Registering components

See also

Writing a component and its properties, methods, and events is only part of the process of component creation. Although a component with only those features can be useful, the real power of components comes from the ability to manipulate them at design time.

In addition to the tasks that make the components do their own work, making your components available at design time requires several steps:

- Registering components with Delphi
- Adding palette bitmaps
- Providing Help on properties and events
- Storing and loading properties

Not all these steps apply to every component. For example, if you don't define any new properties or events, you don't need to provide Help for them. The only step that is always necessary is registration.

See also

Overview of component creation

OOP for component writers

Creating properties

Creating events

Creating methods

Using graphics in components

Handling messages

Registering components with Delphi

See also

In order to have Delphi recognize your components and place them on the Component palette, you must register each component. Registration works on a unit basis, so if you create several components in a single unit, you register them all at once.

To register a component, add a Register procedure to the component unit, which has two aspects:

- Declaring the Register procedure
- Implementing the Register procedure

Once you've set up the registration, you can install the components into the palette as described in the User's Guide.

See also

Adding palette bitmaps

Providing Help on properties and events

Storing and loading properties

Declaring the Register procedure

See also Example

Registration involves writing a single procedure in the component unit, which must have the name Register. Register must appear in the **interface** part of the unit so Delphi can locate it. Within the Register procedure, you call the procedure RegisterComponents for each component you want to register.

Example

The following code shows the outline of a simple unit that creates and registers new component types:

```
unit MyBtns;
interface
type
  { declare your component types here }

procedure Register;       { this must appear in the interface section }

implementation
  { component implementation goes here }
procedure Register;
begin
  { register the components }
end;
end.
```

See also
Implementing the Register procedure

Implementing the Register procedure

See also Example

Inside the Register procedure for a unit containing components, you must register each component you want to add to the Component palette. If the unit contains several components, you can register them all in one step.

To register a component, call the RegisterComponents procedure once for each page of the Component palette you want to add components to. RegisterComponents tells Delphi two important things about the components it registers:

- The name of the palette page you want to install on
- The names of the components to install

The palette-page name is a string. If the name you give for the palette page doesn't already exist, Delphi creates a new page with that name. Delphi stores the names of the standard pages in string-list resources so that international versions of the product can name the pages in their native languages. If you want to install a component on one of the standard pages, you should obtain the string for the page name by calling the LoadStr function, passing the constant representing the string resource for that page, such as srSystem for the System page.

You pass the component names in an open array, which you can construct inside the call to RegisterComponents.

Example

The following Register procedure registers a component named TMyComponent and places it on a Component palette page called "Miscellaneous":

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TMyComponent]);
end;
```

You could also register several components on the same page at once, or register components on different pages, as shown in the following code:

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TFirst, TSecond]);       { two on this page... }
  RegisterComponents('Assorted', [TThird]);      { ...one on another... }
  RegisterComponents(LoadStr(srStandard), [TFourth]);    { ...and one on the Standard page }
end;
```

See also
Declaring the Register procedure

Adding palette bitmaps

See also Example

Every component needs a bitmap to represent the component on the Component palette. If you don't specify your own bitmap, Delphi uses a default bitmap.

Since the palette bitmaps are only needed at design time, you don't compile them into the component unit. Instead, you supply them in a Windows resource file with the same name as the unit, but with the extension .DCR (for "dynamic component resource"). You can create this resource file using the bitmap editor in Delphi. Each bitmap should be 24 pixels square.

For each unit you want to install, supply a palette bitmap file, and within each palette bitmap file, supply a bitmap for each component you register. The bitmap image has the same name as the component. Keep the palette bitmap file in the same directory with the compiled unit, so Delphi can find the bitmaps when it installs the components on the palette.

Example

For example, if you create a component named TMyControl in a unit named ToolBox, you need to create a resource file called TOOLBOX.DCR that contains a bitmap called TMYCONTROL. The resource names are not case-sensitive, but by convention, they are usually in uppercase letters.

See also
Registering components with Delphi
Providing Help on properties and events
Storing and loading properties

Providing Help on properties and events

See also

When you select a component on a form, or a property or event in the Object Inspector, you can press F1 to get Help on that item. Users of your components can get the same kind of documentation for your components if you create the appropriate Help files.

Because Delphi uses a special Help engine that handles searches across multiple Help files, you can provide a small Help file with just the information on your components, and users will be able to find your documentation without having to take any special steps. Your Help becomes part of the user's overall Delphi Help system.

To provide Help to users of your components, you need to understand two things:

- How Delphi handles Help requests
- Merging your Help into Delphi

See also
Registering components with Delphi
Adding palette bitmaps
Storing and loading properties

How Delphi handles Help requests

See also Example

Delphi looks up Help requests based on keywords. That is, when a user presses F1 with a component selected in the Form Designer, Delphi turns the name of the component into a keyword (in this case, "class_" plus the name of the component type), then calls the Windows Help engine (WinHelp) to search for a Help topic that has that keyword.

Keywords are a standard part of the Windows Help system. In fact, WinHelp uses the keywords in a Help file to generate the list in the Search dialog box. Since the keywords used in context-sensitive searches, such as those for a component, are not designed for human readability, you'll enter them as alternate keywords. Alternate keywords do not appear in the Search dialog box.

Example

For example, a user searching for details on a component called TSomething might open the WinHelp Search dialog box and type TSomething, but would never use the alternate form, class_TSomething, used by the Form Designer's context search. The special keyword class_TSomething is therefore invisible to the user, to avoid cluttering the Search list.

See also
Merging your Help into Delphi
Adding special footnotes

Merging your Help into Delphi

See also

Delphi includes all the tools you need to create and merge Windows Help files, including the Windows Help Compiler, HC.EXE. The mechanics of creating Help files for your Delphi components are no different than those for creating any Help file, but there are some conventions you need to follow to be compatible with the Help for the rest of the library.

To learn how to create Help files in general, see Creating Windows Help.

To make your Help files compatible with other Delphi component Help, you need to do all four of the following tasks:

1. Creating the Help file
2. Adding special footnotes
3. Creating the keyword file
4. Merging the Help indexes

When you finish creating the Help for your components, you have several files:

- A compiled Help (.HLP) file
- A Help keyword (.KWF) file
- One or more Help source (.RTF) files
- A Help project (.HPJ) file

The compiled Help and keyword files should go in the directory with the unit that contains your components. If you distribute your components to other users, you should distribute those files along with the compiled unit (.DCU) file. The Help source and Help project files are essentially source code for the Help and keyword files, so be sure to store and archive them accordingly.

See also
How Delphi handles Help requests

Creating the Help file

See also

You can use any tools you want to create a Windows Help file. Delphi's multiple-file search engine can include material from any number of Help files. In addition to the compiled Help file, you need to have the Rich Text Format (RTF) source file available so you can generate the keyword file, although you probably won't distribute the RTF file.

When you installed Delphi, you installed a Help file named "Creating Windows Help" (CWH.HLP) that contains detailed information on how to create Help files for Windows.

To make your component's Help work with the Help for the rest of the components in the library, observe the following conventions:

1. Each component has a screen.

   The component screen should give a brief description of the component's purpose, then list separately all the properties, events, and methods available to end users. Application developers access this screen by selecting the component on a form and pressing F1. For an example of a component screen, place any component on a form and press F1.

   The component screen should have a "K" footnote for keyword searching that includes the name of the component. For example, the keyword footnote for the TMemo component reads "TMemo component."

2. Every property, event, and method that the component adds or changes significantly has a screen.

   A property, event, or method screen should indicate what component the item applies to, show the declared syntax of the item, and describe its use. Application developers see these screens either by highlighting the item in the Object Inspector and pressing F1 or by placing the test cursor in the Code Editor on the name of the item and pressing F1. To see an example of a property screen, select any item in the Object Inspector and press F1.

   The property, event, or method screen should have a "K" footnote for keyword searching includes the name of the item and what kind of item it is. For example, the keyword footnote for the Top property reads "Top property."

Each screen in the Help file will also need special footnotes that Delphi uses for its multiple-file index searches.

See also
Creating Windows Help
Adding special footnotes
Creating the keyword file

Adding special footnotes

See also

Delphi needs special search keys to be able to distinguish between the Help screens for components and other similarly-named items. You should provide the standard keyword-search items for each item ("K" footnotes), but you also need special footnotes for Delphi.

To add keywords for F1 searches from the Object Inspector or the Code Editor, add "B" footnotes to your Help file screens.

"B" footnotes are just like the "K" footnotes used for the standard WinHelp keyword search, but they are used only by the Delphi search engine. The following table shows how to create a "B" footnote for each kind of component Help screen.

Screen type	"B" footnote content	Example
Main component screen	'class_' + component type name	class_TMemo
Generic property or event	'prop_' + property name	prop_WordWrap
	'event_' + event name	event_OnChange
Component-specific property or event	'prop_' + component type name + property name	prop_TMemoWordWrap
	'event_' + component type name + event name	event_TMemoOnChange

It is important to distinguish between generic screens and component-specific screens. A generic screen is one which applies to the specific property or event in all components. For example, the Left property is identical in all components, so its search string is prop_Left. The BorderStyle property, on the other hand, is different depending on what component it belongs to, so the BorderStyle properties of specific components have their own screens. Thus, the edit box component, TEdit, has a screen for its BorderStyle property, with a search string of prop_TEditBorderStyle.

See also
Creating Windows Help
Creating the Help file
Creating the keyword file

Creating the keyword file

See also

When you have created and compiled a Help file for your components and added the special footnotes for keyword searches, you also need to generate a separate keyword file that Delphi can merge into its master search index for topic searches.

To create a keyword (.KWF) file from your Help source (RTF) file,

1. At the DOS prompt, go to the directory that contains the Help source (RTF) file.
2. Run the keyword-file generation application, KWGEN, followed by the name of the Help project (.HPJ) file for your Help file.

   For example, if your Help project file is named SPECIAL.HPJ, you would type the following at the DOS prompt:

```
KWGEN SPECIAL.HPJ
```

   When KWGEN finishes, you will have a keyword file with the same name as the Help file and project file, but with the extension .KWF.

3. Place the keyword file in the same directory with the compiled unit and Help file.

When you install your components into the Component palette, you'll also want to merge the keywords into the master search index for the Delphi Help system.

See also
Creating Windows Help
Adding special footnotes
Creating the Help file

Merging the Help indexes

See also

After you create the keyword file from the Help file for your components, you need to merge the keywords into the master Help index for Delphi.

To merge your keyword file into the Delphi master Help index,

1 Make sure you have placed your keyword (.KWF) file along with the compiled Help (.HLP) file in the directory with the compiled unit that contains your components.

2 Run the HELPINST application. HELPINST is a Windows application installed with Delphi.

When HELPINST finishes, the Delphi master help index (.HDX) file includes the keywords for your component's Help screens.

See also

Creating the Help file

Adding special footnotes

Storing and loading properties

See also

Delphi stores forms and their components in form (.DFM) files. A form file is a binary representation of the properties of a form and its components. When Delphi users add the components you write to their forms, your components must have the ability to write their properties to the form file when saved. Similarly, when loaded into Delphi or executed as part of an application, the components must restore themselves from the form file.

Most of the time you won't need to do anything to make your components work with form files: the ability to store a representation and load from it are part of the inherited behavior of components. In some cases, however, you might want to alter the way a component stores itself or the way it initializes when loaded, so you should understand something about the underlying mechanism.

These are the aspects of property storage you need to understand:

- The store-and-load mechanism
- Specifying default values
- Determining what to store
- Initializing after loading

See also

Registering components with Delphi

Adding palette bitmaps

Providing Help on properties and events

The store-and-load mechanism

See also

When an application developer designs forms, Delphi saves descriptions of the forms in a form (.DFM) file, which it later attaches to the compiled application. When a user runs the application, it reads in those descriptions.

The description of a form consists of a list of the form's properties, along with similar descriptions of each component on the form. Each component, including the form itself, is responsible for storing and loading its own description.

By default, when storing itself, a component writes the values of all its public and published properties that differ from their default values, in the order of their declaration. When loading itself, a component first constructs itself, which sets all properties to their default values, then reads the stored, non-default property values.

This default mechanism serves the needs of most components, and requires no action at all on the part of the component writer. There are several ways you can customize the storing and loading process to suit the needs of your particular components, however.

See also

Specifying default values

Determining what to store

Initializing after loading

Specifying default values

See also Example

Delphi components only save their property values if those values differ from the default values. If you don't specify otherwise, Delphi assumes a property has no default value, meaning the component always stores the property, whatever its value.

A property whose value is not set by a components constructor assumes a zero value. A zero value means whatever value the property assumes when its storage memory is set to zero. That is, numeric values default to zero, Boolean values to False, pointers to **nil**, and so on. If there is any doubt, specify the default value explicitly.

To specify a default value for a property, add the **default** directive and the new default value to the end of the property declaration.

You can also specify a default value when redeclaring a property. In fact, one reason to redeclare a property is to designate a different default value.

Note: Specifying the default value does not automatically assign that value to the property on creation of the object. You must make sure that the component's Create constructor assigns the necessary value.

Example

The following code shows a component declaration that specifies a default value for the Align property and the implementation of the component's constructor that sets the default value. In this case, the new component is a special case of the standard panel component that will be used for status bars in a window, so its default alignment should be to the bottom of its owner.

```
type
  TStatusBar = class(TPanel)
  public
    constructor Create(AOwner: TComponent); override;   { override to set new default }
  published
    property Align default alBottom;      { redeclare with new default value }
  end;
...
constructor TStatusBar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);       { perform inherited initialization }
  Align := alBottom;      { assign new default value for Align }
end;
```

See also

The store-and-load mechanism

Determining what to store

Initializing after loading

Determining what to store

See also Example

You can control whether Delphi stores each of your components' properties. By default, all properties in the **published** part of the object declaration are stored. You can choose to not store a given property at all, or designate a function that determines at run time whether to store the property.

To control whether Delphi stores a property, add the **stored** directive to the property declaration, followed by True, False, or the name of a Boolean method.

You can add a **stored** clause to the declaration or redeclaration of any property.

Example

The following code shows a component that declares three new properties. One is always stored, one is never stored, and the third is stored depending on the value of a Boolean method:

```
type
  TSampleComponent = class(TComponent)
  protected
    function StoreIt: Boolean;
  public{ normally not stored }
    property Important: Integer stored True;     { always stored }
  published{ normally stored always }
    property Unimportant: Integer stored False;  { never stored }
    property Sometimes: Integer stored StoreIt;  { storage depends on function value }
  end;
```

See also

The store-and-load mechanism

Specifying default values

Initializing after loading

Initializing after loading

See also Example

After a component reads all its property values from its stored description, it calls a virtual method called Loaded, which provides a chance to perform any initializations that might be required. The call to Loaded occurs before the form and its controls are shown, so you don't need to worry about initialization causing flicker on the screen.

To initialize a component after it loads its property values, override the Loaded method.

Note: The first thing you do in any Loaded method you write is call the inherited Loaded method. This ensures that any inherited properties are correctly initialized before you perform initializations on your own component.

Example

The following code comes from the TDatabase component. After loading, the database tries to reestablish any connections that were open at the time it was stored, and specifies how to handle any exceptions that occur while connecting.

```
procedure TDatabase.Loaded;
begin
  inherited Loaded;       { always call the inherited method first }
  Modified;{ this sets internal flags }
  try
    if FStreamedConnected then Open;     { reestablish connections }
  except
    if csDesigning in ComponentState then{ at design time... }
      Application.HandleException(Self)  { ...let the Delphi handle the exception }
    else Raise;    { otherwise, reraise }
  end;
end;
```

See also

The store-and-load mechanism

Specifying default values

Determining what to store

Introduction to component writing

See also

Delphi is not just a development environment for visually creating applications with components. It also includes everything you need to create the components for building applications in the same environment, using the same Object Pascal language.

The <Italic>Component Writer's Guide<Default ¶ Font> and its accompanying Help file (CWG.HLP) describe everything you need to know to write components for Delphi applications. The printed manual and the Help file contain the same information in different forms.

This material has two purposes:

1 To teach you how to create working components

2 To ensure that the components you write are well-behaved parts of the Delphi environment

Whether you're writing components for your own applications or for commercial distribution, this book will guide you to writing components that fit in well with any Delphi application.

See also

What's different about writing components?

What is a component?

Creating a component (summary)

Overview of component creation

What is a component?

See also

Components are the building blocks of Delphi applications. Although most components represent visible parts of a user interface, components can also represent nonvisual elements in a program, such as timers and databases.

There are three different levels at which to think about components:

The functional definition of "component"

The technical definition of "component"

The component writer's definition of "component"

See also
What's different about writing components?
Creating a component (summary)

The functional definition of "component"

See also

From the end user's perspective, a component is something to choose from the palette and use in an application by manipulating it in the Forms Designer or in code. From the component writer's perspective, however, a component is an object in code. Although there are few real restrictions on what you can do when writing a component, it's good to keep in mind what the end user expects when using the components you write.

Before you attempt to write components, we strongly recommend that you become familiar with the existing components in Delphi so you can make your components familiar to users. Your goal should be to make your components "feel" as much like other components as possible.

See also

The technical definition of "component"

The component writer's definition of "component"

The technical definition of "component"

See also

At the simplest level, a component is any object descended from the type TComponent. TComponent defines the most basic behavior that all components must have, such as the ability to appear on the Component palette and operate in the Forms Designer.

But beyond that simple definition are several larger issues. For example, although TComponent defines the basic behavior needed to operate the Delphi environment, it can't know how to handle all the specific additions you make to your components. You'll have to specify those yourself.

Although it's not difficult to create well-behaved components, it does require that you pay close attention to the standards and conventions spelled out in this book.

See also

The functional definition of "component"

The component writer's definition of "component"

The component writer's definition of "component"

See also

At a very practical level, a component is any element that can "plug into" the Delphi development environment. It can represent almost any level of complexity, from a simple addition to one of the standard components to a vast, complex interface to another hardware or software system. In short, a component can do or be anything you can create in code, as long as it fits into the component framework.

The definition of a component, then, is essentially an interface specification. This manual spells out the framework onto which you build your specialized code to make it work in Delphi.

Defining the limits of "component" is therefore like defining the limits of programming. We can't tell you every kind of component you can create, any more than we can tell you all the programs you can write in a given language. What we can do is tell you how to write your code so that it fits well in the Delphi environment.

See also

The functional definition of "component"

The technical definition of "component"

What's different about writing components?

See also

There are three important differences between the task of creating a component for use in Delphi and the more common task of creating an application that uses components:

- Component writing is nonvisual
- Component writing requires deeper knowledge of objects
- Component writing follows more conventions

See also

What is a component?

Creating a component (summary)

Component writing is nonvisual

See also

The most obvious difference between writing components and building applications with Delphi is that component writing is done strictly in code. Because the visual design of Delphi applications requires completed components, creating those components requires writing Object Pascal code.

Although you can't use the same visual tools for creating components, you can use all the programming features of the Delphi development environment, including the Code Editor, integrated debugger, and ObjectBrowser.™

See also

Component writing requires deeper knowledge of objects

Component writing follows more conventions

Component writing requires deeper knowledge of objects

See also

Other than the non-visual programming, the biggest difference between creating components and using them is that when you create a new component you need to derive a new object type from an existing one, adding new properties and methods. Component users, on the other hand, use existing components and customize their behavior at design time by changing properties and specifying responses to events.

When deriving new objects, you have access to parts of the ancestor objects unavailable to end users of those same objects. These parts intended only for component writers are collectively called the protected interface to the objects. Descendant objects also need to call on their ancestor objects for a lot of their implementation, so component writers need to be familiar with that aspect of object-oriented programming.

See also

Component writing is nonvisual

Component writing follows more conventions

Component writing follows more conventions

See also

Writing a component is a more traditional programming task than visual application creation, and there are more conventions you need to follow than when you use existing components. Probably the most important thing to do before you start writing components of your own is to really use the components that come with Delphi, to get a feeling for the obvious things like naming conventions, but also for the kinds of abilities component users will expect when they use your components.

The most important thing that component users expect of components is that they should be able to do almost anything to those components at any time. Writing components that fulfill that expectation is not difficult, but it requires some forethought and adherence to conventions.

See also

Component writing is nonvisual

Component writing requires deeper knowledge of objects

Creating a component (summary)

See also

In brief, the process of creating your own component consists of these steps:

1 Create a unit for the new component.

2 Derive a component type from an existing component type.

3 Add properties, methods, and events as needed.

4 Register your component with Delphi.

5 Create a Help file for your component and its properties, methods, and events.

All these steps are covered in detail in this Help file. When you finish, the complete component includes four files:

1 A compiled unit (.DCU file)

2 A palette bitmap (.DCR file)

3 A Help file (.HLP file)

4 A Help-keyword file (.KWF file)

Although only the first file is required, the others make your components much more useful and usable.

See also

What's different about writing components?

What is a component?

OOP for component writers

See also

Working with Delphi, you've encountered the idea that an object contains both data and code, and that you can manipulate objects both at design time and run time. In that sense, you've become a component user.

When you create new kinds of components, you deal with objects in ways that end users never need to. Before you start creating components, you need to be familiar with these topics related to object-oriented programming (OOP) in general:

- Creating new objects
- Ancestors and descendants
- Controlling access
- Dispatching methods
- Objects and pointers

This material assumes familiarity with the topics discussed in Programming with Delphi objects. If you have previously used objects in Borland's Pascal products, you should probably also read Changes in Pascal objects.

See also

Overview of component creation

Creating properties

Creating events

Creating methods

Using graphics in components

Handling messages

Registering components

Creating new objects

See also

The primary difference between component users and component writers is that users manipulate instances of objects, and writers create new types of objects. This concept is fundamental to object-oriented programming, and an understanding of the distinction is extremely important if you plan to create your own components.

The concept of types and instances is not unique to objects. Programmers continually work with types and instances, but they don't generally use that terminology. "Type" is a very common idea in Object Pascal programs, and as a programmer you generally create variables of a type. Those variables are instances of the type.

