object](#)
- [Using pens](#)
- [Changing the pen color](#)
- [Changing the pen width](#)
- [Changing the pen style](#)
- [Changing the pen mode](#)
- [Getting the pen position](#)
- [Using brushes](#)
- [Changing the brush color](#)
- [Changing the brush style](#)
- [Setting the Brush Bitmap property](#)
- [Reading and setting pixels](#)
- [Using Canvas methods to draw graphic objects](#)
- [Drawing lines and polylines](#)
- [Drawing lines](#)
- [Drawing polylines](#)
- [Drawing shapes](#)
- [Drawing rectangles and ellipses](#)
- [Drawing rounded rectangles](#)
- [Drawing polygons](#)
- [Handling multiple drawing objects in your application](#)
- [Keeping track of which drawing tool to use](#)
- [Changing the tool with speed buttons](#)
- [Using drawing tools](#)
- [Drawing shapes](#)
- [Sharing code among event handlers](#)
- [Drawing on a graphic](#)
- [Making scrollable graphics](#)
- [Adding an image control](#)
- [Placing the control](#)
- [Setting the initial bitmap size](#)
- [Drawing on the bitmap](#)
- [Loading and saving graphics files](#)
- [Loading a picture from a file](#)
- [Saving a picture to a file](#)
- [Replacing the picture](#)
- [Using the clipboard with graphics](#)
- [Copying graphics to the clipboard](#)
- [Cutting graphics to the clipboard](#)
- [Pasting graphics from the clipboard](#)
- [Rubber banding example](#)
- [Responding to the mouse](#)
- [What's in a mouse event](#)
- [Responding to a mouse-down action](#)

- Responding to a mouse-up action
- Responding to a mouse move
- Adding a field to a form object to track mouse actions
- Refining line drawing
- Tracking the origin point
- Tracking movement
- Working with multimedia

Creating international applications

[Related topic groups](#)

- [Creating international applications: Overview](#)
- [Internationalization and localization](#)
- [Internationalization](#)
- [Localization](#)
- [Internationalizing applications](#)
- [Enabling application code](#)
- [Character sets](#)
- [OEM and ANSI character sets](#)
- [Double byte character sets](#)
- [Wide characters](#)
- [Including bi-directional functionality in applications](#)
- [Bi-directional properties](#)
- [ParentBiDiMode property](#)
- [FlipChildren method](#)
- [Additional methods](#)
- [Locale-specific features](#)
- [Designing the user interface](#)
- [Text](#)
- [Graphic images](#)
- [Formats and sort order](#)
- [Keyboard mappings](#)
- [Isolating resources](#)
- [Creating resource DLLs](#)
- [Using resource DLLs](#)
- [Dynamic switching of resource DLLs](#)
- [Localizing applications](#)

Working with multimedia

[Related topic groups](#)

- [Adding silent video clips to an application](#)
- [Example of adding silent video clips](#)
- [Adding audio and/or video clips to an application](#)
- [Example of adding audio and/or video clips \(VCL only\)](#)

Working with packages and components

[Related topic groups](#)

- [Working with packages and components: Overview](#)
- [Why use packages?](#)
- [Packages and standard DLLs](#)
- [Runtime packages](#)
- [Using packages in an application](#)
- [Dynamically loading packages](#)
- [Deciding which runtime packages to use](#)
- [Custom packages](#)
- [Design-time packages](#)
- [Installing component packages](#)
- [Creating and editing packages](#)
- [Creating a package](#)
- [Editing an existing package](#)
- [Editing package source files manually](#)
- [Understanding the structure of a package](#)
- [Compiling packages](#)
- [Package-specific compiler directives](#)
- [Weak packaging](#)
- [Using the command-line compiler and linker](#)
- [Package files created by a successful compilation](#)
- [Deploying packages](#)
- [Package collection files](#)

Common programming tasks

[Related topic groups](#)

- [Common programming tasks](#)
- [Understanding classes](#)
- [Defining classes](#)
- [Handling exceptions](#)
- [Protecting blocks of code](#)
- [Responding to exceptions](#)
- [Exceptions and the flow of control](#)
- [Nesting exception responses](#)
- [Protecting resource allocations](#)
- [What kind of resources need protection?](#)
- [Creating a resource protection block](#)
- [Handling RTL exceptions](#)
- [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](#)
- [Handling component exceptions](#)
- [Exception handling with external sources](#)
- [Silent exceptions](#)
- [Defining your own exceptions](#)
- [Declaring an exception object type](#)
- [Raising an exception](#)
- [Using interfaces](#)
- [Interfaces as a language feature](#)
- [Implementing interfaces across the hierarchy](#)
- [Using interfaces with procedures](#)
- [Implementing IInterface](#)
- [IInterfacedObject](#)
- [Using the as operator](#)
- [Reusing code and delegation](#)
- [Using implements for delegation](#)
- [Aggregation](#)
- [Memory management of interface objects](#)
- [Using reference counting](#)
- [Not using reference counting](#)
- [Using interfaces in distributed applications \(VCL only\)](#)
- [Defining custom variants](#)
- [Storing a custom variant type's data](#)
- [Creating a class to enable the custom variant type](#)
- [Enabling casting](#)
- [Implementing binary operations](#)
- [Implementing binary operations](#)
- [Implementing unary operations](#)
- [Copying and clearing custom variants](#)
- [Loading and saving custom variant values](#)