Object types are generally more complex than simple types such as Integer, but by assigning different values to instances of the same type, a user can perform quite different tasks.

For example, it's quite common for a user to create a form containing two buttons, one labeled OK and one labeled Cancel. Each is an instance of type TButton, but by assigning different values to the Text, Default, and Cancel properties and assigning different handlers to the OnClick events, the user makes the two instances do very different things.

See also

Deriving new types

Declaring a new component type

Deriving new types

See also

The purpose of defining object types is to provide a basis for useful instances. That is, the goal is to create an object that you or other users can use in different applications in different circumstances, or at least in different parts of the same application.

There are two reasons to derive new types:

- Changing type defaults to avoid repetition
- Adding new capabilities to a type

In either case, the goal is to create reusable objects. If you plan ahead and design your objects with future reuse in mind, you can save a lot of later work. Give your object types usable default values, but make them customizable.

See also
Declaring a new component type

Changing type defaults to avoid repetition

In all programming tasks, needless repetition is something to avoid. If you find yourself rewriting the same lines of code over and over, you should either place the code in a subroutine or function, or build a library of routines you'll use in many programs.

The same reasoning holds for components. If you frequently find yourself changing the same properties or making the same method calls, you should probably create a new component type that does those things by default.

For example, it's possible that each time you create an application, you find yourself adding a dialog box form to perform a particular function. Although it's not difficult to recreate the dialog box each time, it's also not necessary. You can design the dialog box once, set its properties, and then install the result onto the Component palette as a reusable component. Not only can this reduce the repetitive nature of the task, it also encourages standardization and reduces the chance of error in recreating the dialog box.

Adding new capabilities to a type

The other reason for creating a new kind of component is that you want to add capabilities not already found in the existing components. When you do that, you can either derive from an existing component type (for example, creating a specialized kind of list box) or from an abstract, base type, such as TComponent or TControl.

As a general rule, derive your new component from the type that contains the closest subset of the features you want. You can add capabilities to an object, but you can't take them away, so if an existing component type contains properties that you don't want to include in yours, you should derive from that component's ancestor.

For example, if you want to add some capability to a list box, you would derive your new component from TListBox. However, if you want to add some new capability but exclude some existing capabilities of the standard list box, you need to derive your new list box from TCustomListBox, the ancestor of TListBox. Next, recreate or make visible the list box capabilities you want to include. Finally, add your new features.

Declaring a new component type

See also Example

When you decide that you need to derive a new type of component, you then need to decide what type to derive your new type from. As with adding new capabilities to an existing type, the essential rule to follow is this: Derive from the type that contains as much as possible that you want in your component, but which contains nothing that you don't want in your component.

Delphi provides a number of abstract component types specifically designed for component writers to use as bases for deriving new component types. The How do you create components? topic shows the different types you can start from when you create your own components.

To declare a new component type, add a type declaration to the component's unit.

Example

Here is the declaration of a simple graphical component:

```
type
  TSampleShape = class(TGraphicControl)
  end;
```

A finished component declaration will include property, field, and method declarations before the **end**, but an empty declaration is also valid, and provides a starting point for the addition of component features.

See also
Deriving new types

Ancestors and descendants

See also

From a component user's standpoint, an object is a self-contained entity consisting of properties, methods and events. Component users don't need to know or care about such issues as what object a given component descends from. But these issues are extremely important to you as a component writer.

Component users can take for granted that every component has properties named Top and Left that determine where the component appears on the form that owns it. To them, it does not matter that all components inherit those properties from a common ancestor, TComponent. However, when you create a component, you need to know which object to descend from so as to inherit the appropriate parts. You also need to know everything your component inherits, so you can take advantage of inherited features without recreating them.

From the definition of object types, you know that when you define an object type, you derive it from an existing object type. The type you derive from is called the immediate ancestor of your new object type. The immediate ancestor of the immediate ancestor is called an ancestor of the new type, as are all of its ancestors. The new type is called a descendant of its ancestors.

If you do not specify an ancestor object type, Delphi derives your object from the default ancestor object type, TObject. Ultimately, the standard type TObject is an ancestor of all objects in Object Pascal.

See also

Object hierarchies

Object hierarchies

All the ancestor-descendant relationships in an application result in a hierarchy of objects. You can see the object hierarchy in outline form by opening the Object Browser in Delphi.

The most important thing to remember about object hierarchies is that each "generation" of descendant objects contains more than its ancestors. That is, an object inherits everything that its ancestor contains, then adds new data and methods or redefines existing methods.

However, an object cannot remove anything it inherits. For example, if an object has a particular property, all descendants of that object, direct or indirect, will also have that property.

The general rule for choosing what object to derive from is simple: Pick the object that contains as much as possible of what you want to include in your new object, but which does not include anything you do not want in the new object. You can always add things to your objects, but you cannot take things out.

Controlling access

Object Pascal provides four levels of access control on the parts of objects. Access control lets you specify what code can access what parts of the object. By specifying levels of access, you define the interface to your components. If you plan the interface carefully, you will improve both the usability and reusability of your components.

Unless you specify otherwise, the fields, methods, and properties you add to your objects are published, meaning that any code that has access to the object as a whole also has access to those parts of the object, and the compiler generates run-time type information for those items.

The following table shows the levels of access, in order from most restrictive to most accessible:

Protection	Used for
private	Hiding implementation details
protected	Defining the developer's interface
public	Defining the run-time interface
published	Defining the design-time interface

All protections operate at the level of units. That is, if a part of an object is accessible (or inaccessible) in one part of a unit, it is also accessible (or inaccessible) everywhere else in the unit. If you want to give special protection to an object or part of an object, you need to put it in its own unit.

Hiding implementation details

See also Example

Declaring part of an object as **private** makes that part invisible to code outside the unit in which you declare the object type. Within the unit that contains the declaration, code can access the part as if it were **public**.

Private parts of object types are mostly useful for hiding details of implementation from users of the object. Since users of the object can't access the private parts, you can change the internal implementation of the object without affecting user code.

Note: This notion of privacy differs from some other object-oriented languages, such as C++, where only "friends" of a class can access the private parts of the class. In that sense, you can consider that all objects and other code in a unit are automatically friends of every object declared in that unit.

Example

Here is an example that shows how declaring a field as **private** prevents users from accessing information. The listing shows two form units, with each form having a handler for its OnCreate event. Each of those handlers assigns a value to a private field in one of the forms, but the compiler only allows that assignment in the form that declares the private field.

```
unit HideInfo;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs;
type
  TSecretForm = class(TForm)      { declare new form }
    procedure FormCreate(Sender: TObject);
  private  { declare private part }
    FSecretCode: Integer; { declare a private field }
  end;
var
  SecretForm: TSecretForm;
implementation
procedure TSecretForm.FormCreate(Sender: TObject);
begin
  FSecretCode := 42;      { this compiles correctly }
end;
end. { end of unit }

unit TestHide;     { this is the main form file }
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs,
  HideInfo;{ use the unit with TSecretForm }
type
  TTestForm = class(TForm)
    procedure FormCreate(Sender: TObject);
  end;
var
  TestForm: TTestForm;
implementation
procedure TTestForm.FormCreate(Sender: TObject);
begin
  SecretForm.FSecretCode := 13;   { compiler stops with "Field identifier expected" }
end;
end. { end of unit }
```

Although a program using the HideInfo unit can use objects of type TSecretForm, it cannot access the FSecretCode field in any of those objects.

See also

Defining the developer's interface

Defining the run-time interface

Defining the design-time interface

Defining the developer's interface

See also

Declaring part of an object as **protected** makes that part invisible to code outside the unit in which you declare the object type, much like parts declared **private**. The difference with protected parts, however, is that units that contain object types derived from the object type can access the protected parts.

You can use **protected** declarations to define a developer's interface to the object. That is, users of the object don't have access to the protected parts, but developers (such as component writers) do. In general, that means you can make interfaces available that allow component writers to change the way an object works without making those details visible to end users.

See also

Hiding implementation details

Defining the run-time interface

Defining the design-time interface

Defining the run-time interface

See also Example

Declaring part of an object as **public** makes that part visible to any code that has access to the object as a whole. That is, the public part has no special restrictions on it. If you don't specify any access control (**private**, **protected**, or **public**) on a field, method, or property, that part will be **published**.

Public parts of objects are available at run time to all code, so the public parts of an object define that object's run-time interface. The run-time interface is useful for items that aren't meaningful or appropriate at design time, such as properties that depend on actual run-time information or which are read-only. Methods that you intend for users of your components to call should also be declared as part of the run-time interface.

Note that read-only properties cannot operate at design time, so they should appear in the **public** declaration section.

Example

Here is an example that shows two read-only properties declared as part of a component's run-time interface:

```
type
  TSampleComponent = class(TComponent)
  private
    FTempCelsius: Integer;{ implementation details are private }
    function GetTempFahrenheit: Integer;
  public
    property TempCelsius: Integer read FTempCelsius;     { properties are public }
    property TempFahrenheit: Integer read GetTempFahrenheit;
  end;
...
function TSampleComponent.GetTempFahrenheit: Integer;
begin
  Result := FTempCelsius * 9 div 5 + 32;
end;
```

Since the user cannot change the value of the properties, they cannot appear in the Object Inspector, so they should not be part of the design-time interface.

See also

Hiding implementation details

Defining the developer's interface

Defining the design-time interface

Defining the design-time interface

See also Example

Declaring part of an object as **published** makes that part public and also generates run-time type information for the part. Among other things, run-time type information ensures that the Object Inspector can access properties and events.

Because only published parts show up in the Object Inspector, the published parts of an object define that object's design-time interface. The design-time interface should include any aspects of the object that a user might want to customize at design time, but must exclude any properties that depend on specific information about the run-time environment.

Note: Read-only properties cannot be part of the design-time interface because the user cannot alter them. Read-only properties should be **public**.

Example

Here is an example of a published property. Because it is published, it appears in the Object Inspector at design time.

```
type
  TSampleComponent = class(TComponent)
  private
    FTemperature: Integer;{ implementation details are private }
  published
    property Temperature: Integer read FTemperature write FTemperature;{ writable! }
  end;
```

Temperature, the property in this example, is available at design time, so users of the component can adjust the value.

See also

Hiding implementation details

Defining the developer's interface

Defining the run-time interface

Dispatching methods

Dispatching is the term used to describe how your application determines what code to execute when making a method call. When you write code that calls an object method, it looks like any other procedure or function call. However, objects have three different ways of dispatching methods.

The three types of method dispatch are

- Static
- Virtual
- Dynamic

Virtual and dynamic methods work the same way, but the underlying implementation is different. Both of them are quite different from static methods, however. All of the different kinds of dispatching are important to understand when you create components.

Static methods

See also Example

All object methods are static unless you specify otherwise when you declare them. Static methods work just like regular procedure or function calls. The compiler determines the exact address of the method and links the method at compile time.

The primary advantage of static methods is that dispatching them is very quick. Because the compiler can determine the exact address of the method, it links the method directly. Virtual and dynamic methods, by contrast, use indirect means to look up the address of their methods at run time, which takes somewhat longer.

The other distinction of a static method is that it does not change at all when inherited by another type. That is, if you declare an object type that includes a static method, then derive a new object type from it, the descendant object shares exactly the same method at the same address. Static methods, therefore, always do exactly the same thing, no matter what the actual type of the object.

You cannot override static methods. Declaring a method in a descendant type with the same name as a static method in the object's ancestor replaces the ancestor's method.

Example

In the following code, the first component declares two static methods. The second declares two static methods with the same names that replace the methods in the first.

```
type
  TFirstComponent = class(TComponent)
    procedure Move;
    procedure Flash;
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move;       { different from the inherited method, despite same declaration }
    function Flash(HowOften: Integer): Integer;  { this is also different }
  end;
```

See also
Virtual methods

Virtual methods

See also

Calling a virtual method is just like calling any other method, but the mechanism for dispatch is a little more complex because it is also more flexible. Virtual methods enable you to redefine methods in descendant objects, but still call the methods the same way. That is, the address of the method isn't determined at compile time. Instead, the object looks up the address of the method at run time.

To declare a new virtual method, add the directive **virtual** after the method declaration.

The **virtual** directive in a method declaration creates an entry in the object's virtual method table, or VMT. The VMT holds the addresses of all the virtual methods in an object type.

When you derive a new object from an existing object type, the new type gets its own VMT, which includes all the entries from its ancestor's VMT, plus any additional virtual methods declared in the new object. In addition, the descendant object can override any of its inherited virtual methods.

See also
Static methods
Overriding methods
Dynamic methods

Overriding methods

See also Example

Overriding a method means extending or refining it, rather than replacing it. That is, a descendant object type can redeclare and reimplement any of the methods declared in its ancestors. You can't override static methods, because declaring a static method with the same name as an inherited static method replaces the inherited method completely.

To override a method in a descendant object type, add the directive **override** to the end of the method declaration.

Using **override** will cause a compile-time error if

- The method does not exist in the ancestor object
- The ancestor's method of that name is static
- The declarations are not otherwise identical (names and types of parameters, procedure vs. function, and so on)

Example

The following code shows the declaration of two simple components. The first declares three methods, each with a different kind of dispatching. The other, derived from the first, replaces the static method and overrides the virtual methods.

```
type
  TFirstComponent = class(TCustomControl)
    procedure Move;        { static method }
    procedure Flash; virtual;     { virtual method }
    procedure Beep; dynamic;      { dynamic virtual method }
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move;        { declares new method }
    procedure Flash; override;    { overrides inherited method }
    procedure Beep; override;     { overrides inherited method }
  end;
```

See also

Virtual methods

Dynamic methods

Dynamic methods

See also Example

Dynamic methods are virtual methods with a slightly different dispatch mechanism. Because dynamic methods don't have entries in the object's virtual method table, they can reduce the amount of memory the object consumes. Dispatching dynamic methods is somewhat slower than dispatching regular virtual methods, however, so if a method call is time-critical or repeated often, you should probably make the method virtual, rather than dynamic.

To declare a dynamic virtual method, add the directive **dynamic** after the method declaration.

Instead of creating an entry in the object's virtual method table, **dynamic** assigns a number to the method, and stores the address of the associated code. Unlike the virtual method table, which contains the addresses of all of an object's virtual methods, inherited and introduced, the dynamic method list contains entries only for methods introduced or overridden by a particular object type. Inherited dynamic methods are dispatched by searching each ancestor's dynamic method list, in reverse order of inheritance.

There is a variation on dynamic methods used for handling messages, including Windows messages, in an application. The topic Handling messagesdiscusses these message-handling methods. The dispatch mechanism for message handlers is identical to that for other dynamic methods, but you declare them differently.

See also
Virtual methods
Overriding methods

Objects and pointers

One thing to be aware of when writing components that you don't need to consider when using existing components is that every Object Pascal object (and therefore every component) is really a pointer. The compiler automatically dereferences the object pointers for you, so in most cases, you never need to think about objects being pointers.

This becomes important, however, when you pass objects as parameters.

In general, you should pass objects by value rather than by reference. That is, when declaring an object as a parameter to a routine, you should not make it a **var** parameter. The reason is that objects are already pointers, which are references. Passing an object as a **var** parameter, then, would be passing a reference to the reference.

Overview of component creation

See also

This set of topics provides a broad overview of component architecture, the philosophy of component design, and the process of writing components for Delphi applications.

The main topics discussed are

- The Visual Component Library
- Components and objects
- How do you create components?
- What goes in a component?
- Creating a new component
- Testing uninstalled components

All this material assumes you have some familiarity with using Delphi and its standard components.

See also

OOP for component writers

Creating properties

Creating events

Creating methods

Using graphics in components

Handling messages

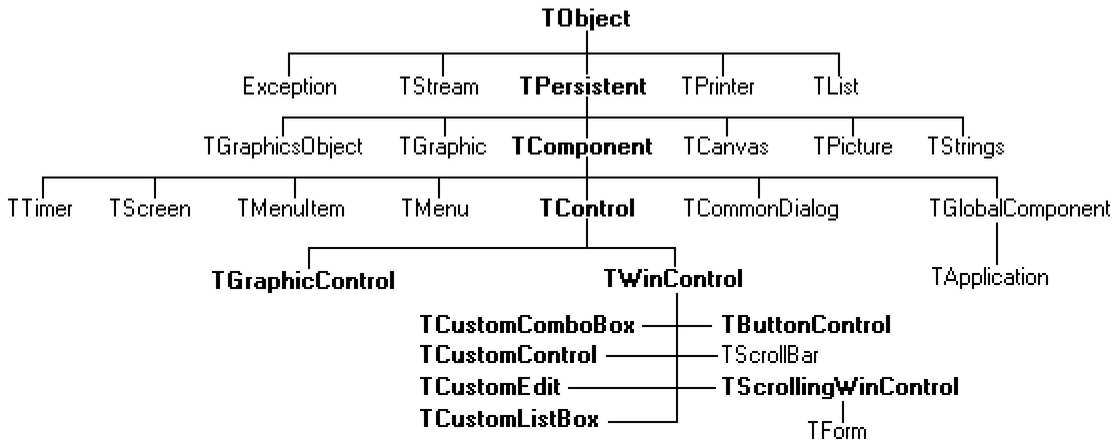
Registering components

The Visual Component Library

See also

Delphi's components are all part of an object hierarchy called the Visual Component Library (VCL). The following figure shows the relationship of the objects that make up VCL.

Note that the type TComponent is the shared ancestor of every component in the VCL. TComponent provides the minimal properties and events necessary for a component to work in Delphi. The various branches of the library provide other, more specialized capabilities.



When you create a component, you add to the VCL by deriving a new object from one of the existing object types in the hierarchy.

See also
Components and objects
How do you create components?
What goes in a component?
Creating a new component
Testing uninstalled components

Components and objects

See also

Because components are objects, component writers work with objects at a different level than component users do. Creating new components requires that you derive new types of objects. OOP for component writers describes in detail the kinds of object-oriented tasks component writers need to use.

Briefly, there are two main differences between creating components and using components. When creating components,

- You have access to parts of the object that are inaccessible to end users
- You add new parts (such as properties) to your components

Because of these differences, you need to be aware of more conventions, and you need to think in terms of how end users will use the components you write.

See also
How do you create components?
What goes in a component?
Creating a new component
Testing uninstalled components

How do you create components?

See also

A component can be almost any program element you want to manipulate at design time. Creating a new component means deriving a new component object type from an existing type. You can derive a new component from any existing component, but the following are the most common ways to create new components:

- Modifying existing controls
- Creating original controls
- Creating graphic controls
- Subclassing Windows controls
- Creating nonvisual components

The following table summarizes the different kinds of components and the object types you use as starting points for each.

To do this	Start with this type
Modify an existing component	Any existing component, such as TButton or TListBox, or an abstract component type, such as TCustomListBox.
Create an original control	TCustomControl
Create a graphic control	TGraphicControl
Use an existing Windows control	TWinControl
Create a non-visual component	TComponent

You can also derive other objects that are not components, but you cannot manipulate them in a form. Delphi includes a number of that kind of object, such as TINIFile or TFont.

See also

The Visual Component Library

Components and objects

What goes in a component?

Creating a new component

Testing uninstalled components

Modifying existing controls

See also

The simplest way to create a component is to start from an existing, working component and customize it. You can derive a new component from any of the components provided with Delphi. For instance, you might want to change the default property values of one of the standard controls.

There are certain controls, such as list boxes and grids, that have a number of variations on a basic theme. In those cases, Delphi provides an abstract control type (with the word "custom" in its name, such as TCustomGrid) from which to derive customized versions.

For example, you might want to create a special kind of list box that does not have some of the properties of the standard TListBox type. You can't remove a property from an ancestor type, so you need to derive your component from something higher in the hierarchy than TListBox. Rather than forcing you to go clear back to an abstract control type and reinvent all the list box functions, the Visual Component Library (VCL) provides TCustomListBox, which implements all the properties needed for a list box, but does not publish all of them.

When deriving a component from one of the abstract types such as TCustomListBox, you publish those properties you want to make available in your component and leave the rest protected.

See also
Creating nonvisual components
Creating original controls
Creating graphic controls
Subclassing Windows controls

Creating original controls

See also

A standard control is an item that's visible at run time, usually one the user can interact with. These standard controls all descend from the object type TCustomControl. When you create an original control (one that's not related to any existing control), you use TCustomControl as the starting point.

The key aspect of a standard control is that it has a window handle, embodied in a property called Handle. The window handle means that Windows "knows about" the control, so that, among other things,

- The control can receive the input focus
- You can pass the handle to Windows API functions (Windows needs a handle to identify which window to operate on.)

If your control doesn't need to receive input focus, you can make it a graphic control, which saves system resources.

All the components that represent standard windows controls, such as push buttons, list boxes, and edit boxes, descend from TWinControl except TLabel, since label controls never receive the input focus.

See also

Creating nonvisual components

Modifying existing controls

Creating graphic controls

Subclassing Windows controls

Creating graphic controls

See also

Graphic controls are very similar to custom controls, but they don't carry the overhead of being Windows controls. That is, Windows doesn't know about graphic controls. They have no window handles, and therefore consume no system resources. The main restriction on graphic controls is that they cannot receive the input focus.

Delphi supports the creation of custom controls through the type TGraphicControl. TGraphicControl is an abstract type derived from TControl. Although you can derive controls from TControl, you're better off deriving from TGraphicControl, which provides a canvas to paint on, and handles WM_PAINT messages, so all you need to do is override the Paint method.

See also

Creating nonvisual components

Creating original controls

Modifying existing controls

Subclassing Windows controls

Subclassing Windows controls

See also

Windows has a concept called a window class that is somewhat similar to the object-oriented programming concept of object or class. A window class is a set of information shared between different instances of the same sort of window or control in Windows.

When you create a new kind of control (usually called a custom control) in traditional Windows programming, you define a new window class and register it with Windows. You can also base a new window class on an existing class, which is called subclassing.

In traditional Windows programming, if you wanted to create a custom control, you had to write it in a dynamic-link library (DLL), much like the standard Windows controls, and provide an interface to it.

Using Delphi, you can create a component "wrapper" around any existing Windows class. So if you already have a library of custom controls that you want to use in your Delphi applications, you can create Delphi components that let you use your existing controls and derive new controls from them just as you would any other component.

See also
Modifying existing controls
Creating original controls
Creating graphic controls
Creating nonvisual components

Creating nonvisual components

See also

The abstract object type TComponent is the base type for all components. The only components you'll create directly from TComponent are nonvisual components. Most of the components you'll write will probably be various kinds of visual controls.

TComponent defines all the properties and methods essential for a component to participate in the Form Designer. Thus, any component you derive from TComponent will already have design capability built into it.

Nonvisual components are fairly rare. You'll mostly use them as an interface for nonvisual program elements (much as Delphi uses them for database elements) and as placeholders for dialog boxes (such as the file dialog boxes).

See also

Creating original controls

Modifying existing controls

Creating graphic controls

Subclassing Windows controls

What goes in a component?

See also

There are few restrictions on what you can put in the components you write. However, there are certain conventions you should follow if you want to make your components easy and reliable for the people who will use them.

This section discusses the philosophies underlying the design of components, including the following topics:

- Removing dependencies
- Properties and events and methods
- Graphics encapsulation
- Registration

See also
The Visual Component Library
Components and objects
How do you create components?
Creating a new component
Testing uninstalled components

Removing dependencies

See also Example

Perhaps the most important philosophy behind the creation of Delphi's components is the necessity of removing dependencies. One of the things that makes components so easy for end users to incorporate into their applications is the fact that there are generally no restrictions on what they can do at any given point in their code.

The very nature of components suggests that different users will incorporate them into applications in varying combinations, orders, and environments. You should design your components so that they function in any context, without requiring any preconditions.

See also

Properties and events and methods

Graphics encapsulation

Registration

An example of removing dependencies

An excellent example of removing dependencies in components is the Handle property of windowed controls. If you've written Windows applications before, you know that one of the most difficult and error-prone aspects of getting a program running is making sure that you don't access a window or control until you've created it by calling the CreateWindow API function. Calling API functions with invalid handles causes a multitude of problems.

Delphi components protect users from worrying about window handles and whether they are valid by ensuring that a valid handle is always available when needed. That is, by using a property for the window handle, the component can check whether the window has been created, and therefore whether there is a valid window handle. If the handle isn't already valid, the property creates the window and returns the handle. Thus, any time a user's code accesses the Handle property, it is assured of getting a valid handle.

By removing the background tasks such as creating the window, components allow developers to focus on what they really want to do. If a developer needs to pass a window handle to an API function, it shouldn't be necessary to first check to make sure there's a valid handle and, if necessary, create the window. With component-based programming, the programmer can write assuming that things will work, instead of constantly checking for things that might go wrong.

Although it might take a little more time to create components that don't have dependencies, it's generally time well spent. Not only does it keep users of your components from having to repeatedly perform the same tasks, but it also reduces your documentation and support burdens, since you don't have to provide and explain numerous warnings or resolve the problems users might have with your components.

Properties and events and methods

See also

Outside of the visible image the component user manipulates in the form at design time, the most obvious attributes of a component are its properties, events, and methods. Each of these is sufficiently important that it has its own section in this file, but this section will explain a little of the philosophy of implementing them.

Properties

Properties give the component user the illusion of setting or reading the value of a variable in the component while allowing the component writer to hide the underlying data structure or to implement side effects of accessing the value.

There are several advantages to the component writer in using properties:

- Properties are available at design time.

  This allows the component user to set and change initial values of properties without having to write code.

- Properties can check values or formats as the user assigns them.

  Validating user input prevents errors caused by invalid values.

- The component can construct appropriate values on demand.

  Perhaps the most common type of error programmers make is to reference a variable that hasn't had an initial value assigned. By making the value a property, you can ensure that the value read from the property is always valid.

Creating properties explains how to add properties to your components.

Events

Events are connections between occurrences determined by the component writer (such as mouse actions and keystrokes) and code written by component users ("event handlers"). In essence, an event is the component writer's way of providing a hook for the component user to specify what code to execute when a particular occurrence happens.

It is events, therefore, that allow component users to be component users instead of component writers. The most common reason for subclassing in traditional Windows applications is that users want to specify a different response to, for example, a Windows message. But in Delphi, component users can specify handlers for predefined events without subclassing, so they don't need to derive their own components.

Creating events explains how to add events for standard Windows occurrences or events you define yourself.

Methods

Methods are procedures or functions built into a component. Component users use methods to direct a component to perform a specific action or return a certain value not covered by a property. Methods are also useful for updating several related properties with a single call.

Because they require execution of code, methods are only available at run time.

Creating methods explains how to add methods to your components.

See also

Removing dependencies

Graphics encapsulation

Registration

Graphics encapsulation

See also

Delphi takes most of the drudgery out of Windows graphics by encapsulating the various graphic tools into a canvas. The canvas represents the drawing surface of a window or control, and contains other objects, such as a pen, a brush, and a font. A canvas is much like a Windows device context, but it takes care of all the bookkeeping for you.

If you've ever written a graphic Windows application, you're familiar with the kinds of requirements Windows' graphics device interface (GDI) imposes on you, such as limits on the number of device contexts available, and restoring graphic objects to their initial state before destroying them.

When working with graphics in Delphi, you need not concern yourself with any of those things. To draw on a form or component, you access the Canvas property. If you want to customize a pen or brush, you set the color or style. When you finish, Delphi takes care of disposing of the resources. In fact, it caches resources, so if your application frequently uses the same kinds of resources, it will probably save a lot of creating and recreating.

Of course, you still have full access to the Windows GDI, but you'll often find that your code is much simpler and runs faster if you use the canvas built into Delphi components. Delphi's graphics features are detailed in Using graphics in components.

See also

Removing dependencies

Properties and events and methods

Registration

Registration

See also

Before your components can operate in Delphi at design time, you have to register them with Delphi. Registration tells Delphi where you want your component to appear on the Component palette. There are also some customizations you can make to the way Delphi stores your components in the form file. Registration is explained in Registering components.

See also
Removing dependencies
Properties and events and methods
Graphics encapsulation

Creating a new component

See also

There are several steps you perform whenever you create a new component. All the examples given that create new components assume you know how to perform these steps.

You can create a new component two ways:

- Creating a component manually
- Using the Component Expert

Once you do either of those, you have at least a minimally functional component ready to install on the Component palette. After installing, you can add your new component to a form and test it in both design time and run time. You can then add more features to the component, update the palette, and continue testing.

See also

The Visual Component Library

Components and objects

How do you create components?

What goes in a component?

Testing uninstalled components

Creating a component manually

See also

The easiest way to create a new component is to use the Component Expert. However, you can also perform the same steps manually.

Creating a component manually requires three steps:

1. Creating a new unit
2. Deriving the component object
3. Registering the component

See also
Using the Component Expert

Creating a new unit

See also

A unit is a separately compiled module of Object Pascal code. Delphi uses units for a number of purposes. Every form has its own unit, and most components (or logical groups of components) have their own units as well.

When you create a component, you either create a new unit for the component, or add the new component to an existing unit.

To create a unit for a component, choose File|New Unit.

Delphi creates a new file and opens it in the Code Editor.

To add a component to an existing unit, choose File|Open to choose the source code for an existing unit.

Note: When adding a component to an existing unit, make sure that unit already contains only component code. Adding component code to a unit that contains, for example, a form, will cause errors in the Component palette.

Once you have either a new or existing unit for your component, you can derive the component object.

See also

Deriving the component object

Registering the component

Deriving the component object

See also Example

Every component is an object descended from the type TComponent, from one of its more specialized descendants, such as TControl or TGraphicControl, or from an existing component type. How do you create components? describes which types to descend from for different kinds of components.

Deriving new objects is explained in more detail in Creating new objects.

To derive a component object, add an object type declaration to the **interface** part of the unit that will contain the component.

Example

To create the simplest component type, a non-visual component descended directly from TComponent, add the following type definition to the **interface** part of your component unit:

```
type
  TNewComponent = class(TComponent)
  end;
```

You can now register TNewComponent. Note that the new component does nothing different from TComponent yet. However, you've created a framework on which you'll build your new component.

See also

Creating a new unit

Registering the component

Registering the component

See also Example

Registering a component is a simple process that tells Delphi what components to add to its component library, and which pages of the Component palette the components should appear on. Registering components describes the registration process and its nuances in much more detail.

To register a component,

1 Add a procedure named Register to the **interface** part of the component's unit.

  Register takes no parameters, so the declaration is very simple:

```
procedure Register;
```

  If you're adding a component to a unit that already contains components, it should already have a Register procedure declared, so you don't need to change the declaration.

2 Write the Register procedure in the **implementation** part of the unit, calling RegisterComponents for each component you want to register.

  RegisterComponents is a procedure that takes two parameters: the name of a Component palette page and a set of component types. If you're adding a component to an existing registration, you can either add the new component to the set in the existing statement, or add a new statement that calls RegisterComponents.

Example

To register a component named TNewComponent and place it on the Samples page of the Component palette, add the following Register procedure to the unit that contains TNewComponent's declaration:

```
procedure Register;
begin
  RegisterComponents('Samples', [TNewComponent]);
end;
```

Once you register a component, you can install the component onto the Component palette.

See also
Declaring the Register procedure
Implementing the Register procedure
Creating a new unit
Deriving the component object

Using the Component Expert

See also

You can use the Component Expert to create a new component. Using the Component Expert simplifies the initial stages of creating a new component, as you need only specify three things:

- The name of the new component
- The ancestor type
- The Component palette page you want it to appear on

The Component Expert performs the same tasks you would do when creating a component manually, namely

- Creating a new unit
- Deriving the component object
- Registering the component

The Component Expert cannot add components to an existing unit. You must add components to existing units manually.

To open the Component Expert, choose File|New Component.

After you fill in the fields in the Component Expert, choose OK. Delphi creates a new unit containing the type declaration and the Register procedure, and adds a **uses** clause that includes all the standard Delphi units.

You should save the unit right away, giving the unit a meaningful name.

See also
Creating a component manually

Testing uninstalled components

See also Example

You can test the run-time behavior of a component before you install it on the Component palette. This is particularly useful for debugging newly-created components, but you can use the same technique for testing any component, regardless of whether the component appears on the Component palette.

In essence, you can test an uninstalled component by emulating the actions performed by Delphi when a user places a component from the Component palette on a form.

To test an uninstalled component, do the following:

1 Add the name of component's unit to the form unit's **uses** clause.

2 Add an object field to the form to represent the component.

This is one of the main differences between the way you add components and the way Delphi does it. You add the object field to the public part at the bottom of the form's type declaration. Delphi would add it above, in the part of the type declaration that it manages.

You should never add fields to the Delphi-managed part of the form's type declaration. The items in that part of the type declaration correspond to the items stored in the form file. Adding the names of components that do not exist on the form can render your form file invalid.

3 Attach a handler to the form's OnCreate event.

4 Construct the component in the form's OnCreate handler.

When you call the component's constructor, you must pass a parameter specifying the owner of the component (the component responsible for destroying the component when the time comes). You will nearly always pass Self as the owner. In a method, Self is a reference to the object that contains the method. In this case, in the form's OnCreate handler, Self refers to the form.

5 Assign the Parent property.

Setting the Parent property is always the first thing to do after constructing a control. The parent is the component that visually contains the control, which is most often the form, but might be a group box or panel. Normally, you'll set Parent to Self, that is, the form. Always set Parent before setting other properties of the control.

Warning: If your component is not a control (that is, if TControl is not one of its ancestors), skip this step. If you accidentally set the form's Parent property to Self instead of the component's, you can cause Windows to crash.

6 Set any other component properties as desired.

Example

Suppose you want to test a new component of type TNewComponent in a unit named NewTest. Create a new project, then follow the steps to end up with a form unit that looks like this:

```
unit Unit1;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, NewTest;{ 1. Add NewTest to uses clause }

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);       { 3. Attach a handler to OnCreate }
  private
    { Private declarations }
  public
    { Public Declarations }
    NewComponent1: TNewComponent; { 2. Add an object field }
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  NewComponent1 := TNewComponent.Create(Self);   { 4. Construct the component }
  NewComponent1.Parent := Self;   { 5. Set Parent property if component is a control }
  NewComponent.Left := 12;{ 6. Set other properties... )
  ...{ ...continue as needed }
end;
end.
```

See also

The Visual Component Library

Components and objects

How do you create components?

What goes in a component?

Creating a new component

Creating properties

See also

Properties are the most distinctive parts of components, largely because component users can see and manipulate them at design time and get immediate feedback as the components react in real time. Properties are also important because, if you design them well, they make your components easier for others to use and easier for you to maintain.

In order to best make use of properties in your components, you should understand the following:

- Why create properties?
- Types of properties
- Publishing inherited properties
- Defining component properties
- Creating array properties
- Writing property editors

See also

Overview of component creation
OOP for component writers
Creating events
Creating methods
Using graphics in components
Handling messages
Registering components

Why create properties?

See also

Properties provide significant advantages, both for you as a component writer and for the users of your components. The most obvious advantage is that properties can appear in the Object Inspector at design time. That simplifies your programming job, because instead of handling several parameters to construct an object, you just read the values assigned by the user.

From the component user's standpoint, properties look like variables. Users can set or read the values of properties much as if those properties were object fields. About the only thing they cannot do with a property that they would with a variable is pass it as a **var** parameter.

From the component writer's standpoint, however, properties provide much more power than simple object fields because

- Users can set properties at design time.

  This is very important, because unlike methods, which are only available at run time, properties let users customize components before running an application. In general, your components should not contain a lot of methods; most of them can probably be encapsulated into properties.

- Unlike an object field, a property can hide implementation details from users.

  For example, the data might be stored internally in an encrypted form, but when setting or reading the value of the property, the data would appear unencrypted. Although the value of a property might be a simple number, the component might look up the value from a database or perform complex calculations to arrive at that value.

- Properties allow side effects to outwardly simple assignments.

  What appears to be a simple assignment involving an object field is really a call to a method, and that method could do almost anything.

  A simple but clear example is the Top property of all components. Assigning a new value to Top doesn't just change some stored value; it causes the component to relocate and repaint itself. The effects of property setting need not be limited to an individual component. Setting the Down property of a speed-button component to True causes the speed button to set the Down properties of all other speed buttons in its group to False.

- The implementation methods for a property can be virtual, meaning that what looks like a single property to a component user might do different things in different components.

See also

Types of properties

Publishing inherited properties

Defining component properties

Creating array properties

Writing property editors

Types of properties

See also

A property can be of any type that a function can return (since the implementation of the property can use a function). All the standard rules of Pascal type compatibility apply to properties, just as they would to variables. Type compatibility is explained in Chapter 4 of the Delphi User's Guide.

The most important aspect of choosing types for your properties is that different types appear differently in the Object Inspector. The Object Inspector uses the type of the property to determine what choices appear to the user. You can specify a different property editor when you register your components, as explained in Writing property editors.

Property type	**Object Inspector treatment**
Simple	Numeric, character, and string properties appear in the Object Inspector as numbers, characters, and strings, respectively. The user can type and edit the value of the property directly.
Enumerated	Properties of enumerated types (including Boolean) display the value as defined in the source code. The user can cycle through the possible values by double-clicking the value column. There is also a drop-down list that shows all possible values of the enumerated type.
Set	Properties of set types appear in the Object Inspector looking like a set. By expanding the set, the user can treat each element of the set as a Boolean value: True if the element is included in the set or False if it's not included.
Object	Properties that are themselves objects often have their own property editors. However, if the object that is a property also has published properties, the Object Inspector allows the user to expand the list of object properties and edit them individually. Object properties must descend from TPersistent.
Array	Array properties must have their own property editors. The Object Inspector has no built-in support for editing array properties.

See also

Why create properties?

Publishing inherited properties

Defining component properties

Creating array properties

Writing property editors

Publishing inherited properties

See also Example

All components inherit properties from their ancestor types. When you derive a new component from an existing component type, your new component inherits all the properties in the ancestor type. If you derive instead from one of the abstract types, many of the inherited properties are either **protected** or **public**, but not **published**.

If you need more information about levels of protection such as **protected**, **private**, and **published**, see Controlling access.

To make a **protected** or **public** property available to users of your components, you must redeclare the property as **published**.

Redeclaring means adding the declaration of an inherited property to the declaration of a descendant object type.

Example

If you derive a component from TWinControl, it inherits a Ctl3D property, but that property is **protected**, so users of the component cannot access Ctl3D at design time or run time. By redeclaring Ctl3D in your new component, you can change the level of protection to either **public** or **published**.

The following code shows a redeclaration of Ctl3D as **published**, making it available at design time:

```
type
  TSampleComponent = class(TWinControl)
  published
    property Ctl3D;
  end;
```

Note that redeclarations can only make a property less restricted, not more restricted. Thus, you can make a **protected** property **public**, but you cannot "hide" a **public** property by redeclaring it as **protected**.

When you redeclare a property, you specify only the property name, not the type and other information described in Defining component properties. You can also declare new default values when redeclaring a property, or specify whether to store the property.

See also
Why create properties?
Types of properties
Defining component properties
Creating array properties
Writing property editors

Defining component properties

See also

The syntax for property declarations is explained in detail in the topic for the reserved word **property**. This section focuses on the particulars of declaring properties in Delphi components and the conventions used by the standard components.

Specific topics include

- The property declaration
- Internal data storage
- Direct access
- Access methods
- Default property values

See also
Why create properties?
Types of properties
Publishing inherited properties
Creating array properties
Writing property editors

The property declaration

See also Example

Declaring a property and its implementation is straightforward. You add the property declaration to the declaration of your component object type.

To declare a property, you specify three things:

- The name of the property
- The type of the property
- Methods to read and/or set the value of the property

At a minimum, a component's properties should be declared in a **public** part of the component's object-type declaration, making it easy to set and read the properties from outside the component at run time.

To make the property editable at design time, declare the property in a **published** part of the component's object type declaration. Published properties automatically appear in the Object Inspector. Public properties that aren't published are available only at run time.

Example

Here is a typical property declaration:

```
type
  TYourComponent = class(TComponent)
  ...
  private
    FCount: Integer;       { field for internal storage }
    function GetCount: Integer;   { read method }
    procedure SetCount(ACount: Integer); { write method }
  public
    property Count: Integer  read GetCount write SetCount;      { property declaration }
  end;
```

See also

Internal data storage

Direct access

Access methods

Default property values

Internal data storage

See also

There are no restrictions on how you store the data for a property. In general, however, Delphi's components follow these conventions:

- Property data is stored in object fields.
- Identifiers for properties' object fields start with the letter F, and incorporate the name of the property. For example, the raw data for the Width property defined in TControl is stored in an object field called FWidth.
- Object fields for property data should be declared as **private**. This ensures that the component that declares the property has access to them, but component users and descendant components don't. Descendant components should use the inherited property itself, not direct access to the internal data storage, to manipulate a property.

The underlying principle behind these conventions is that only the implementation methods for a property should access the data behind that property. If a method or another property needs to change that data, it should do so through the property, not by direct access to the stored data. This ensures that the implementation of an inherited property can change without invalidating descendant components.

See also
The property declaration
Direct access
Access methods
Default property values

Direct access

See also Example

The simplest way to make property data available is direct access. That is, the **read** and **write** parts of the property declaration specify that assigning or reading the property value goes directly to the internal storage field without calling an access method. Direct access is useful when the property has no side effects, but you want to make it available in the Object Inspector.

It is common to have direct access for the **read** part of a property declaration but use an access method for the **write** part, usually to update the status of the component based on the new property value.

Example

The following component-type declaration shows a property that uses direct access for both the **read** and **write** parts:

```
type
  TSampleComponent = class(TComponent)
  private  { internal storage is private }
    FReadOnly: Boolean;   { declare field to hold property value }
  published{ make property available at design time }
    property ReadOnly: Boolean read FReadOnly write FReadOnly;
  end;
```

See also

The property declaration

Internal data storage

Access methods

Access methods

See also

The syntax for property declarations allows the **read** and **write** parts of a property declaration to specify access methods instead of an object field. Regardless of how a particular property implements its **read** and **write** parts, however, that implementation should be **private**, and descendant components should use the inherited property for access. This ensures that use of a property will not be affected by changes in the underlying implementation.

Making access methods private also ensures that component users don't accidentally call those methods, inadvertently modifying a property.

The read method

The **read** method for a property is a function that takes no parameters, and returns a value of the same type as the property. By convention, the function's name is "Get" followed by the name of the property. For example, the **read** method for a property named Count would be named GetCount.

The only exception to the "no parameters" rule is for array properties, which pass their indexes as parameters.

The **read** method manipulates the internal storage data as needed to produce the value of the property in the appropriate type.

If you don't declare a **read** method, the property is write-only. Write-only properties are very rare, and generally not very useful.

The write method

The **write** method for a property is always a procedure that takes a single parameter, of the same type as the property. The parameter can be passed by reference or by value, and can have any name you choose. By convention, the procedure's name is "Set" followed by the name of the property. For example, the **write** method for a property named Count would be named SetCount.

The value passed in the parameter is used to set the new value of the property, so the **write** method needs to perform any manipulation needed to put the appropriate values in the internal storage.

If you don't declare a **write** method, the property is read-only.

It's common to test whether the new value actually differs from the current value before setting the value. For example, here's a simple **write** method for an integer property called Count that stores its current value in a field called FCount:

```
procedure TMyComponent.SetCount(Value: Integer);
begin
  if Value <> FCount then
  begin
    FCount := Value;
    Update;
  end;
end;
```

See also

The property declaration

Internal data storage

Direct access

Default property values

Default property values

See also Example

When you declare a property, you can optionally declare a default value for the property. The default value for a component's property is the value set for that property in the component's constructor. For example, when you place a component from the Component palette on a form, Delphi creates the component by calling the component's constructor, which determines the initial values of the component's properties.