- [Using the TCustomVariantType descendant](#)
- [Writing utilities to work with a custom variant type](#)
- [Supporting properties and methods in custom variants](#)
- [Working with strings](#)
- [Character types](#)
- [String types](#)
- [Short strings](#)
- [Long strings](#)
- [WideString](#)
- [PChar types](#)
- [OpenString](#)
- [Runtime library string handling routines](#)
- [Wide character routines](#)
- [Commonly used long string routines](#)
- [Declaring and initializing strings](#)
- [Mixing and converting string types](#)
- [String to PChar conversions](#)
- [String dependencies](#)
- [Returning a PChar local variable](#)
- [Passing a local variable as a PChar](#)
- [Compiler directives for strings](#)
- [Strings and characters: related topics](#)
- [Working with files](#)
- [Manipulating files](#)
- [Deleting a file](#)
- [Finding a file](#)
- [Renaming a file](#)
- [File date-time routines](#)
- [Copying a file](#)
- [File types with file I/O](#)
- [Using file streams](#)
- [Creating and opening files](#)
- [Using the file handle](#)
- [Reading and writing to files](#)
- [Reading and writing strings](#)
- [Seeking a file](#)
- [File position and size](#)
- [Copying](#)
- [Converting measurements](#)
- [Performing conversions](#)
- [Adding new measurement types](#)
- [Creating a simple conversion family and adding units](#)
- [Using a conversion function](#)
- [Using a class to manage conversions](#)
- [Defining data types](#)

Writing multi-threaded applications

[Related topic groups](#)

- [Writing multi-threaded applications](#)
- [Defining thread objects](#)
- [Initializing the thread](#)
- [Writing the thread function](#)
- [Using the main VCL/CLX thread](#)
- [Using thread-local variables](#)
- [Checking for termination by other threads](#)
- [Handling exceptions in the thread function](#)
- [Writing clean-up code](#)
- [Coordinating Threads](#)
- [Avoiding simultaneous access](#)
- [Locking objects](#)
- [Using critical sections](#)
- [Using the multi-read exclusive-write synchronizer](#)
- [Waiting for other threads](#)
- [Waiting for a thread to finish executing](#)
- [Waiting for a task to be completed](#)
- [Executing thread objects](#)
- [Overriding the default priority](#)
- [Starting and stopping threads](#)
- [Debugging multi-threaded applications](#)

Using Object Pascal with the VCL

[Related topic groups](#)

- [Using the Object Repository](#)
- [Sharing items within a project](#)
- [Adding items to the Object Repository](#)
- [Sharing objects in a team environment](#)
- [Using an Object Repository item in a project](#)
- [Copying an item](#)
- [Inheriting an item](#)
- [Using an item](#)
- [Using project templates](#)
- [Modifying shared items](#)
- [Specifying a default project, new form, and main form](#)
- [Enabling Help in applications](#)
- [Help system interfaces](#)
- [Implementing ICustomHelpViewer](#)
- [Communicating with the Help Manager](#)
- [Asking the Help Manager for information](#)
- [Displaying keyword-based Help](#)
- [Displaying tables of contents](#)
- [Implementing IExtendedHelpViewer](#)
- [Implementing IHelpSelector](#)
- [Registering Help system objects](#)
- [Using Help in a VCL Application](#)
- [How TApplication processes VCL Help](#)
- [How VCL controls process Help](#)
- [Using Help in a CLX Application](#)
- [How TApplication processes CLX Help](#)
- [How CLX controls process Help](#)
- [Calling a Help system directly](#)
- [Using IHelpSystem](#)
- [Customizing the IDE Help system](#)

Developing cross-platform applications

[Related topic groups](#)

- [Using CLX for cross-platform development](#)
- [Creating cross-platform applications](#)
- [Porting VCL applications to CLX](#)
- [Porting techniques](#)
- [Porting your application](#)
- [CLX versus VCL](#)
- [What CLX does differently](#)
- [Missing in CLX](#)
- [Features that will not port](#)
- [CLX and VCL unit comparison](#)
- [Features that will not port](#)
- [Sharing source files between Windows and Linux](#)
- [Environmental differences between Windows and Linux](#)
- [Directory structure on Linux](#)
- [Writing portable code](#)
- [Using conditional directives](#)
- [Terminating conditional directives](#)
- [Emitting messages](#)
- [Including inline assembler code](#)
- [Messages and system events](#)
- [Programming differences on Linux](#)
- [Cross-platform database applications](#)
- [dbExpress differences](#)
- [Component-level differences](#)
- [User interface-level differences](#)
- [Porting database applications to Linux](#)
- [Updating data in dbExpress applications](#)
- [Cross-platform Internet applications](#)
- [Porting Internet applications to Linux](#)

Link not found

The topic you requested is either not available or not linked to this Help system. This can occur if you launched this Help file from a system on which Delphi has not yet been installed, or if the subject matter you are requesting is not available in your edition of Delphi.



The topic you requested is now loading. If it does not appear within a few seconds, the topic is either not available or not linked to this Help system. This can occur if you launched this Help file from a system on which Delphi has not yet been installed, or if the subject matter you are requesting is not available in your edition of Delphi.