Delphi uses the declared default value to determine whether to store a property in a form file. For more information on storing properties and the importance of default values, see Storing and loading properties. If you do not specify a default value for a property, Delphi always stores the property.

To declare a default value for a property, append the **default** directive to the property's declaration (or redeclaration), followed by the default value.

Note: Declaring a default value in the property declaration does not actually set the property to that value. It is your responsibility as the component writer to ensure that the component's constructor actually sets the property to that value.

Specifying no default value

When redeclaring a property, you can specify that the property has no default value, even if the inherited property specified one.

To designate a property as having no default value, append the **nodefault** directive to the property's declaration.

When you declare a property for the first time, there is no need to specify **nodefault**, because the absence of a declared default value means the same thing.

Example

Here is the declaration of a component that includes a single Boolean property named IsTrue with a default value of True, including the constructor that sets the default value.

```
type
  TSampleComponent = class(TComponent)
  private
    FIsTrue: Boolean;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property IsTrue: Boolean read FIsTrue write FIsTrue default True;
  end;
...
constructor TSampleComponent.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);       { always call the inherited constructor }
  FIsTrue := True; { set the default value }
end;
```

Note that if the default value for IsTrue had been False, you would not need to set it explicitly in the constructor, since all objects (and therefore, components) always initialize all their fields to zero, and a "zeroed" Boolean value is False.

See also
The property declaration
Internal data storage
Direct access
Access methods

Creating array properties

See also Example

Some properties lend themselves to being indexed, much like arrays. That is, they have multiple values that correspond to some kind of index value. An example in the standard components is the Lines property of the memo component. Lines is an indexed list of the strings that make up the text of the memo, which you can treat as an array of strings. In this case, the array property gives the user natural access to a particular element (a string) in a larger set of data (the memo text).

Array properties work just like other properties, and you declare them in largely the same way. The only differences in declaring array properties are as follows:

- The declaration for the property includes one or more indexes with specified types. Indexes can be of any type.
- The **read** and **write** parts of the property declaration, if specified, must be methods. They cannot be object fields.
- The access methods for reading and writing the property values take additional parameters that correspond to the index or indexes. The parameters must be in the same order and of the same type as the indexes specified in the property declaration.

Although they seem quite similar, there are a few important distinctions between array properties and arrays. Unlike the index of an array, the index type for an array property does not have to be an integer type. You can index a property on a string, for example. In addition, you can only reference individual elements of an array property, not the entire range of the property.

Example

Here's the declaration of a property that returns a string based on an integer index:

```
type
  TDemoComponent = class(TComponent)
  private
    function GetNumberName(Index: Integer): string;
  public
    property NumberName[Index: Integer]: string read GetNumberName;
  end;
...
function TDemoComponent.GetNumberName(Index: Integer): string;
begin
  Result := 'Unknown';
  case Index of
    -MaxInt..-1: Result := 'Negative';
    0: Result := 'Zero';
    1..100: Result := 'Small';
    101..MaxInt: Result := 'Large';
  end;
end;
```

See also
Why create properties?
Types of properties
Publishing inherited properties
Defining component properties
Writing property editors

Writing property editors

See also

The Object Inspector provides default editing for all types of properties. You can, however, provide an alternate editor for specific properties by writing and registering property editors. You can register property editors that apply only to the properties in the components you write, but you can also create editors that apply to all properties of a certain type.

At the simplest level, a property editor can operate in either or both of two ways: displaying and allowing the user to edit the current value as a text string, and displaying a dialog box that permits some other kind of editing. Depending on the property being edited, you might find it useful to provide either or both kinds.

Writing a property editor requires five steps:

- Deriving a property-editor object
- Editing the property as text
- Editing the property as a whole
- Specifying editor attributes
- Registering the property editor

See also
Why create properties?
Types of properties
Publishing inherited properties
Defining component properties
Creating array properties

Deriving a property-editor object

See also Example

The DsgnIntf unit defines several kinds of property editors, all of which descend from TPropertyEditor. When you create a property editor, your property-editor object can either descend directly from TPropertyEditor or indirectly through one of the property-editor types described in the table below.

To create a property-editor object, derive a new object type from one of the existing property editor types.

The DsgnIntf unit also defines some very specialized property editors used by unique properties such as the component name. The listed property editors are the ones more useful for user-defined properties.

Type	Properties edited
TOrdinalProperty	All ordinal-property editors (those for integer, character, and enumerated properties) descend from TOrdinalProperty.
TIntegerProperty	All integer types, including predefined and user-defined subranges.
TCharProperty	Char-type and subranges of Char, such as 'A'..'Z'.
TEnumProperty	Any enumerated type.
TFloatProperty	All floating-point numbers.
TStringProperty	Strings, including strings of specified length, such as **string**[20]
TSetElementProperty	Individual elements in sets, shown as Boolean values
TSetProperty	All sets. Sets are not directly editable, but can expand into a list of set-element properties.
TClassProperty	Objects. Displays the name of the object type and allows expansion of the object's properties.
TMethodProperty	Method pointers, most notably events.
TComponentProperty	Components in the same form. The user cannot edit the component's properties, but can point to a specific component of a compatible type.
TColorProperty	Component colors. Shows color constants if applicable, otherwise displays hexadecimal value. Drop-down list contains the color constants. Double-click opens the color-selection dialog box.
TFontNameProperty	Font names. The drop-down list displays all currently installed fonts.
TFontProperty	Fonts. Allows expansion of individual font properties as well as access to the font dialog box.

Example

One of the simplest property editors is TFloatPropertyEditor, the editor for properties that are floating-point numbers. Here is its declaration:

```
type
  TFloatProperty = class(TPropertyEditor)
  public
    function AllEqual: Boolean; override;
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
  end;
```

See also

Editing the property as text

Editing the property as a whole

Specifying editor attributes

Registering the property editor

Editing the property as text

See also Examples

All properties need to provide a string representation of their values for the Object Inspector to display. Most properties also allow the user to type in a new value for the property. Property-editor objects provide virtual methods you can override to convert between the text representation and the actual value.

The methods you override are called GetValue and SetValue. Your property editor also inherits a set of methods used for assigning and reading different sorts of values, as shown in the following table

Property type	"Get" method	"Set" method
Floating point	GetFloatValue	SetFloatValue
Method pointer (event)	GetMethodValue	SetMethodValue
Ordinal type	GetOrdValue	SetOrdValue
String	GetStrValue	SetStrValue.

When you override a GetValue method, you will call one of the "Get" methods, and when you override SetValue, you will call one of the "Set" methods.

Displaying the property value

The property editor's GetValue method returns a string that represents the current value of the property. The Object Inspector uses this string in the value column for the property. By default, GetValue returns 'unknown'.

To provide a string representation of your property, override the property editor's GetValue method.

If the property isn't a string value, your GetValue must convert the value into a string representation.

Setting the property value

The property editor's SetValue method takes a string typed by the user in the Object Inspector, converts it into the appropriate type, and sets the value of the property. If the string does not represent a proper value for the property, SetValue should raise an exception and not use the improper value.

To read string values into properties, override the property editor's SetValue method.

Examples

Here are the GetValue and SetValue methods for TIntegerProperty. Integer is an ordinal type, so GetValue calls GetOrdValue and converts the result to a string. SetValue converts the string to an integer, performs some range checking, and calls SetOrdValue.

```
function TIntegerProperty.GetValue: string;
begin
  Result := IntToStr(GetOrdValue);
end;

procedure TIntegerProperty.SetValue(const Value: string);
var
  L: Longint;
begin
  L := StrToInt(Value);   { convert string to number }
  with GetTypeData(GetPropType)^ do      { this uses compiler data for type Integer }
    if (L < MinValue) or (L > MaxValue) then     { make sure it's in range... }
      raise EPropertyError.Create(       { ...otherwise, raise exception }
        FmtLoadStr(SOutOfRange, [MinValue, MaxValue]));
  SetOrdValue(L);  { if in range, go ahead and set value }
end;
```

The specifics of the particular examples here are less important than the principle: GetValue converts the value to a string; SetValue converts the string and validates the value before calling one of the "Set" methods.

See also

Deriving a property-editor object

Editing the property as a whole

Specifying editor attributes

Registering the property editor

Editing the property as a whole

See also   Example

You can optionally provide a dialog box in which the user can visually edit a property. The most common use of property editors is for properties that are themselves objects. An example is the Font property, for which the user can open a font dialog box to choose all the attributes of the font at once.

To provide a whole-property editor dialog box, override the property-editor object's Edit method.

Note that Edit methods use the same "Get" and "Set" methods used in writing GetValue and SetValue methods. In fact, an Edit method calls both a "Get" method and a "Set" method. Since the editor is type-specific, there is usually no need to convert the property values to strings. The editor generally deals with the value "as retrieved."

When the user clicks the '...' button next to the property or double-clicks the value column, the Object Inspector calls the property editor's Edit method.

Example

The Color properties found in most components use the standard Windows color dialog box as a property editor. Here is the Edit method from TColorProperty, which invokes the dialog box and uses the result:

```
procedure TColorProperty.Edit;
var
  ColorDialog: TColorDialog;
begin
  ColorDialog := TColorDialog.Create(Application);      { construct the editor }
  try
    ColorDialog.Color := GetOrdValue;    { use the existing value }
    if ColorDialog.Execute then   { if the user OKs the dialog... }
      SetOrdValue(ColorDialog.Color);    { ...use the result to set value }
  finally
    ColorDialog.Free;     { destroy the editor }
  end;
end;
```

See also
Deriving a property-editor object
Editing the property as text
Specifying editor attributes
Registering the property editor

Specifying editor attributes

See also Example

The property editor must provide information that the Object Inspector can use to determine what tools to display. For example, the Object Inspector needs to know whether the property has subproperties or can display a list of possible values.

To specify editor attributes, override the property editor's GetAttributes method.

GetAttributes is a function that returns a set of values of type TPropertyAttributes that can include any or all of the following values:

Flag	Meaning if included	Related method
paValueList	The editor can give a list of enumerated values.	GetValues
paSubProperties	The property has subproperties that can display.	GetProperties
paDialog	The editor can display a dialog box for editing the entire property.	Edit
paMultiSelect	The property should display when the user selects more than one component.	N/A
paAutoUpdate	Updates the component after every change instead of waiting for approval of the value.	SetValue
paSortList	The Object Inspector should sort the value list.	N/A
paReadOnly	Users cannot modify the property value.	N/A

Example

Color properties are more versatile than most, in that they allow several ways for users to choose them in the Object Inspector: typing, selection from a list, and customized editor. TColorProperty's GetAttributes method, therefore, includes several attributes in its return value:

```
function TColorProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paMultiSelect, paDialog, paValueList];
end;
```

See also

Deriving a property-editor object

Editing the property as text

Editing the property as a whole

Registering the property editor

Registering the property editor

See also Example

Once you create a property editor, you need to register it with Delphi. Registering a property editor associates a type of property with a specific property editor. You can register the editor with all properties of a given type or just with a particular property of a particular type of component.

To register a property editor, call the RegisterPropertyEditor procedure.

RegisterPropertyEditor takes four parameters:

- A type-information pointer for the type of property to edit.

  This is always a call to the built-in function TypeInfo, such as TypeInfo(TMyComponent).
- The type of the component to which this editor applies. If this parameter is **nil**, the editor applies to all properties of the given type.
- The name of the property. This parameter only has meaning if the previous parameter specifies a particular type of component. In that case, you can specify the name of a particular property in that component type to which this editor applies.
- The type of property editor to use for editing the specified property.

Example

Here is an excerpt from the procedure that registers the editors for the standard components on the Component palette:

```
procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(TComponent), nil, '', TComponentProperty);
  RegisterPropertyEditor(TypeInfo(TComponentName), TComponent, 'Name',
    TComponentNameProperty);
  RegisterPropertyEditor(TypeInfo(TMenuItem), TMenu, '', TMenuItemProperty);
end;
```

The three statements in this procedure cover the different uses of RegisterPropertyEditor:

- The first statement is the most typical. It registers the property editor TComponentProperty for all properties of type TComponent (or descendants of TComponent that do not have their own editors registered). In general, when you register a property editor, you've created an editor for a particular type, and you want to use it for all properties of that type, so the second and third parameters are **nil** and an empty string, respectively.
- The second statement is the most specific kind of registration. It registers an editor for a specific property in a specific type of component. In this case, the editor is for the Name property of all components.
- The third statement is more specific than the first, but not as limited as the second. It registers an editor for all properties of type TMenuItem in components of type TMenu.

See also

Deriving a property-editor object

Editing the property as text

Editing the property as a whole

Specifying editor attributes

Creating events

See also

Events are very important parts of components, although the component writer doesn't usually need to do much with them. An event is a link between an occurrence in the system (such as a user action or a change in focus) that a component might need to respond to and a piece of code that responds to that occurrence. The responding code is an event handler, and is nearly always written by the component user.

By using events, application developers can customize the behavior of components without having to change the objects themselves. As a component writer, you use events to enable application developers to customize the behavior of your components.

Events for the most common user actions (such as mouse actions) are built into all the standard Delphi components, but you can also define new events and give them their own events. In order to create events in a component, you need to understand the following:

- What are events?
- Implementing the standard events
- Defining your own events

Delphi implements events as properties, so you should already be familiar with Creating properties before you attempt to create or change a component's events.

See also

Overview of component creation

OOP for component writers

Creating properties

Creating methods

Using graphics in components

Handling messages

Registering components

What are events?

See also

Loosely defined, an event is a mechanism that links an occurrence to some code. More specifically, an event is a method pointer, a pointer to a specific method in a specific object instance.

From the component user's perspective, an event is just a name related to a system event, such as OnClick, that the user can assign a specific method to call. For example, a push button called Button1 has an OnClick method. By default, Delphi generates an event handler called Button1Click in the form that contains the button and assigns it to OnClick. When a click event occurs on the button, the button calls the method assigned to OnClick, in this case, Button1Click.

Thus, the component user sees the event as a way of specifying what user-written code the application should call when a specific event occurs.

From the component writer's perspective, there's a lot more to an event. The most important thing to remember, however, is that you're providing a link, a place where the component's user can attach code in response to certain kinds of occurrences. Your components provide outlets where the user can "plug in" specific code.

In order to write an event, you need to understand the following:

- Events are method pointers
- Events are properties
- Event-handler types
- Event handlers are optional

See also

Implementing the standard events

Defining your own events

Events are method pointers

See also Example

Delphi uses method pointers to implement events. A method pointer is a special pointer type that points to a specific method in a specific object instance. As a component writer, you can treat the method pointer as a placeholder: your code detects that an event occurs, so you call the method (if any) specified by the user for that event.

Method pointers work just like any other procedural type, but they maintain a hidden pointer to an object instance. When the user assigns a handler to a component's event, the assignment is not just to a method with a particular name, but rather to a specific method of a specific object instance. That instance is usually the form that contains the component, but it need not be.

Example

All controls inherit a dynamic method called Click for handling click events. The implementation of Click calls the user's click-event handler, if any:

```
procedure TControl.Click;
begin
  if Assigned(OnClick) then OnClick(Self);
end;
```

If the user has assigned a handler to a control's OnClick event, clicking the control results in that method being called. If no handler is assigned, nothing happens.

See also

Events are properties

Event-handler types

Event handlers are optional

Events are properties

See also

Components use properties to implement their events. Unlike most other properties, events don't use methods to implement their **read** and **write** parts. Instead, event properties use a private object field of the same type as the property.

By convention, the field's name is the same as the name of the property, but preceded by the letter F. For example, the OnClick method's pointer is stored in a field called FOnClick of type TNotifyEvent, and the declaration of the OnClick event property looks like this:

```
type
  TControl = class(TComponent)
  private
    FOnClick: TNotifyEvent;       { declare a field to hold the method pointer }
    ...
  protected
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
  end;
```

To learn about TNotifyEvent and other event types, see Event-handler types.

As with any other property, you can set or change the value of an event at run time. The main advantage to having events be properties, however, is that component users can assign handlers to events at design time, using the Object Inspector.

See also
Events are method pointers
Event-handler types
Event handlers are optional

Event-handler types

See also

Because an event is a pointer to an event handler, the type of the event property must be a method pointer type. Similarly, any code to be used as an event handler must be an appropriately typed method of an object.

All event-handler methods are procedures. To be compatible with an event of a given type, an event-handler method must have the same number and type of parameters, in the same order, passed in the same way.

Delphi defines method types for all its standard events. When you create your own events, you can use an existing type if that's appropriate, or define one of your own.

Event-handler types are procedures

Although the compiler allows you to declare method-pointer types that are functions, you should never do so for handling events. Because an empty function returns an undefined result, an empty event handler that was a function might not always be valid. For this reason, all your events and their associated event handlers must be procedures.

Although an event handler cannot be a function, you can still get information back from the user's code using **var** parameters. When doing this, make sure you assign a valid value to the parameter before calling the handler so you don't require the user's code to change the value.

An example of passing **var** parameters to an event handler is the key-pressed event, of type TKeyPressEvent. TKeyPressEvent defines two parameters, one to indicate which object generated the event, and one to indicate which key was pressed:

```
type
  TKeyPressEvent = procedure(Sender: TObject; var Key: Char) of object;
```

Normally, the Key parameter contains the character pressed by the user. Under certain circumstances, however, the user of the component might want to change the character. One example might be to force all characters to uppercase in an editor. In that case, the user could define the following handler for keystrokes:

```
procedure TForm1.Edit1KeyPressed(Sender: TObject; var Key: Char);
begin
  Key := UpCase(Key);
end;
```

You can also use **var** parameters to let the user override the default handling.

See also

Events are method pointers

Events are properties

Event handlers are optional

Event handlers are optional

See also

The most important thing to remember when creating events for your components is that users of your components might not attach handlers to the events. That means that your component shouldn't fail or generate errors simply because a user of the component failed to attach a handler to a particular event.

The mechanics of calling handlers and dealing with events that have no attached handler are explained in Calling the event, but the principle has important implications for the design of your components and their events.

The optional nature of event handlers has two aspects:

- Component users do not have to handle events.

  Events happen almost constantly in a Windows application. Just by moving the mouse pointer across a component you cause Windows to send numerous mouse-move messages to the component, which the component translates into OnMouseMove events. In most cases, however, users of components don't care to handle the mouse move events, and this does not cause a problem. The component does not depend on the mouse events being handled.

  Similarly, the components you create should not be dependent on users handling the events they generate.

- Component users can write any code they want in an event handler.

  In general, there are no restrictions on the code a user can write in an event handler. The components in the Delphi component library all have events written in such a way that they minimize the chances of user-written code generating unexpected errors. Obviously, you can't protect against logic errors in user code, but you can, for example, ensure that all data structures are initialized before calling events so that users don't try to access invalid information.

See also

Events are method pointers

Events are properties

Event-handler types

Implementing the standard events

See also

All the controls that come with Delphi inherit events for all of the most common Windows events. Collectively, these are called the standard events. Although all these events are built into the standard controls, by default they are **protected**, meaning end users can't attach handlers to them. When you create a control, you can choose to make events visible to users of your control.

There are three things you need to consider when incorporating the standard events into your controls:

- What are the standard events?
- Making events visible
- Changing the standard event handling

See also

What are events?

Defining your own events

What are the standard events?

See also

There are two categories of standard events: those defined for all controls and those defined only for the standard windows controls.

Standard events for all controls

The most basic events are defined in the object type TControl. All controls, whether windowed or graphical or custom, inherit these events. The following table lists all the events available in all controls:

OnClick	OnDragDrop	OnEndDrag	OnMouseMove
OnDblClick	OnDragOver	OnMouseDown	OnMouseUp

All the standard events have corresponding protected dynamic methods declared in TControl, with names that correspond to the event names, but without the preceding "On." For example, OnClick events call a method named Click.

Standard events for standard controls

In addition to the events common to all controls, standard controls (those descended from TWinControl) have the following events:

OnEnter	OnKeyDown	OnKeyPress
OnKeyUp	OnExit	

As with the standard events in TControl, the windowed-control events have corresponding methods.

See also

Changing the standard event handling

Making events visible

Making events visible

See also Example

The declarations of the standard events are **protected**, as are the methods that correspond to them. If you want to make those events accessible to users either at run time or design time, you need to redeclare the event property as either **public** or **published**.

Redeclaring a property without specifying its implementation keeps the same implementation methods, but changes the protection level. Thus, you can take an event that's defined in the standard TControl but not made visible to users and promote it to a level that the user can see and use it.

Example

If you create a component that needs to surface the OnClick event at design time, you add the following to the component's type declaration:

```
type
  TMyControl = class(TCustomControl)
    ...
  published
    property OnClick;     { makes OnClick available in Object Inspector }
  end;
```

See also

What are the standard events?

Changing the standard event handling

Changing the standard event handling

See also Example

If you want to change the way your custom component responds to a certain kind of event, you might be tempted to write some code and assign it to the event. As a component user, that's exactly what you would do. However, when you're creating components you can't do that, because you need to keep the event available for the users of the component.

This is precisely the reason for the protected implementation methods associated with each of the standard events. By overriding the implementation method, you can modify the internal event handling, and by calling the inherited method you can maintain the standard handling, including the event for the user's code.

The order in which you call the inherited method is significant. As a general rule, you call the inherited method first, allowing the user's event-handler code to execute before your customizations (and in some cases, to keep from executing the customizations). However, there might be times when you want to execute your code before calling the inherited method. For example, if the inherited code is somehow dependent on the status of the component and your code changes that status, you should make the changes and then allow the user's code to respond to the changed status.

Example

Suppose you're writing a component and you want to modify the way your new component responds to clicks. Instead of assigning a handler to the OnClick event as a component user would do, you override the protected method Click:

```
procedure TMyControl.Click;
begin
  inherited Click; { perform standard handling, including calling handler }
  { your customizations go here }
end;
```

See also
What are the standard events?
Making events visible

Defining your own events

See also

Defining entirely new events is a relatively rare thing. More often you will refine the handling of existing events. However, there are times when a component introduces behavior that is entirely different from any other, so you'll need to define an event for it.

There are four steps involved in defining an event:

- Triggering the event
- Defining the handler type
- Declaring the event
- Calling the event

See also

What are events?

Implementing the standard events

Triggering the event

See also Example

The first issue you encounter when defining your own events that you don't need to consider when using the standard events is what triggers the event. For some events, the answer is obvious. For example, a mouse-down event occurs when the user presses the left button on the mouse and Windows sends a WM_LBUTTONDOWN message to the application. Upon receiving that message, a component calls its MouseDown method, which in turn calls any code the user has attached to the OnMouseDown event.

But some events are less clearly tied to specific external events. A scroll bar, for example, has an OnChange event, triggered by numerous kinds of occurrences, including keystrokes, mouse clicks, or changes in other controls. When defining your events, you must ensure that all the appropriate occurrences call the proper events.

Example

Here are the methods TControl uses to handle the WM_LBUTTONDOWN message from Windows. DoMouseDown is a private implementation method that provides generic handling for left, right, and middle button clicks, translating the parameters of the Windows message into values for the MouseDown method.

```
type
  TControl = class(TComponent)
  private
    FOnMouseDown: TMouseEvent;
    procedure DoMouseDown(var Message: TWMMouse; Button: TMouseButton;
      Shift: TShiftState);
    procedure WMLButtonDown(var Message: TWMLButtonDown); message WM_LBUTTONDOWN;
  protected
    procedure MouseDown(Button: TMouseButton; Shift: TShiftState;
      X, Y: Integer); dynamic;
  end;
...
procedure TControl.MouseDown(Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Assigned(FOnMouseDown) then
    FOnMouseDown(Self, Button, Shift, X, Y);     { call handler, if any }
end;

procedure TControl.DoMouseDown(var Message: TWMMouse; Button: TMouseButton;
  Shift: TShiftState);
begin
  with Message do
    MouseDown(Button, KeysToShiftState(Keys) + Shift, XPos, YPos);     { call dynamic
method }
end;

procedure TControl.WMLButtonDown(var Message: TWMLButtonDown);
begin
  inherited;       { perform default handling }
  if csCaptureMouse in ControlStyle then MouseCapture := True;
  if csClickEvents in ControlStyle then Include(FControlState, csClicked);
  DoMouseDown(Message, mbLeft, []);      { call the generic mouse-down method }
end;
```

Two kinds of events

There are two kinds of occurrences you might need to provide events for: state changes and user interactions. The mechanics of handling them are the same, but the semantics are slightly different.

User-interaction events will nearly always be triggered by a message from Windows, indicating that the user did something your component might need to respond to. State-change events might also be related to messages from Windows (focus changes or being enabled, for example), but they can also occur through changes in properties or other code. You have total control over the triggering of events you define. You should be consistent and complete so that users of your components know how to use the events.

See also

Defining the handler type

Declaring the event

Calling the event

Defining the handler type

See also

Once you determine that your event occurred, you have to define how you want the event handled. That means determining the type of the event handler. In most cases, handlers for the events you define yourself will be either simple notifications or event-specific types. It's also possible to get information back from the handler.

Simple notifications

A notification event is one that only tells you that the particular event happened, with no specific information about when or where. Notifications use the type TNotifyEvent, which carries only one parameter, the sender of the event. Thus, all a handler for a notification "knows" about the event is what kind of event it was, and what component the event happened to. For example, click events are notifications. When you write a handler for a click event, all you know is that a click occurred and which component was clicked.

Notification is a one-way process. There is no mechanism to provide feedback or prevent further handling of a notification.

Event-specific handlers

In some cases, it's not enough to know just what event happened and what component it happened to. For example, if the event is a key-press event, it's likely that the handler will want to know which key the user pressed. In these cases, you need handler types that include parameters with any necessary information about the event.

If your event was generated in response to a message, it's likely that the parameters you pass to the event handler come directly from the message parameters.

Returning information from the handler

Because all event handlers are procedures, the only way to pass information back from a handler is through a **var** parameter. Your components can use such information to determine how or whether to process an event after the user's handler executes.

For example, all the key events (OnKeyDown, OnKeyUp, and OnKeyPress) pass the value of the key pressed in a **var** parameter named Key. The event handler can change Key so that the application sees a different key as being involved in the event. This is a way to, for instance, force typed characters to uppercase.

See also

Triggering the event

Declaring the event

Calling the event

Declaring the event

See also

Once you've determined the type of your event handler, you're ready to declare the method pointer and the property for the event. Be sure to give the event a meaningful and descriptive name so that users can understand what the event does. Try to be consistent with names of similar properties in other components.

Event names start with "On"

The names of all the standard events in Delphi begin with "On." This is only a convention; the compiler doesn't enforce it. The Object Inspector determines that a property is an event by looking at the type of the property: all method-pointer properties are assumed to be events and appear on the Events page.

Component users expect to find events in the alphabetical list of names starting with "On." Using other kinds of names will likely confuse them.

See also

Triggering the event

Defining the handler type

Calling the event

Calling the event

See also

In general, it's best to centralize calls to an event. That is, create a virtual method in your component that calls the user's event handler (if the user assigns one) and provides any default handling.

Putting all the event calls in one place ensures that someone deriving a new component from your component can customize event handling by overriding that one method, rather than searching through your code looking for places you call the event.

There are two other considerations when calling the event:

- Empty handlers must be valid
- Users can override default handling

See also

Triggering the event

Defining the handler type

Declaring the event

Empty handlers must be valid

See also Example

You should never create a situation in which an empty event handler causes an error. That is, the proper functioning of your component should not depend on a particular response from the user's event-handler code. In fact, an empty handler should produce the same result as no handler at all.

Components should never require the user to use them in a particular way. An important aspect of that principle is that component users expect no restrictions on what they can do in an event handler.

Example

Since an empty handler should behave the same as no handler, the code for calling the user's handler should look like this:

```
if Assigned(OnClick) then OnClick(Self);
{ perform default handling }
```

You should never have something like this:

```
if Assigned(OnClick) then OnClick(Self)
else { perform default handling };
```

See also
Users can override default handling

Users can override default handling

See also Example

For some kinds of events, the user might want to replace the default handling or even suppress all responses. To enable users to do that, you need to pass a **var** parameter to the handler and check for a certain value when the handler returns.

Note that this is in keeping with the notion that empty handlers should have the same effect as no handler at all: since an empty handler won't change the values of any **var** parameters, the default handling always takes place after calling the empty handler.

Example

When handling key-press events the user can suppress the component's default handling of the keystroke by setting the **var** parameter Key to a null character (#0). The logic for supporting that looks like this:

```
if Assigned(OnKeyPress) then OnKeyPress(Self, Key);
if Key <> #0 then  { perform default handling };
```

The actual code is a little different from this because it's dealing with Windows messages, but the logic is the same. By default, the component calls any user-assigned handler, then performs its standard handling. If the user's handler sets Key to a null character, the component skips the default handling.

See also
Empty handlers must be valid

Creating methods

See also

Component methods are no different from any other object's methods. That is, they are just procedures and functions built into the structure of a component object. Although there are essentially no restrictions on what you can do with the methods of a component, Delphi does use some standards you should follow.

The guidelines to follow when writing methods for your components include

- Avoiding interdependencies
- Naming methods
- Protecting methods
- Making methods virtual
- Declaring methods

As a general rule, minimize the number of methods users need to call to use your components. A lot of the features you might be inclined to implement as methods are probably better encapsulated into properties. Doing so provides an interface that suits the Delphi environment, and also lets users access them at design time.

See also

Overview of component creation

OOP for component writers

Creating properties

Creating events

Using graphics in components

Handling messages

Registering components

Avoiding interdependencies

See also

At all times when writing components, minimize the preconditions imposed on the component user. To the greatest extent possible, component users should able to do anything they want to a component, whenever they want to do it. There will be times when you can't accommodate that, but your goal should be to come as close as possible.

Although it's impossible to list all the possible kinds of interdependencies you want to avoid, this list gives you an idea of the kinds of things to avoid:

- Methods that the user must call in order to use the component
- Methods that need to execute in a particular order
- Methods that put the component into a state or mode where certain events or methods could be invalid

The best way to handle these situations is to ensure that you provide ways out of them. For example, if calling a method puts your component into a state where calling another method might be invalid, then write that other method in such a way that if the user calls it when the component is in a bad state, the method corrects that state before executing its main code. At a minimum, you should raise an exception in cases when a user calls a method that is invalid.

In other words, if you create a situation where parts of your code depend on each other, the burden should be on you to be sure that using the code in incorrect ways does not cause the user problems. A warning message, for example, is preferable to crashing if the user doesn't accommodate your interdependencies.

See also
Naming methods
Protecting methods
Making methods virtual
Declaring methods

Naming methods

See also

Delphi imposes no restrictions on what you name methods or their parameters. However, there are a few conventions that make methods easier for users of your components. Keep in mind that the nature of a component architecture dictates that many different kinds of people might use your components.

If you're accustomed to writing code that only you or a small group of programmers uses, you might not think too much about how you name things. It is a good idea to make your method names clear because people unfamiliar with your code (and even unfamiliar with coding) might have to use your components.

Here are some suggestions for making clear method names:

- Make names descriptive.

  A name like PasteFromClipboard is much more informative than simply Paste or PFC.

- Procedure names should be active.

  Use active verbs in your procedure names. For example, ReadFileNames is much more helpful than DoFiles.

- Function names should reflect the nature of what they return.

  Although it might be obvious to you as a programmer that a function named X returns the horizontal position of something, a name like GetHorizontalPosition is more universally understandable.

As a final consideration, make sure the method really needs to be a method. A good guideline is that method names have verbs in them. If you find that you create a lot of methods that do not have verbs in their names, consider whether those methods ought to be properties.

See also
Avoiding interdependencies
Protecting methods
Making methods virtual
Declaring methods

Protecting methods

See also

All parts of objects, including fields, methods, and properties, can have various levels of protection, as explained in Controlling access. Choosing the appropriate level of protection for methods is quite straightforward.

As a general rule, methods you write in your components will be either **public** or **protected**. The exception to that rule is methods that implement properties, which should always be **private**. Otherwise, you rarely need to make a method **private**, unless it is truly specific to that particular type of component, to the point that even components derived from it should not have access to it.

Note: There is generally no reason for declaring a method (other than an event handler) as **published**. Doing so looks to the end user exactly as if the method were **public**.

Methods that should be public

Methods that should be protected

Methods that should be private

See also

Avoiding interdependencies

Naming methods

Making methods virtual

Declaring methods

Methods that should be public

See also

Any methods that users of your components need to be able to call must be declared as **public**. Keep in mind that most method calls occur in event handlers, so methods should avoid unduly tying up system resources or putting Windows in a state where it can't respond to the user.

Note: Constructors and destructors should always be public.

See also

Methods that should be protected

Methods that should be private

Methods that should be protected

See also

Any methods that are implementation methods for the component should be **protected**. That keeps users from calling them at the wrong time. If you have methods that a user's code should not call, but which descendant objects will need to call, you should declare the methods as **protected**.

For example, suppose you have a method that relies on having certain data set up for it beforehand. If you make that method public, there's a chance a user will call it before setting up the data. On the other hand, by making it protected, you ensure that the user can't call it directly. You can then set up other, public methods that ensure that data setup occurs before calling the protected method.

See also
Methods that should be public
Methods that should be private

Methods that should be private

See also

The one category of methods that should always be private is property-implementation methods. You definitely don't want users calling methods that manipulate the data for a property. They should only access that information by accessing the property itself. You can ensure that by making the property public and its implementation methods private.

If descendant objects need to override the implementation of a method, they can do so, but instead of overriding the implementation methods, they must access the inherited property value through the property itself.

See also

Methods that should be public

Methods that should be protected

Making methods virtual

See also

Virtual methods in Delphi components are no different from virtual methods in other objects. You make methods virtual when you want different types to be able to execute different code in response to the same method call.

If you create components intended to be used directly by end users, you can probably make all your methods static. On the other hand, if you create components of a more abstract nature, which other component writers will use as the starting point for their own components, consider making the added methods virtual. That way, components derived from your components can override the inherited virtual methods.

See also
Avoiding interdependencies
Naming methods
Protecting methods
Declaring methods

Declaring methods

See also Example

Declaring a method in a component is no different from declaring any object method.

To declare a new method in a component, you do two things:

- Add the declaration to the component's object-type declaration
- Implement the method in the **implementation** part of the component's unit

Example

The following code shows a component that defines two new methods: one protected static method and one public virtual method.

```
type
  TSampleComponent = class(TControl)
  protected
    procedure MakeBigger; { declare protected static method }
  public
    function CalculateArea: Integer; virtual;    { declare public virtual method }
  end;
...
implementation
...
procedure TSampleComponent.MakeBigger;   { implement first method }
begin
  Height := Height + 5;
  Width := Width + 5;
end;

function TSampleComponent.CalculateArea: Integer;       { implement second method }
begin
  Result := Width * Height;
end;
```

See also
Avoiding interdependencies
Naming methods
Protecting methods
Making methods virtual

Using graphics in components

See also

Windows provides a powerful Graphics Device Interface (GDI) for drawing device-independent graphics. Unfortunately, GDI imposes a lot of extra requirements on the programmer, such as managing graphic resources. As a result, you spend a lot of time doing things other than what you really want to do: creating graphics.

Delphi takes care of all the GDI drudgery for you, allowing you to spend your time doing productive work instead of searching for lost handles or unreleased resources. Delphi tackles the tedious tasks so you can focus on the productive and creative ones.

Note that, as with any part of the Windows API, you can call GDI functions directly from your Delphi application if you want to. However, you will probably find that using Delphi's encapsulation of the graphic functions is a much more productive way to create graphics.

There are several important topics dealing with graphics in Delphi:

- Overview of Delphi graphics
- Using the canvas
- Working with pictures
- Offscreen bitmaps
- Responding to changes

See also

Overview of component creation
OOP for component writers
Creating properties
Creating events
Creating methods
Handling messages
Registering components

Overview of Delphi graphics

See also Example

Delphi encapsulates the Windows GDI at several levels. The most important to you as a component writer is the way components display their images on the screen. When calling GDI functions directly, you need to have a handle to a device context, into which you have selected various drawing tools such as pens and brushes and fonts. After rendering your graphic images, you must then restore the device context to its original state before disposing of it.

Instead of forcing you to deal with graphics at that detailed level, Delphi provides a simple yet complete interface: your component's Canvas property. The canvas takes care of making sure it has a valid device context, and releases the context when you're not using it. Similarly, the canvas has its own properties representing the current pen, brush, and font.

The canvas manages all those resources for you, so you need not concern yourself with creating, selecting, and releasing things such as pen handles. You just tell the canvas what kind of pen it should use, and it takes care of the rest.

One of the benefits of letting Delphi manage graphic resources is that it can cache resources for later use, which can greatly speed up repetitive operations. For example, if you have a program that repeatedly creates, uses, and disposes of a particular kind of pen tool, you need to repeat those steps each time you use it. Because Delphi caches graphic resources, chances are good that a tool you use repeatedly is still in the cache, so instead of having to recreate a tool, Delphi reuses an existing one.

Example

As an example of how much simpler Delphi's graphics code can be, here are two samples of code. The first uses standard GDI functions to draw a yellow ellipse outlined in blue on a window in an application written with ObjectWindows. The second uses a canvas to draw the same ellipse in an application written with Delphi.

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
var
  PenHandle, OldPenHandle: HPEN;
  BrushHandle, OldBrushHandle: HBRUSH;
begin
  PenHandle := CreatePen(PS_SOLID, 1, RGB(0, 0, 255));  { create blue pen }
  OldPenHandle := SelectObject(PaintDC, PenHandle);     { tell DC to use blue pen }
  BrushHandle := CreateSolidBrush(RGB(255, 255, 0));    { create a yellow brush }
  OldBrushHandle := SelectObject(PaintDC, BrushHandle); { tell DC to use yellow brush }
  Ellipse(HDC, 10, 10, 50, 50);   { draw the ellipse }
  SelectObject(OldBrushHandle);   { restore original brush }
  DeleteObject(BrushHandle);      { delete yellow brush }
  SelectObject(OldPenHandle);     { restore original pen }
  DeleteObject(PenHandle);{ destroy blue pen }
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
  with Canvas do
  begin
    Pen.Color := clBlue;  { make the pen blue }
    Brush.Color := clYellow;      { make the brush yellow }
    Ellipse(10, 10, 50, 50);      { draw the ellipse }
  end;
end;
```

See also

Using the canvas

Working with pictures

Offscreen bitmaps

Responding to changes

Using the canvas

See also

The canvas object encapsulates Windows graphics at several levels, ranging from high-level functions for drawing individual lines, shapes, and text to intermediate-level properties for manipulating the drawing capabilities of the canvas to low-level access to the Windows GDI.

The following table summarizes the capabilities of the canvas.

Level	**Operation**	**Tools**
High	Drawing lines and shapes	Methods such as MoveTo, LineTo, Rectangle, and Ellipse
	Displaying and measuring text	TextOut, TextHeight, TextWidth, and TextRect methods
	Filling areas	FillRect and FloodFill methods
Intermediate	Customizing text and graphics	Pen, Brush, and Font properties
	Manipulating pixels	Pixels property
	Copying and merging images	Draw, StretchDraw, BrushCopy, and CopyRect methods; CopyMode property
Low	Calling Windows GDI functions	Handle property

See also
Overview of Delphi graphics
Working with pictures
Offscreen bitmaps
Responding to changes

Working with pictures

See also

Most of the graphics work you do in Delphi is limited to drawing directly on the canvases of components and forms. Delphi also provides for handling standalone graphic images, however, such as bitmaps, metafiles, and icons, including automatic management of palettes.

There are three important aspects to working with pictures in Delphi:

- Picture or graphic or canvas?
- Graphics in files
- Handling palettes

See also
Overview of Delphi graphics
Using the canvas
Offscreen bitmaps
Responding to changes

Picture or graphic or canvas?

See also

There are three kinds of objects in Delphi that deal with graphics:

- A canvas, which represents a bitmapped drawing surface on a form, graphic control, printer, or bitmap. A canvas is always a property of something else, never a standalone object.
- A graphic, which represents a graphic image of the sort usually found in a file or resource, such as a bitmap, icon, or metafile. Delphi defines object types TBitmap, TIcon, and TMetafile, all descended from a generic TGraphic. You can also define your own graphic objects. By defining a minimal standard interface for all graphics, TGraphic provides a simple mechanism for applications to use different kinds of graphics easily.
- A picture, which is a container for a graphic, meaning it could contain any of the graphic object types. That is, an item of type TPicture can contain a bitmap, an icon, a metafile, or a user-defined graphic type, and an application can access them all in the same way through the picture object. For example, the image control has a property called Picture, of type TPicture, enabling the control to display images from many kinds of graphics.

Keep in mind that a picture object always has a graphic, and a graphic might have a canvas (the only standard graphic that has a canvas is TBitmap). Normally, when dealing with a picture, you work only with the parts of the graphic object exposed through TPicture. If you need access to the specifics of the graphic object itself, you can refer to the picture's Graphic property.

See also

Graphics in files

Handling palettes

Graphics in files

See also Example

All pictures and graphics in Delphi can load their images from files and store them back again (or into different files). You can load or store the image of a picture at any time.

To load an image into a picture from a file, call the picture's LoadFromFile method.

To save an image from a picture into a file, call the picture's SaveToFile method.

LoadFromFile and SaveToFile each take the name of a file as the only parameter. LoadFromFile uses the extension of the file name to determine what kind of graphic object it will create and load. SaveToFile saves whatever type of file is appropriate for the type of graphic object being saved.

Example

To load a bitmap into an image control's picture pass the name of a bitmap file to the picture's LoadFromFile method:

```
procedure TForm1.LoadBitmapClick(Sender: TObject);
begin
  Image1.Picture.LoadFromFile('RANDOM.BMP');
end;
```

The picture recognized .BMP as the standard extension for bitmap files, so it creates its graphic as a TBitmap, then calls that graphic's LoadFromFile method. Since the graphic is a bitmap, it loads the image from the file as a bitmap.

See also
Picture or graphic or canvas?
Handling palettes

Handling palettes

See also

When running on a palette-based device, Delphi controls automatically support palette realization. That is, if you have a control that has a palette, you can use two methods inherited from TControl to control how Windows accommodates that palette.

Palette support for controls has these two aspects:

- Specifying a palette for a control
- Responding to palette changes

Most controls have no need for a palette. However, controls that contain graphic images (such as the image control) might need to interact with Windows and the screen device driver to ensure the proper appearance of the control. Windows refers to this process as realizing palettes.

Realizing palettes is the process of ensuring that the frontmost window uses its full palette, and that windows in the background use as much of their palettes as possible, then map any other colors to the closest available colors in the "real" palette. As windows move in front of one another, Windows continually realizes the palettes.

Note: Delphi itself provides no specific support for creating or maintaining palettes, other than in bitmaps. However, if you have a palette handle, Delphi controls can manage it for you.

See also

Picture or graphic or canvas?

Graphics in files

Specifying a palette for a control

See also Example

If you have a palette you want to apply to a control, you can tell your application to use that palette.

To specify a palette for a control, override the control's GetPalette method to return the handle of the palette.

Specifying the palette for a control does two things for your application:

1 It tells the application that your control's palette needs to be realized.

2 It designates the palette to use for realization.

Example

The clearest example of a control using a palette is the image control, TImage. The image control gets its palette (if any) from the picture it contains. TImage overrides GetPalette to return the palette of its picture:

```
type
  TImage = class(TGraphicControl)
  protected
    function GetPalette: HPALETTE; override;     { override the method }
  ...
  end;
...
function TImage.GetPalette: HPALETTE;
begin
  Result := 0;     { default result is no palette }
  if FPicture.Graphic is TBitmap then    { only bitmaps have palettes }
    Result := TBitmap(FPicture.Graphic).Palette; { use it if available }
end;
```

See also
Responding to palette changes

Responding to palette changes

See also

If your control specifies a palette by overriding GetPalette, Delphi automatically takes care of responding to palette messages from Windows. The method that handles the palette messages is PaletteChanged. For normal operation, you should never need to alter the default behavior of PaletteChanged.

The primary role of PaletteChanged is to determine whether to realize the control's palette in the foreground or the background. Windows handles this realization of palettes by making the topmost window have a foreground palette, with other windows resolved in background palettes. Delphi goes one step farther, in that it also realizes palettes for controls within a window in tab order. The only time you might need to override this default behavior is if you want a control that is not first in tab order to have the foreground palette.

See also
Specifying a palette for a control

Offscreen bitmaps

See also

When drawing complex graphic images, a common technique in Windows programming is to create an offscreen bitmap, draw the image on the bitmap, and then copy the complete image from the bitmap to the final destination onscreen. Using an offscreen image reduces flicker caused by repeated drawing directly to the screen.

The bitmap object in Delphi to represent bitmapped images in resources and files can also work as an offscreen image.

There are two main aspects to working with offscreen bitmaps:

- Creating and managing offscreen bitmaps
- Copying bitmapped images

See also
Overview of Delphi graphics
Using the canvas
Working with pictures
Responding to changes

Creating and managing offscreen bitmaps

See also   Example

When creating complex graphic images, you should generally avoid drawing them directly on a canvas that appears onscreen. Instead of drawing on the canvas for a form or control, you can construct a bitmap object, draw on its canvas, and then copy its completed image to the onscreen canvas.

The most common use of an offscreen bitmap is in the Paint method of a graphic control. As with any temporary object, the bitmap should be protected with a **try**..**finally** block:

```
type
  TFancyControl = class(TGraphicControl)
  protected
    procedure Paint; override;    { override the Paint method }
  end;

procedure TFancyControl.Paint;
var
  Bitmap: TBitmap; { temporary variable for the offscreen bitmap }
begin
  Bitmap := TBitmap.Create;       { construct the bitmap object }
  try
    { draw on the bitmap }
    { copy the result into the control's canvas }
  finally
    Bitmap.Free;   { destroy the bitmap object }
  end;
end;
```

Example

For an example of painting a complex image on an offscreen bitmap, see the source code for the Gauge control from the Samples page of the Component palette. The gauge draws its different shapes and text on an offscreen bitmap before copying them to the screen. Source code for the gauge is in the file GAUGES.PAS in the SOURCE\SAMPLES subdirectory.

See also
Copying bitmapped images

Copying bitmapped images

See also

Delphi provides four different ways to copy images from one canvas to another. Depending on the effect you want to create, you call different methods.

The following table summarizes the image-copying methods in canvas objects.

To create this effect	Call this method
Copy an entire graphic	Draw
Copy and resize a graphic	StretchDraw
Copy part of a canvas	CopyRect
Copy a bitmap with raster operations	BrushCopy

See also
Creating and managing offscreen bitmaps

Responding to changes

See also Example

All graphic objects, including canvases and their owned objects (pens, brushes, and fonts) have events built into them for responding to changes in the object. By using these events, you can make your components (or the applications that use them) respond to changes by redrawing their images.

Responding to changes in graphic objects is particularly important if you publish those objects as part of the design-time interface of your components. The only way to ensure that the design-time appearance of the component matches the properties set in the Object Inspector is to respond to changes in the objects.

To respond to changes in a graphic object, assign a method to the object's OnChange event.

Example

The shape component publishes properties representing the pen and brush it uses to draw its shape. The component's constructor assigns a method to the OnChange event of each, causing the component to refresh its image if either the pen or brush changes:

```
type
  TShape = class(TGraphicControl)
  public
    procedure StyleChanged(Sender: TObject);
  end;
...
implementation
...
constructor TShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);       { always call the inherited constructor! }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;    { construct the pen }
  FPen.OnChange := StyleChanged;  { assign method to OnChange event }
  FBrush := TBrush.Create;{ construct the brush }
  FBrush.OnChange := StyleChanged;       { assign method to OnChange event }
end;

procedure TShape.StyleChanged(Sender: TObject);
begin
  Invalidate(True);       { erase and repaint the component }
end;
```

See also
Overview of Delphi graphics
Using the canvas
Working with pictures
Offscreen bitmaps

Handling messages

See also

One of the keys to traditional Windows programming is handling the messages sent by Windows to applications. Delphi handles most of the common ones for you, but in the course of creating components it is possible either that you will need to handle messages that Delphi doesn't already handle or that you will create your own messages and need to handle them.

There are three aspects to working with messages:

- Understanding the message-handling system
- Changing message handling
- Creating new message handlers

See also

Overview of component creation

OOP for component writers

Creating properties

Creating events

Creating methods

Using graphics in components

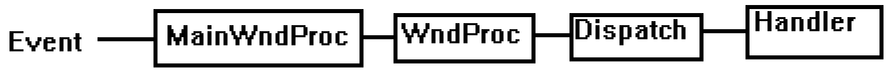
Registering components

Understanding the message-handling system

See also

All Delphi objects have a built-in mechanism for handling messages, called message-handling methods or message handlers. The basic idea of message handlers is that the object receives messages of some sort and dispatches them, calling one of a set of specified methods depending on the message received. If no specific method exists for a particular message, there is a default handler.

The following diagram shows the message-dispatch system:



The Delphi component library defines a message-dispatching system that translates all Windows messages (including user-defined messages) directed to a particular object into method calls. You should never need to alter this message-dispatch mechanism. All you'll need to do is create message-handling methods.

See also

Changing message handling

Creating new message handlers

What's in a Windows message?

See also

A Windows message is a data record that contains several useful fields. The most important of these is an integer-size value that identifies the message. Windows defines a lot of messages, and the Messages unit declares identifiers for all of them. The other useful information in a message comes in two parameter fields and a result field.

One parameter contains 16 bits, the other 32 bits. You often see Windows code that refers to those values as wParam and lParam, for "word parameter" and "long parameter." Often, each parameter will contain more than one piece of information, and you see references to names such as lParamHi, which refers to the high-order word in the long parameter.

Originally, Windows programmers had to remember (or look up) what each parameter contained. More recently, Microsoft has named the parameters. This so-called "message cracking" makes it much simpler to understand what information accompanies each message. For example, the parameters to the WM_KEYDOWN message are now called VKey and KeyData, which gives much more specific information than wParam and lParam.

Delphi defines a specific record type for each different type of message that gives a mnemonic name to each parameter. For example, mouse messages pass the x- and y-coordinates of the mouse event in the long parameter, one in the high-order word, and the other in the low-order word. Using the mouse-message record, however, you need not concern yourself with which word is which, because you refer to the parameters by the names XPos and YPos instead of lParamLo and lParamHi.

See also

Using message parameters

Dispatching messages

Dispatching messages

See also

When an application creates a window, it registers a window procedure with the Windows kernel. The window procedure is the routine that handles messages for the window. Traditionally, the window procedure contains a huge **case** statement with entries for each message the window has to handle. Keep in mind that "window" in this sense means just about anything on the screen: each window, each control, and so on. Every time you create a new type of window, you have to create a complete window procedure.

Delphi simplifies message dispatching in several ways:

- Every component inherits a complete message-dispatching system.
- The dispatch system has default handling. You only define handlers for messages you need to respond to specially.
- You can modify small parts of the message-handling and rely on inherited methods for most processing.

The greatest benefit of this message dispatch system is that you can safely send any message to any component at any time. If the component doesn't have a handler defined for the message, the default handling takes care of it, usually by ignoring the message.

Tracing the flow of messages

Delphi registers a method called MainWndProc as the window procedure for each type of component in an application. MainWndProc contains an exception-handling block, passing the message record from Windows to a virtual method called WndProc and handling any exceptions by calling the application object's HandleException method.

MainWndProc is a static method that contains no special handling for any particular messages. Customizations take place in WndProc, since each component type can override the method to suit its particular needs.

WndProc methods check for any special conditions that affect their processing so they can "trap" unwanted messages. For example, while being dragged, components ignore keyboard events, so TWinControl.WndProc only passes along keyboard events if the component is not being dragged. Ultimately, WndProc calls Dispatch, a static method inherited from TObject, which determines what method to call to handle the message.

Dispatch uses the Msg field of the message record to determine how to dispatch a particular message. If the component defines a handler for that particular message, Dispatch calls the method. If the component doesn't define a handler for that message, Dispatch calls DefaultHandler.

See also

Understanding the message-handling system

What's in a Windows message?

Changing message handling

See also

Before changing the message-handling of your components, make sure that's what you really want to do. Delphi translates most Windows messages into events that both the component writer and the component user can handle. Rather than changing the message-handling behavior, you should probably change the event-handling behavior.

To change the message handling, you override the message-handling method. You can also prevent a component from handling a message under certain circumstances by trapping the message.

See also

Understanding the message-handling system

Creating new message handlers

Overriding the handler method

See also Example

To change the way a component handles a particular message, you override the message-handling method for that message. If the component doesn't already handle the particular message, you need to declare a new message-handling method.

To override a message-handling method, you declare a new method in your component with the same message index as the method it overrides. Do not use the **override** directive; you must use the **message** directive and a matching message index.

Note that the name of the method and the type of the single **var** parameter do not have to match the overridden method. Only the message index is significant. However, for clarity, it is best to follow the convention of naming message-handling methods after the messages they handle.

Example

For example, to override a component's handling of the WM_PAINT message, you redeclare the WMPaint method:

```
type
  TMyComponent = class(...)
    ...
    procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
  end;
```

See also

Using message parameters

Trapping messages

Using message parameters

See also Example

Once inside a message-handling method, your component has access to all the parameters of the message record. Since the message is always a **var** parameter, the handler can change the values of the parameters if necessary. The only parameter that changes frequently is the Result field. Result is the value the Windows documentation refers to as the "return value" for the message: the value returned by the SendMessage call that sends the message.

Because the type of the Message parameter to the message-handling method varies with the message being handled, you should refer to the documentation on Windows messages for the names and meanings of individual parameters.

If for some reason you need to refer to the message parameters by their old-style names (wParam, lParam, and so on), you can typecast Message to the generic type TMessage, which uses those parameter names.

See also
What's in a Windows message?
Overriding the handler method
Trapping messages

Trapping messages

See also Example

Under certain circumstances, you might want your components to ignore certain messages. That is, you want to keep the component from dispatching the message to its handler. To trap a message that way, you override the virtual method WndProc.

The WndProc method screens messages before passing them to the Dispatch method, which in turn determines which method gets to handle the message. By overriding WndProc, your component gets a chance to filter out messages before dispatching them.

In general, an override of WndProc looks like this:

```
procedure TMyControl.WndProc(var Message: TMessage);
begin
  { tests to determine whether to continue processing }
  inherited WndProc(Message);     { dispatch normally }
end;
```

Example

Here is part of the WndProc method for TControl. TControl defines entire ranges of mouse messages that it filters when a user is dragging and dropping controls. Overriding WndProc helps this in two ways:

- It can filter ranges of messages instead of having to specify handlers for each one.
- It can preclude dispatching the message at all, so the handlers are never called.

```
procedure TControl.WndProc(var Message: TMessage);
begin
  ...
  if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
    if Dragging then      { handle dragging specially }
      DragMouseMsg(TWMMouse(Message))
    else
      ...  { handle others normally }
    end;
...  {otherwise process normally }
end;
```

See also

Overriding the handler method

Using message parameters

Creating new message handlers

See also

Since Delphi provides handlers for most common Windows messages, the time you will most likely need to create new message handlers is when you define your own messages.

Working with user-defined messages has two aspects:

- Defining your own messages
- Declaring a new message-handling method

See also

Understanding the message-handling system

Changing message handling

Defining your own messages

See also

A number of the standard components define messages for internal use. The most common reasons for defining messages are broadcasting information not covered by standard Windows messages and notification of state changes.

Defining a message is a two-step process. The steps are

- Declaring a message identifier
- Declaring a message-record type

See also
Declaring a new message-handling method

Declaring a message identifier

See also Example

A message identifier is an integer-sized constant. Windows reserves the messages below 1,024 for its own use, so when you declare your own messages you should start above that level.

The constant WM_USER represents the starting number for user-defined messages. When defining message identifiers, you should base them on WM_USER.

Be aware that some standard Windows controls use messages in the user-defined range. These include list boxes, combo boxes, edit boxes, and command buttons. If you derive a component from one of these and want to define a new message for it, be sure to check the Messages unit to see what messages Windows already defines for that control.

Example

The following code shows two user-defined messages:

```
const
  WM_MYFIRSTMESSAGE = WM_USER + 0;
  WM_MYSECONDMESSAGE = WM_USER + 1;
```

See also
Declaring a message-record type

Declaring a message-record type

See also Example

If you want to give useful names to the parameters of your message, you need to declare a message-record type for that message. The message record is the type of the parameter passed to the message-handling method. If you don't use the message's parameters, or if you want to use the old-style parameter notation (wParam, lParam, and so on), you can use the default message record, TMessage.

To declare a message-record type, follow these conventions:

1 Name the record type after the message, preceded by a T.

2 Call the first field in the record Msg, of type TMsgParam.

3 Define the next two bytes to correspond to the word-size parameter.

4 Define the next four bytes to correspond to the long parameter.

5 Add a final field called Result, of type LongInt.

Example

For example, here is the message record for all mouse messages, TWMMouse:

```
type
  TWMMouse = record
    Msg: TMsgParam;       { first is the message ID }
    Keys: Word;    { this is the wParam }
    case Integer of       { two ways to look at the lParam }
      0: (
        XPos: Integer;    { either as x- and y-coordinates... }
        YPos: Integer);
      1: (
        Pos: TPoint;      { ...or as a single point }
        Result: Longint); { and finally, the result field }
  end;
```

Note that TWMMouse uses a variant record to define two different sets of names for the same parameters.

See also
Declaring a message identifier

Declaring a new message-handling method

See also Example

There are two sets of circumstances that require you to declare new message-handling methods:

- Your component needs to handle a Windows message that isn't already handled by the standard components.
- You have defined your own message for use by your components.

To declare a message-handling method, do the following:

1 Declare the method in a **protected** part of the component's class declaration.

2 Make the method a procedure.

3 Name the method after the message it handles, but without any underline characters.

4 Pass a single **var** parameter called Message, of the type of the message record.

5 Write code for any handling specific to the component.

6 Call the inherited message handler.

Example

Here's the declaration of a message handler for a user-defined message called CM_CHANGECOLOR:

```
type
  TMyComponent = class(TControl)
    ...
    procedure CMChangeColor(var Message: TMessage); message CM_CHANGECOLOR;
  end;

procedure TMyComponent.CMChangeColor(var Message: TMessage);
begin
  Color := Message.lParam;{ set color from long parameter }
  inherited;       { call inherited handler }
end;
```

See also
Defining your own messages

ClassInfo method

See also

Applies to

TObject (all objects)

Declaration

```
class function ClassInfo: Pointer;
```

Description

The ClassInfo method returns a pointer to the run-time type information (RTTI) table for the object type. The RTTI table contains information about the object type, its ancestor type, and all its published properties, methods, and events.

RTTI exists mostly for the internal use of the Delphi environment. If you need further descriptions of the contents of the RTTI table, see the TypInfo unit.

See also
ClassName method
ClassParent method
ClassType method

ClassName method

See also Example

Applies to

TObject (all objects)

Declaration

```
class function ClassName: ShortString;
```

Description

The ClassName method returns a string containing the name of the actual type of an object. For example, you can assign any type of object to a variable of type TObject. If you then call that variable's ClassName method, it returns the actual type of the assigned object, rather than TObject.

For most purposes, you don't need the name of an object type. If you need to know the type of an object, the **is** operator or the ClassType method provide more useful information than ClassName.

Example

The following code shows an OnDragDrop event handler for a list box on a form that accepts drops from any kind of object. This handler adds the name of the type of the source of the dragged item to the list box, no matter what type. Note that the Source parameter, although it is of type TObject, "knows" what type of object it really refers to.

```
procedure TForm1.ListBox1DragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  Accept := True;  { accept from any source }
end;

procedure TForm1.ListBox1DragDrop(Sender, Source: TObject; X, Y: Integer);
begin
  ListBox1.Items.Add(Source.ClassName);
end;
```

See also
ClassInfo method
ClassParent method
ClassType method

ClassNameIs method

See also

Applies to

TObject (all objects)

Declaration

```
class function ClassNameIs(const Name: string): Boolean;
```

Description

The ClassNameIs method returns True if the long string passed in the Name parameter matches the name of the class. Otherwise, it returns False. In most cases, you'll find it simpler to check the value of the ClassName property, but ClassNameIs can be more efficient when querying objects in other modules.

See also
ClassName method

ClassParent method

See also

Applies to

TObject (all objects)

Declaration

```
class function ClassParent: TClass;
```

Description

The ClassParent method returns the type of the immediate ancestor of the object type. You can use ClassParent to determine whether a particular object or type of object is assignment compatible with another. ClassParent returns **nil** for type TObject.

The InheritsFrom method and the **is** and **as** operators use the ClassParent method, but you will rarely have need to call it.

See also
ClassInfo method
ClassName method
ClassType method

ClassType method

See also Example

Applies to

TObject (all objects)

Declaration

```
function ClassType: TClass;
```

Description

The ClassType method returns the type of the object. In essence, this allows your code to dynamically determine the specific kind of object it is using. The **is** operator provides similar information, but can determine whether an object is of a specific type or of an assignment-compatible type. Both the **is** and **as** operators call the ClassType method.

Note: Unlike the other methods that return information about object types (ClassInfo, ClassName, ClassParent), ClassType is not a class method. That is, you can only call ClassType to determine the type of an instance, not of an object-type reference.

Example

The following code shows two different ways to check the type of the Sender parameter to an OnDragOver event handler for a list box. The first handler allows the user to drop items dragged only from items of type TLabel. The second accepts items dragged either from items of type TLabel or from any descendant of TLabel.

```
procedure TForm1.ListBox1DragOver(Sender, SOurce: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  if Sender.ClassType = TLabel then Accept := True;
end;

procedure TForm1.ListBox1DragOver(Sender, SOurce: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  if Sender is TLabel then Accept := True;
end;
```

See also
ClassInfo method
ClassName method
ClassParent method

CleanupInstance method

See also

Applies to

TObject (all objects)

Declaration

```
procedure CleanupInstance;
```

Description

The CleanupInstance method performs finalization on any long strings and records within the class. The FreeInstance method calls CleanupInstance.

See also
FreeInstance method

Create method

See also

Applies to

TObject (all objects)

Declaration

```
constructor Create;
```

Description

The Create method constructs a new object instance. Create returns an instance of the type being created, allocated on the global heap. As with all constructors, Create calls the NewInstance method to allocate the memory for the instance and the InitInstance method to initialize the allocated memory before executing its own code.

By default, Create allocates the number of bytes returned by the InstanceSize method and initializes the allocated memory to zeros.

When declaring new component types, always add the **override** directive if your new component declares a Create method. The Create method of TComponent is virtual, so to ensure that Delphi calls the correct constructor when a user drops your component on a form, you must override the Create method.

Note: When you override the Create constructor in a descendant object type, you should call the inherited Create to complete the initialization of inherited fields and properties. Always use the **inherited** keyword when calling the inherited Create, rather than specifying the ancestor type, as calling AncestorType.Create actually constructs an additional instance of that ancestor type.

See also

Destroy method

NewInstance method

InitInstance method

InstanceSize method

DefaultHandler method

See also

Applies to

TObject

Declaration

```
procedure DefaultHandler(var Message); virtual;
```

Description

The DefaultHandler method provides message handling for all messages that an object does not have specific handlers for. The DefaultHandler method defined by TObject does nothing; that is, it performs no processing on the message record passed to it in the Message parameter.

Calling **inherited** in a message-handling method results in a call to the ancestor's DefaultHandler method if that ancestor does not specify a handler for the message being handled.

TControl and TWinControl both override DefaultHandler to handle messages for all their descendant types. TControl.DefaultHandler handles the messages Windows sends to manage the control's text: WM_GETTEXT, WM_SETTEXT, and WM_GETTEXTLENGTH. TWinControl.DefaultHandler passes any otherwise-unhandled messages to the control's window procedure by calling the CallWindowProc API function.

See also
Dispatch method

Destroy method

See also

Applies to

TObject (all objects)

Declaration

```
destructor Destroy; virtual;
```

Description

The Destroy method destroys and disposes of an object instance. You should rarely call Destroy directly, but rather call Free instead, since Free checks to ensure that the object instance is not **nil** before calling Destroy.

Note: When you declare a Destroy method in an object type, always add the **override** directive to the declaration and call the inherited Destroy as the last statement in the redeclared method. Since Destroy is a virtual method, overriding it ensures that the proper inherited behavior occurs.

See also

Create method

Free method

FreeInstance method

Dispatch method

See also

Applies to

TObject (all objects)

Declaration

```
procedure Dispatch(var Message);
```

Description

The Dispatch method calls message-handling methods ("handlers") for the object, based on the contents of the Message parameter. Message is untyped, so you can pass any kind of data to Dispatch, but you will usually pass a message record such as TMessage or one of the specific message-record types.

The only assumption Dispatch makes about the data in Message is that the first two bytes contain a message ID, which is an integer-type number that determines which message handler Dispatch calls to handle the message. If the object has no handler assigned for that particular message, Dispatch calls DefaultHandler instead.

To dispatch a particular message, Dispatch first looks in the list of message handlers declared for the object called on to dispatch the message. If the object does not specifically handle the message, Dispatch checks the message-handler list of the ancestor type, and continues checking ancestors until it either finds a specific handler or runs out of ancestors, in which case it calls DefaultHandler.

See also

DefaultHandler method

Perform method

FieldAddress method

See also

Applies to

TObject (all objects)

Declaration

```
function FieldAddress(const Name: ShortString): Pointer;
```

Description

The FieldAddress method returns the address of the object field named by the Name parameter. If the object has no published field by that name, FieldAddress returns **nil**.

See also
MethodAddress method

Free method

See also

Applies to

TObject (all objects)

Declaration

```
procedure Free;
```

Description

The Free method calls the Destroy method if the object instance is non-**nil**. It is generally a good idea to call Free instead of calling Destroy directly.

See also
Destroy method

FreeInstance method

See also

Applies to

TObject (all objects)

Declaration

```
procedure FreeInstance; virtual;
```

Description

The FreeInstance method deallocates the memory allocated by the NewInstance method. Both NewInstance and FreeInstance call InstanceSize to determine the amount of memory to allocate and deallocate.

You should only override FreeInstance if you also override NewInstance.

See also

NewInstance method

InstanceSize method

InheritsFrom method

See also Example

Applies to

TObject (all objects)

Declaration

```
class function InheritsFrom(AClass: TClass): Boolean;
```

Description

The InheritsFrom method returns True if the object type specified in the AClass parameter is an ancestor of the object type or the type of the object itself. Otherwise, it returns False. The **is** and **as** operators use InheritsFrom in their implementations.

Because InheritsFrom is a class method, you can use it to determine the relationship of two object types. The **is** operator, on the other hand, can only determine the inheritance relationship of an instance.

Example

The following event handler for an edit box click event contains two statements that perform equivalent operations:

```
procedure TForm1.Edit1Click(Sender: TObject);
begin
  if Sender is TEdit then TEdit(Sender).Color := clGreen;
  if Sender.InheritsFrom(TEdit) then TEdit(Sender).Color := clGreen;
end;
```

See also
ClassParent method

InitInstance method

See also

Applies to

TObject (all objects)

Declaration

```
class procedure InitInstance(Instance: Pointer): TObject;
```

Description

The InitInstance method initializes a newly-allocated object instance to all zeros. Note that InitInstance is not virtual, so you cannot override it.

See also

Create method

NewInstance method

InstanceSize method

See also

Applies to

TObject (all objects)

Declaration

```
class function InstanceSize: Longint;
```

Description

The InstanceSize method returns the size in bytes of each instance of the object type. You can call InstanceSize to determine how much memory to allocate for a pool of objects.

The NewInstance method calls InstanceSize to determine how much memory to allocate from the heap to contain a particular instance. The FreeInstance method also uses the value returned from InstanceSize to deallocate the object's memory.

Note: InstanceSize is not a virtual method, so you cannot override it. If you override NewInstance to change the amount of memory allocated for the instance, you should also override FreeInstance to free the same amount.

See also

NewInstance method

FreeInstance method

MethodAddress method

See also

Applies to

TObject (all objects)

Declaration

```
class function MethodAddress(const Name: ShortString): Pointer;
```

Description

The MethodAddress method returns the address of the published method named by the Name parameter. If the object has no published method by that name, MethodAddress returns **nil**.

See also

MethodName method

FieldAddress method

MethodName method

See also

Applies to

TObject (all objects)

Declaration

```
class function MethodName(Address: Pointer): ShortString;
```

Description

The MethodName method returns a string containing the name of the method located at Address. If Address does not point to a published method of the object, MethodName returns an empty string.

See also
MethodAddress method

NewInstance method

See also

Applies to

TObject (all objects)

Declaration

```
class function NewInstance: TObject; virtual;
```

Description

The NewInstance method allocates memory for each instance of an object type and returns a pointer to that new instance. You should never call NewInstance directly; all constructors call NewInstance automatically.

You can, however, override NewInstance to change the way a particular type allocates its memory. For example, if you need to allocate a large number of identical objects that all need to be in memory at the same time, you could allocate a large block of memory for the entire group, then override NewInstance to use part of that larger block for each instance.

Note: If you override NewInstance to change the memory allocation for an object type, you might also need to override FreeInstance to deallocate that same memory.

Both NewInstance and FreeInstance use the value returned by InstanceSize to determine the amount of memory to allocate or deallocate.

See also

FreeInstance method

InstanceSize method

TClass type

See also

Unit

System

Declaration

```
type
    TClass = class of TObject;
```

Description

The TClass type is the generic type for class references. Just as TObject is assignment-compatible with objects of any type, TClass is assignment-compatible with any class-reference type. VCL objects and components uses TClass in cases where a property needs to accept class-type references of any type.

See also
TObject object (protected)

TObject object (protected)

See also Methods Objects

Unit

System

Description

The TObject object type is the ultimate ancestor of every object (and therefore, every component) in Delphi. If you do not specify an ancestor type when you declare new object type, Delphi automatically uses TObject as the ancestor.

TObject declares a number of methods that every object inherits or overrides:

- Construction and destruction

  Every object has a constructor called Create that initializes the object, calling NewInstance to allocate the memory for the instance and InitInstance to fill that memory with zeros.

  There is also a virtual destructor called Destroy. In addition, there is a method named Free that calls Destroy only if the instance is non-**nil**. Destroy calls FreeInstance to release the memory allocated by NewInstance. FreeInstance in turn calls CleanupInstance to finalize long strings and other data structures before their memory is deallocated.

- Type-information methods

  The methods ClassInfo, ClassName, ClassNameIs, ClassParent, ClassType, InheritsFrom, and InstanceSize all provide useful information about the object type or its instances.

  In addition, you can get run-time type information (RTTI) on published parts of an object from the following methods: FieldAddress, MethodAddress, and MethodName.

- Message-handling methods

  Every object also has built-in support for the handling of messages, with the Dispatch and DefaultHandler methods.

Methods

ClassInfo	DefaultHandler	InheritsFrom
ClassName	Destroy	InitInstance
ClassNameIs	Dispatch	InstanceSize
ClassParent	FieldAddress	MethodAddress
ClassType	Free	MethodName
CleanupInstance	FreeInstance	NewInstance
Create		

See also
TClass type

ActivateHint method

See also

Applies to

THintWindow

Declaration

```
procedure ActivateHint(Rect: TRect; const AHint: string); virtual;
```

Description

The ActivateHint method activates the hint window, setting its caption to the string passed in AHint and its bounding rectangle to Rect. By default, ActivateHint calls the SetWindowPos API function to display the hint window.

After activation, all aplication messages go through the hint window's IsHintMsg method, which determines whether to deactivate the hint window.

To change the appearance of the hint window, override the hint window's Paint method. The hint window should not modify the rectangle passed in Rect, since the application has explicitly set the rectangle.

See also

IsHintMsg method

Paint method

AlignControls method

See also

Applies to

TWinControl

Declaration

```
procedure AlignControls(AControl: TControl; var Rect: TRect); virtual;
```

Description

The AlignControls method aligns any controls for which the control is the parent within the rectangle passed in Rect. AlignControls uses the Align property value for each child control to determine how to align it. The AControl parameter can be **nil**. If you specify a control in AControl, that control takes precedence in alignment over other, similarly-aligned controls.

See also
Align property

Broadcast method

See also

Applies to

TWinControl

Declaration

```
procedure Broadcast(var Message);
```

Description

The Broadcast method sends the message passed in the Message parameter to each of the child controls of the windowed control.

See also
NotifyControls method

CancelModes method

Applies to
TControl

Declaration
```
procedure CancelModes;
```

Description
The CancelModes method gets the control out of any modal states that could interfere with other operations. Such modal states include having lists dropped down, menus open, mouse captured, and so on. Calling CancelModes ensures that the control is is condition to begin a different modal state.

Canvas property

See also

Applies to

TGraphicControl, TCustomControl

Declaration

```
property Canvas: TCanvas;
```

Description

The Canvas property of a graphic control or custom control is a read-only property providing access to the control's drawing surface at run time. The most important use for Canvas is in Paint methods.

See also
Paint method

ChangeScale method

See also

Applies to

TControl

Declaration

```
procedure ChangeScale(M, D: Integer); dynamic;
```

Description

The ChangeScale method repositions and resizes the control by the ratio M/D. The ScaleBy method calls ChangeScale to perform its rescaling. TWinControl overrides ChangeScale to also rescale any controls for which the control is the parent before rescaling itself. TForm also overrides ChangeScale to adjust its client area and font size as needed.

See also
ScaleBy method

Click method

Applies to
TControl, TMediaPlayer

Description
There are two different methods called Click.   One is the protected implementation method for OnClick events in all controls, introduced by TControl. The other is a protected click-response method in the Media Player component.

Click method for all controls

Click method for media player components

Click method

See also

Applies to

TControl

Declaration

```
procedure Click; dynamic;
```

Description

The Click method is the protected implementation method for a control's OnClick event. The Click method defined in TControl does nothing except call any event handler attached to the OnClick event. You can override Click to provide other responses in addition to the inherited event-handler call.

A control calls Click whenever it receives a left-button mouse-up message (WM_LBUTTONUP) if that message indicates that the mouse up occurred near the corresponding mouse-down. Many specific controls call Click in other circumstances, such as when a user types the shortcut key for a button or menu item.

The description of the OnClick event lists all the occurrences that trigger click events.

See also
OnClick event

Component messages

See also

There are a number of messages defind by Delphi, passed among components to indicate state changes that other components might need to respond to. Your components can handle these messages just as they would any Windows message.

Most of the component messages are remappings of standard Windows messages. When a Delphi component receives a Windows message, it usually needs to pass along the message to its owned components. Instead of passing along that same message, Delphi components usually send a corresponding component message. By mapping the messages into a range unused by Windows, Delphi makes sure it does not get spurious responses from components.

The following table lists the component messages defined by Delphi.

CM_ACTIVATE	CM_ENTER	CM_PARENTCTRL3DCHANGED
CM_APPKEYDOWN	CM_EXIT	CM_PARENTFONTCHANGED
CM_APPSYSCOMMAND	CM_FOCUSCHANGED	CM_PARENTSHOWHINTCHANGED
CM_BASE	CM_FONTCHANGE	CM_RELEASE
CM_BUTTONPRESSED	CM_FONTCHANGED	CM_SHOWHINTCHANGED
CM_CANCELMODE	CM_GOTFOCUS	CM_SHOWINGCHANGED
CM_COLORCHANGED	CM_HITTEST	CM_SYSCOLORCHANGE
CM_CTL3DCHANGED	CM_ICONCHANGED	CM_TABFONTCHANGED
CM_CURSORCHANGED	CM_INVOKEHELP	CM_TEXTCHANGED
CM_DEACTIVATE	CM_LOSTFOCUS	CM_TIMECHANGE
CM_DESIGNHITTEST	CM_MOUSEENTER	CM_VISIBLECHANGED
CM_DIALOGKEY	CM_MOUSELEAVE	CM_WANTSPECIALKEY
CM_DIALOGCHAR	CM_MENUCHANGED	CM_WINDOWHOOK
CM_ENABLEDCHANGED	CM_PARENTCOLORCHANGED	CM_WININICHANGE

See also
Component notifications

Component notifications

See also

Component notifications are a special kind of component message. They are defined in a specific range of message that do not conflict with standard Windows messages. A component notification is always a message sent from a parent to its children.

The following table lists the component notifications defined by Delphi.

CN_BASE	CN_DELETEITEM	CN_PARENTNOTIFY
CN_CHAR	CN_DRAWITEM	CN_SYSKEYDOWN
CN_CHARTOITEM	CN_HSCROLL	CN_SYSCHAR
CN_COMMAND	CN_KEYDOWN	CN_VKEYTOITEM
CN_COMPAREITEM	CN_KEYUP	CN_VSCROLL
CN_CTLCOLOR	CN_MEASUREITEM	

See also
Component messages

Control property

Applies to

TControlCanvas

Declaration

```
property Control: TControl;
```

Description

The Control property is the control associated with the control-canvas object. The control-canvas object uses the Control property to get information from its associated control.

ControlState property

See also

Applies to

TControl

Declaration

```
property ControlState: TControlState;
```

Description

The ControlState property holds a set of values indicating the current state of a control at run time. The various flags in the ControlState property indicate such conditions as having been clicked or needing alignment.

The items in ControlState are all specific to controls, and are in addition to the state flags in the ComponentState property, which apply to all components.

See also
TControlState type

ControlStyle property

See also

Applies to

TControl

Declaration

```
property ControlStyle: TControlStyle;
```

Description

The ControlStyle property contains a set of style flags indicating certain attributes of a control type. These flags indicate such styles as whether the control captures mouse events or has a fixed size.

In general, these flags do not change among different instances of a type of control, nor do they vary at run time. Such transient conditions are indicated by the ControlState property.

See also
TControlStyle type

CreateHandle method

See also

Applies to

TWinControl, TControlCanvas, TCanvas

Declaration

```
procedure CreateHandle; virtual;
```

Description

The CreateHandle method generates a handle for the Handle property of the object if it does not already have one.

- For windowed controls, CreateHandle calls the CreateWnd method to create the window handle.
- For canvas objects, CreateHandle does nothing.
- For control-canvas objects, CreateHandle obtains a device-context handle from the control associated with the canvas.

See also
Handle property
HandleNeeded method
WndProc method

CreateParams method

See also Example

Applies to

TWinControl

Declaration

```
procedure CreateParams(var Params: TCreateParams); virtual;
```

Description

The CreateParams method initializes the window-creation parameter record passed in the Params parameter. TWinControl implements CreateParams by setting all the fields of Params to generic base values. A number of the standard controls override CreateParams to change one or more of the default values in Params.

The CreateWnd method calls CreateParams to initialize the parameters it will pass to CreateWindowHandle. By overriding CreateParams, you can customize the way a component creates its Windows representation. When overriding CreateParams, you should always call the inherited method first to set the default values, then make any desired adjustments.

Example

The TNoteBook component overrides CreateParams to ensure that the control clips any child controls at the boundaries of the notebook:

```
procedure TNotebook.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params); { call the inherited first }
  with Params do Style := Style or WS_CLIPCHILDREN;     { add a style flag }
end;
```

See also

CreateWnd method

CreateWindowHandle method

CreateSubClass method

See also Example

Applies to

TWinControl

Declaration

```
procedure CreateSubClass(var Params: TCreateParams; ControlClassName:
  PChar);
```

Description

The CreateSubClass method generates a Delphi component derived from an existing Windows window class. Call CreateSubClass in the CreateParams method of a subclassed control, after calling the inherited CreateParams. Delphi uses CreateSubClass to provide access to the standard Windows controls.

Example

Here is the CreateParams method for the standard Delphi button component, TButton, which subclasses the window class called BUTTON:

```
procedure TButton.CreateParams(var Params: TCreateParams);
const
  ButtonStyles: array[Boolean] of Longint =(BS_PUSHBUTTON, BS_DEFPUSHBUTTON);
begin
  inherited CreateParams(Params); { fill in default values }
  CreateSubClass(Params, 'BUTTON');       { subclass from BUTTON }
  with Params do Style := Style or ButtonStyles[FDefault]       { use Default property, too }
end;
```

See also
CreateParams method

CreateWindowHandle method

See also

Applies to

TWinControl

Declaration

```
procedure CreateWindowHandle(const Params: TCreateParams); virtual;
```

Description

The CreateWindowHandle method creates a window handle by calling the CreateWindowEx API function, passing parameters from the record passed in Params.

See also
CreateWnd method
DestroyWindowHandle method

CreateWnd method

See also

Applies to

TWinControl

Declaration

```
procedure CreateWnd; virtual;
```

Description

The CreateWnd method creates a Windows control corresponding to the windowed-control component.

CreateWnd first calls the CreateParams method to initialize the window-creation parameters, then calls CreateWindowHandle to create the window handle for the control. Adjusts the size of the newly-created window and then sets the control's font by calling the Perform method, passing the WM_SETFONT message.

See also

RecreateWnd method

DestroyWnd method

CursorToIdent function

See also

Unit

Controls

Declaration

```
function CursorToIdent(Cursor: Longint; var Ident: string): Boolean;
```

Description

The CursorToIdent function converts a cursor value passed in Cursor to an identifier string returned in Ident. CursorToIdent returns True if Cursor corresponds to one of the standard system cursors, represented by the identifiers in the TCursor type, and sets Ident to a string containing the cursor's name. If Cursor is not one of the named cursors, CursorToIdent returns False and does not change Ident.

See also

TCursor type

IdentToCursor function

CursorToString function

See also

Unit

Controls

Declaration

```
function CursorToString(Cursor: TCursor): string;
```

Description

The CursorToString function returns a string representing the cursor value passed in the Cursor parameter. CursorToString calls CursorToIdent to obtain the names of the standard cursor types. If CursorToIdent cannot locate an appropriate name, CursorToString converts the numeric value of Cursor to its string representation.

See also

CursorToIdent function

StringToCursor function

DblClick method

See also

Unit

Controls

Applies to

TControl

Declaration

```
procedure DblClick; dynamic;
```

Description

The DblClick method is the protected implementation method for a control's OnDblClick event. The DblClick method inherited from TControl does nothing except call any event handler attached to the OnDblClick event. You can override DblClick to provide other responses in addition to the inherited event-handler call.

A control calls the DblClick method in response to a left-button double-click message (WM_LBUTTONDBLCLICK) from Windows.

See also
OnDblClick event

DefWndProc property

See also

Applies to

TWinControl

Declaration

```
property DefWndProc: Pointer;
```

Description

The DefWndProc property specifies the default window procedure for the windowed control. DefWndProc initially points to the default procedure defined in the window-creation parameters defined in CreateParams. Windowed controls call the default window procedure to handle all messages in their DefaultHandler method, as long as the windowed control has a valid window handle.

In essence, calling DefWndProc to handle a message is equivalent to asking for the standard Windows handling of that message. DefWndProc fills the role for windowed controls that DefaultHandler fills for all Delphi objects.

See also

CreateParams method

WndProc method

DestroyHandle method

See also

Applies to

TWinControl

Declaration

```
procedure DestroyHandle;
```

Description

The DestroyHandle method destroys the windowed control's window handle without destroying the control. The control can later recreate the handle if needed. DestroyHandle is the coverse operation to CreateHandle. In general, it is best to call the high-level CreateHandle and DestroyHandle methods, rather than the lower level methods such as CreateWnd and DestroyWnd.

If the control has windowed controls as child controls, DestroyHandle calls each of their DestroyHandle methods before calling DestroyWnd to destroy its own handle.

See also
CreateHandle method

DestroyWindowHandle method

See also

Applies to

TWinControl

Declaration

```
procedure DestroyWindowHandle; virtual;
```

Description

The DestroyWindowHandle method calls the Windows API function DestroyWindow to destroy the window handle created in the CreateWindowHandle method. TWinControl's Destroy method calls DestroyWindowHandle to destroy any window handle associated with a windowed control before destroying the object. The DestroyWnd method also calls DestroyWindowHandle.

See also

CreateWindowHandle method

DestroyWnd method

DestroyWnd method

See also

Applies to

TWinControl

Declaration

```
procedure DestroyWnd; virtual;
```

Description

The DestroyWnd method destroys the windowed control's window handle, first saving a copy of the control's text in internal storage, then freeing any device contexts and finally calling DestroyWindowHandle.

DestroyWnd is the converse operation to the CreateWnd method.

See also

CreateWnd method

DestroyWindowHandle method

DisableAlign method

See also

Applies to
TWinControl

Declaration
```
procedure DisableAlign;
```

Description
The DisableAlign method temporarily disables the realigning of controls within the windowed control. It is a good idea to disable alignment while performing multiple manipulations of controls, such as reading from a form file or scaling. Each call to DisableAlign must have a corresponding call to EnableAlign.

DisableAlign increments a reference count that EnableAlign later decrements. When the reference count reaches zero, EnableAlign performs any needed realignments.

See also

EnableAlign method

DoEnter method

See also

Applies to

TWinControl

Declaration

```
procedure DoEnter; dynamic;
```

Description

The DoEnter method is the protected implementation method for a windowed control's OnEnter event. The DoEnter method defined in TWinControl does nothing except call any event handler attached to the OnEnter event. You can override DoEnter to provide other responses in addition to the inherited event-handler call.

See also

OnEnter event

DoExit method

DoExit method

See also

Applies to

TWinControl

Declaration

```
procedure DoExit; dynamic;
```

Description

The DoExit method is the protected implementation method for a windowed control's OnExit event. The DoExit method defined in TWinControl does nothing except call any event handler attached to the OnExit event. You can override DoExit to provide other responses in addition to the inherited event-handler call.

See also

OnExit event

DoEnter method

DoKeyDown method

See also

Applies to

TWinControl

Declaration

```
procedure DoKeyDown(var Message: TWMKey): Boolean;
```

Description

The DoKeyDown method is a protected method that performs some preprocessing before calling the KeyDown method that implements the OnKeyDown event. In most cases, you should not need to override DoKeyDown, as it is primarily concerned with filtering messages to determine whether the control should actually handle them.

The return value from DoKeyDown determines whether the application should continue processing the key-down message passed in the Message parameter. A return value of True indicates that the key-down occurrence has been handled completely, and no further processing is needed. A return value of False indicates that the application should continue passing the key-down message through the inherited processing.

By default, DoKeyDown allows parent forms with the KeyPreview property set to True to preemptively process the key-down message. If the form does not handle the message, DoKeyDown translates the message parameters into the appropriate types and calls KeyDown, which in turn calls the OnKeyDown event handler, if any. If KeyDown does not suppress further processing, DoKeyDown then checks for presses of the F1 key to invoke the application's Help system, if any.

See also

KeyDown method

OnKeyDown event

DoKeyPress method

DoKeyUp method

DoKeyPress method

See also

Applies to

TWinControl

Declaration

```
procedure DoKeyPress(var Message: TWMKey): Boolean;
```

Description

The DoKeyPress method is a protected method that performs some preprocessing before calling the KeyPress method that implements the OnKeyPress event. In most cases, you should not need to override DoKeyPress, as it is primarily concerned with filtering messages to determine whether the control should actually handle them.

The return value from DoKeyPress determines whether the application should continue processing the key-press message passed in the Message parameter. A return value of True indicates that the key-press occurrence has been handled completely, and no further processing is needed. A return value of False indicates that the application should continue passing the key-press message through the inherited processing.

By default, DoKeyPress allows parent forms with the KeyPreview property set to True to preemptively process the key-down message. If the form does not handle the message, DoKeyPress translates the message parameters into the appropriate types and calls KeyPress, which in turn calls the OnKeyPress event handler, if any.

See also
KeyPress method
OnKeyPress event
DoKeyDown method
DoKeyUp method

DoKeyUp method

See also

Applies to

TWinControl

Declaration

```
procedure DoKeyUp(var Message: TWMKey): Boolean;
```

Description

The DoKeyUp method is a protected method that performs some preprocessing before calling the KeyUp method that implements the OnKeyUp event. In most cases, you should not need to override DoKeyUp, as it is primarily concerned with filtering messages to determine whether the control should actually handle them.

The return value from DoKeyUp determines whether the application should continue processing the key-up message passed in the Message parameter. A return value of True indicates that the key-up occurrence has been handled completely, and no further processing is needed. A return value of False indicates that the application should continue passing the key-up message through the inherited processing.

By default, DoKeyUp allows parent forms with the KeyPreview property set to True to preemptively process the key-down message. If the form does not handle the message, DoKeyUp translates the message parameters into the appropriate types and calls KeyUp, which in turn calls the OnKeyUp event handler, if any.

See also
KeyUp method
OnKeyUp event
DoKeyDown method
DoKeyPress method

DragCanceled method

See also

Applies to

TControl

Declaration

```
procedure DragCanceled; dynamic;
```

Description

The DragCanceled method provides an opportunity for a control from which a drag-and-drop operation started (called the source) to respond when the user cancels the operation. The user cancels the operation by dropping the item over a component that does not accept the item or by pressing Esc while dragging.

Delphi calls the source control's DragCanceled method before invoking the OnEndDrag event. By default, DragCanceled does nothing, but you can override it to respond to the cancellation of the drag.

See also
OnEndDrag event

EnableAlign method

See also

Applies to

TWinControl

Declaration

```
procedure EnableAlign;
```

Description

The EnableAlign method decrements the reference count incremented by a call to the DisableAlign method. Calls to DisableAlign should always be paired with calls to EnableAlign. When the reference count reaches zero, EnableAlign calls the Realign method to perform any pending realignments.

See also

DisableAlign method

Realign method

FindControl function

Unit
Controls

Declaration
```
function FindControl(Handle: HWnd): TWinControl;
```

Description
The FindControl function returns the windowed control corresponding to the window handle passed in the Handle parameter. If Handle is zero or there is no windowed control corresponding to Handle, FindControl returns **nil**.

FindDragTarget function

See also

Unit

Controls

Declaration

```
function FindDragTarget(const Pos: TPoint; AllowDisabled: Boolean):
  TControl;
```

Description

The FindDragTarget function returns the control at the screen coordinates passed in Pos. The AllowDisabled parameter specifies whether to include disabled controls in the search. If there is no control at the specified location (or if there is a disabled control, but AllowDisabled is False), FindDragTarget returns False.

See also
ControlAtPos method

FindNextControl method

See also

Applies to

TWinControl

Declaration

```
function FindNextControl(CurControl: TWinControl;
    GoForward, CheckTabStop, CheckParent: Boolean): TWinControl;
```

Description

The FindNextControl method returns the windowed control's next child control in tab order after CurControl. If CurControl is not a child of the windowed control, FindNextControl returns the first child control in tab order.

The GoForward parameter controls the direction of the search. If GoForward is True, FindNextControl searches forward through the child controls in tab order.

The CheckTabStop and CheckParent parameters control whether FindNextControl performs certain checks on the controls it finds. If CheckTabStop is True, the returned control must have its TabStop property set to True. If CheckParent is True, the returned control's Parent property must indicate the windowed control.

FindNextControl calls the GetTabOrderList method to build its list of possible "next" controls.

See also
GetTabOrderList method

FreeHandle method

See also

Applies to

TControlCanvas

Declaration

```
procedure FreeHandle;
```

Description

The FreeHandle method releases the device-context handle used by the control-canvas object.

See also
CreateHandle method

GetCaptureControl function

See also

Unit

Controls

Declaration

```
function GetCaptureControl: TControl;
```

Description

The GetCaptureControl function returns the control that has currently "captured" the mouse. If there is no such control, GetCaptureControl returns **nil**.

The description of the MouseCapture property explains capturing the mouse.

See also

MouseCapture property

SetCaptureControl procedure

GetClientOrigin method

See also

Applies to

TControl, TWinControl

Declaration

```
function GetClientOrigin: TPoint; virtual;
```

Description

The GetClientOrigin method is a protected access method for the ClientOrigin property.

GetClientOrigin returns a point indicating the position of the top left corner of the control in screen coordinates. TControl implements GetClientOrigin by adding the control's Left and Top values to its parent controls ClientOrigin. TWinControl overrides GetClientOrigin to make use of the Windows API function ClientToScreen.

See also

ClientOrigin property

GetClientRect method

GetClientRect method

See also

Applies to

TControl, TWinControl, TForm

Declaration

```
function GetClientRect: TRect; virtual;
```

Description

The GetClientRect method is a protected access method for the ClientRect property.

GetClientRect returns a rectangle defining the client area of the control. TControl implements GetClientRect to return a rectangle with its Top and Left fields set to zero, and its Bottom and Right fields set to the control's Height and Width, respectively. TWinControl overrides GetClientRect to call the Windows API function GetClientRect. TForm also overrides GetClientRect to return the rectangle inside the form's frame.

See also
GetClientOrigin method

GetCursorValues procedure

See also

Unit

Controls

Declaration

```
procedure GetCursorValues(Proc: TGetStrProc);
```

Description

The GetCursorValues procedure builds a list of the names of the cursors available to the current application by calling the method passed in Proc once for each cursor registered with the system, passing the name of each.

See also
CursorToString function

GetDeviceContext method

See also

Applies to

TControl, TWinControl

Declaration

```
function GetDeviceContext(var WindowHandle: HWnd): HDC; virtual;
```

Description

The GetDeviceContext method returns a device context for the control represented by the window handle passed in the WindowHandle parameter. TControl implements GetDeviceContext by returning the device context returned from its parent control's GetDeviceContext. TWinControl overrides GetDeviceContext to call the Windows API function GetDC, passing the windowed control's Handle property and setting WindowHandle to Handle.

See also
PaletteChanged method

GetPalette method

See also

Applies to

TControl

Declaration

```
function GetPalette: HPALETTE;
```

Description

The GetPalette method specifies a palette to use for a control. Delphi provides no specific support for creating or maintaining palettes. However, if you have a palette handle, you can override GetPalette for a control to cause the control to use that palette.

Example

The Image component overrides GetPalette to assign the palette from its associated bitmap as the component's palette:

```
function TImage.GetPalette: HPALETTE;
begin
  Result := 0;     { default result is no palette }
  if FPicture.Graphic is TBitmap then     { only bitmaps have palettes }
    Result := TBitmap(FPicture.Graphic).Palette; { use it if available }
end;
```

See also
PaletteChanged method

GetTabOrderList method

See also

Applies to

TWinControl

Declaration

```
procedure GetTabOrderList(List: TList);
```

Description

The GetTabOrderList method builds a list of controls in tab order by iterating through the windowed control's internal tab-order list, adding each of the controls to List, including any controls contained in those controls. The result is a list of all the controls and their owned controls, in tab order.

The FindNextControl method calls GetTabOrderList to build a complete list of the controls that might be "next."

See also
FindNextControl method

IdentToCursor function

See also

Unit

Controls

Declaration

```
function IdentToCursor(const Ident: string; var Cursor: Longint): Boolean;
```

Description

The IdentToCursor function converts a cursor identifier into its corresponding cursor value by searching through the list of cursors available to the application and finding one with a name that matches the string passed in Ident. If the search produces a match, IdentToCursor sets the Cursor parameter to the cursor value and returns True. If the search produces no match, IdentToCursor returns False and makes no change to Cursor.

See also
CursorToIdent function

IsControl property

See also

Applies to

TControl

Declaration

```
property IsControl: Boolean;
```

Description

The IsControl property determines whether a form stores its form-specific properties to its form file. By default, IsControl is always False for all controls, including forms (since TForm descends from TControl). IsControl has no effect, except when saving forms.

You can use the Delphi forms designer to create complex controls such as panels by creating those controls as forms, placing and naming their contained controls, and attaching code to events. You can then save the form, edit the form as text, and in the text version set IsControl to True. The next time you reload and save the form, only those form properties appropriate to use as a control are stored, not the properties specific to TForm. You can then edit the form file as text, changing the type of the component from TForm to the desired control type, such as TPanel.

See also
TDesigner object

IsControlMouseMsg method

See also

Applies to

TWinControl

Declaration

```
function IsControlMouseMsg(var Message: TWMMouse): Boolean;
```

Description

The IsControlMouseMsg method returns True if the mouse message passed in the Message parameter is directed to one of the windowed control's child controls. Windows takes care of sending messages to windowed child controls, but for nonwindowed child controls, Windows sends the messages to the parent control, which must then determine which, if any, of its child controls should receive the message.

The WndProc method of a windowed control calls IsControlMouseMsg to process all mouse message sent to the windowed control.

See also
WndProc method

IsHintMsg method

See also

Applies to

THintWindow

Declaration

```
function IsHintMsg(var Msg: TMsg): Boolean; virtual;
```

Description

The IsHintMsg method checks application messages while the hint window is on the screen (after the call to ActivateHint). Upon seeing a mouse, keyboard, command, or activation message, IsHintMsg returns True, causing the application to hide the hint window.

See also
ActivateHint method

KeyDown method

See also

Applies to

TWinControl

Declaration

```
procedure KeyDown(var Key: Word; Shift: TShiftState); dynamic;
```

Description

The KeyDown method is the protected implementation method for a windowed control's OnKeyDown event. The KeyDown method inherited from TWinControl does nothing except call any event handler attached to the OnKeyDown event. You can override KeyDown to provide other responses in addition to the inherited event-handler call.

A windowed control calls KeyDown in response to a key-down message (WM_KEYDOWN) from Windows. The actual sequence is that the message goes to a private message handler that calls the DoKeyDown method. If DoKeyDown determines that the control should, in fact, process the character, it decodes the parameters of the key-down message and passes the key code and shift-key state to KeyDown in the Key and Shift parameters, respectively.

Either KeyDown or the OnKeyDown event handler it calls can suppress further processing of a key by setting the Key parameter to zero.

See also

OnKeyDown event

DoKeyDown method

KeyPress method

KeyUp method

KeyPress method

See also

Applies to

TWinControl

Declaration

```
procedure KeyPress(var Key: Char); dynamic;
```

Description

The KeyPress method is the protected implementation method for a windowed control's OnKeyPress event.   The KeyPress method inherited from TWinControl does nothing except call any event handler attached to the OnKeyPress event. You can override KeyPress to provide other responses in addition to the inherited event-handler call.

A windowed control calls KeyPress in response to a key-press   message (WM_CHAR) from Windows. The actual sequence is that the message goes to a private message handler that calls the DoKeyPress method.   If DoKeyPress determines that the control should, in fact, process the character, it decodes the parameters of the key-down message and passes the key code to KeyPress in the Key parameters, respectively.

Either KeyPress or the OnKeyPress event handler it calls can suppress further processing of a character by setting the Key parameter to zero.

See also

OnKeyPress event

DoKeyPress method

KeyDown method

KeyUp method

KeyUp method

See also

Applies to

TWinControl

Declaration

```
procedure KeyUp(var Key: Word; Shift: TShiftState); dynamic;
```

Description

The KeyUp method is the protected implementation method for a windowed control's OnKeyUp event. The KeyUp method inherited from TWinControl does nothing except call any event handler attached to the OnKeyUp event. You can override KeyUp to provide other responses in addition to the inherited event-handler call.

A windowed control calls KeyUp in response to a key-up message (WM_KEYUP) from Windows. The actual sequence is that the message goes to a private message handler that calls the DoKeyUp method. If DoKeyUp determines that the control should, in fact, process the character, it decodes the parameters of the key-down message and passes the key code and shift-key state to KeyUp in the Key and Shift parameters, respectively.

Either KeyUp or the OnKeyUp event handler it calls can suppress further processing of a key by setting the Key parameter to zero.

See also

OnKeyUp event

DoKeyUp method

KeyDown method

KeyPress method

MainWndProc method

See also

Applies to

TWinControl

Declaration

```
procedure MainWndProc(var Message: TMessage);
```

Description

The MainWndProc method is the main window procedure for a windowed control, meaning it is the routine Windows calls when it has messages for that control. MainWndProc does not process or dispatch the messages itself, but rather calls the WndProc method to do that. MainWndProc provides an exception-handling block around WndProc, ensuring that if any unhandled exceptions occur in the application, they go to the application object's HandleException method.

See also

WndProc method

DefWndProc property

MouseCapture property

See also

Applies to

TControl

Declaration

```
property MouseCapture: Boolean;
```

Description

The MouseCapture property indicates whether the control has "captured" mouse events. When a control captures the mouse, all subsequent mouse events go to that control, known as the capture control, until the user releases the mouse.

A control becomes the capture control when the user drags an item from it. In addition, if the control has the csCaptureMouse flag set in its ControlStyle property, it becomes the capture control when the user presses the left mouse button over it, until the user releases the mouse button.

MouseCapture uses the GetCaptureControl function and the SetCaptureControl procedure in its implementation.

See also

GetCaptureControl function

SetCaptureControl procedure

MouseDown method

See also

Applies to

TControl

Declaration

```
procedure MouseDown(Button: TMouseButton; Shift: TShiftState; X, Y:
  Integer); dynamic;
```

Description

The MouseDown method is the protected implementation method for a control's OnMouseDown event. The MouseDown method inherited from TControl does nothing except call any event handler attached to the OnMouseDown event. You can override MouseDown to provide other responses in addition to the inherited event-handler call.

A control calls MouseDown in response to any of the Windows mouse-down messages (WM_LBUTTONDOWN, WM_MBUTTONDOWN, WM_RBUTTONDOWN), decoding the message parameters into the shift-key state and position, which it passes in the Shift, X, and Y parameters, respectively. The value of the Button parameter depends on which of the Windows messages triggered the event.

See also

[OnMouseDown event]

[MouseMove method]

[MouseUp method]

MouseMove method

See also

Applies to

TControl

Declaration

```
procedure MouseMove(Shift: TShiftState; X, Y: Integer); dynamic;
```

Description

The MouseMove method is the protected implementation method for a control's OnMouseMove event. The MouseMove method inherited from TControl does nothing except call any event handler attached to the OnMouseMove event. You can override MouseMove to provide other responses in addition to the inherited event-handler call.

A control calls MouseMove in response to any of the Windows mouse-move messages (WM_MOUSEMOVE), decoding the message parameters into the shift-key state and position, which it passes in the Shift, X, and Y parameters, respectively.

See also

OnMouseMove event

MouseDown method

MouseUp method

MouseUp method

See also

Applies to

TControl

Declaration

```
procedure MouseUp(Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
  dynamic;
```

Description

The MouseUp method is the protected implementation method for a control's OnMouseUp event. The MouseUp method inherited from TControl does nothing except call any event handler attached to the OnMouseUp event. You can override MouseUp to provide other responses in addition to the inherited event-handler call.

A control calls MouseUp in response to any of the Windows mouse-up messages (WM_LBUTTONUP, WM_MBUTTONUP, WM_RBUTTONUP), decoding the message parameters into the shift-key state and position, which it passes in the Shift, X, and Y parameters, respectively. The value of the Button parameter depends on which of the Windows messages triggered the event.

See also

OnMouseUp event

MouseDown method

MouseMove method

NotifyControls method

See also

Applies to

TWinControl

Declaration

```
procedure NotifyControls(Msg: Word);
```

Description

The NotifyControls method sends the message with the message ID passed in the Msg parameter to all the controls in the windowed control's Controls array property. Delphi uses NotifyControls to update all the controls in a form of such occurrences as changes in the parent color or font.

NotifyControls constructs a generic message record, filling its message ID field with the value of Msg and setting its parameter and result fields to zeros, then calls the Broadcast method to send the message to all the child controls.

See also

Broadcast method

Component messages

Paint method

See also

Applies to

TGraphicControl, TCustomControl, TCustomGrid

Declaration

```
procedure Paint; virtual;
```

Description

The Paint method renders the image of a graphic control or a custom control. Graphic controls and custom controls respond to WM_PAINT messages by initializing the control's canvas and calling Paint.

When you create a graphic control or custom control, you always override Paint to draw the image of the control.

See also

OnPaint event

Canvas property

PaintControls method

See also

Applies to

TWinControl

Declaration

```
procedure PaintControls(DC: HDC; First: TControl);
```

Description

The PaintControls method paints each of the child controls in a windowed control using the device context passed in the DC parameter. The First parameter indicates the starting point in the windowed control's child-control list to paint. If First is **nil** or does not indicate one of the child controls, PaintControls paints all the child controls.

The Repaint method calls PaintControls for the control's parent, passing the control in First to repaint the control and any controls it might intersect. PaintHandler also calls PaintControls after calling PaintWindow.

See also

Repaint method

PaintHandler method

Paint method

PaintHandler method

See also

Applies to

TWinControl

Declaration

```
procedure PaintHandler(var Message: TWMPaint);
```

Description

The PaintHandler method responds to WM_PAINT messages for a windowed control by calling the BeginPaint and EndPaint API functions, and between them paints the control's background by calling PaintWindow and any child controls by calling PaintControls.

See also

PaintControls method

PaintWindow method

PaintWindow method

See also

Applies to

TWinControl, TCustomControl

Declaration

```
procedure PaintWindow(DC: HDC); virtual;
```

Description

The PaintWindow method repaints the contents of a windowed control.

PaintWindow sends a WM_PAINT message to the windowed control's DefaultHandler message, setting the message record's WParam field to the value passed in DC and the other parameter and result fields to zeros.

TCustomControl overrides PaintWindow to use its Canvas property and Paint method, setting the canvas handle to the value passed in DC before calling Paint.

See also
DefaultHandler method

PaletteChanged method

See also

Applies to

TControl, TWinControl

Declaration

```
function PaletteChanged(Foreground: Boolean): Boolean; dynamic;
```

Description

The PaletteChanged method responds to changes in the system's palette by realizing the control's palette as returned from GetPalette. You should never need to alter the default behavior of PaletteChanged. The Foreground parameter indicate whether the palette realization should treat this control's palette as being in the foreground or the background.

The return value from PaletteChanged indicates whether the control actually realized its palette, returning True only if GetPalette returned a valid palette.

See also
GetPalette method

Perform method

See also

Applies to

TControl

Declaration

```
function Perform(Msg, WParam: Word; LParam: Longint): Longint;
```

Description

The Perform method enables a control to send itself a message. Perform bypasses the Windows message queue and sends the message directly to the control's window procedure.

Perform fills a message record (of type TMessage) with the message ID passed in the Msg parameter, the message parameters passed in WParam and LParam, and a result field of zero. Perform then passes the message record to the WndProc method for processing.

See also

WndProc method

Dispatch method

Realign method

See also

Applies to

TWinControl

Declaration

```
procedure Realign;
```

Description

The Realign method forces the windowed control to realign the components within it. If none of the controls that have the windowed control as their parent have Align properties set to values other than alNone, Realign has no effect.

The EnableAlign method calls Realign when its reference count reaches zero.

See also

AlignControls method

DisableAlign method

EnableAlign method

RecreateWnd method

See also

Applies to

TWinControl

Declaration

```
procedure RecreateWnd;
```

Description

The RecreateWnd method destroys the window handle associated with a windowed control and then creates another that reflects the control's current state. Windows cannot reflect certain kinds of changes to a control without destroying the control and recreating it. For example, to reflect a change in the border style of an edit box, you must recreate the control.

RecreateWnd calls DestroyHandle to destroy the window handle, then regenerates the handle as part of redisplaying the control.

See also

DestroyHandle method

CreateWnd method

ReleaseHandle method

See also

Applies to

THintWindow

Declaration

```
procedure ReleaseHandle;
```

Description

The ReleaseHandle method calls the DestroyHandle method to destroy the window handle used by the hint window.

See also
DestroyHandle method

ScaleControls method

See also

Applies to

TWinControl

Declaration

```
procedure ScaleControls(M, D: Integer);
```

Description

The ScaleControls method rescales the child controls of a windowed control without rescaling the control itself. ScaleControls calls the ChangeScale method of each child control, passing through the M and D parameters.

See also
ChangeScale method

SelectFirst method

See also

Applies to

TWinControl

Declaration

```
procedure SelectFirst;
```

Description

The SelectFirst method locates the first selctable control in the windowed control by calling FindNextControl with **nil** as the current control. Makes the first selectable control the active control.

See also

FindNextControl method

SelectNext method

SelectNext method

See also

Applies to

TWinControl

Declaration

```
procedure SelectNext(CurControl: TWinControl; GoForward, CheckTabStop:
  Boolean);
```

Description

The SelectNext method moves the input focus from CurControl to the next control in Z-order. If the FindNextControl method cannot locate an appropriate control (as defined by the GoForward and CheckTabStop parameters), SelectNext restores the focus to CurControl.

See also
FindNextControl method

SetCaptureControl procedure

See also

Unit

Controls

Declaration

```
procedure SetCaptureControl(Control: TControl);
```

Description

The SetCaptureControl procedure makes the control passed in the Control parameter the current capture control, which receives all mouse events until either another control captures the mouse or the mouse is "uncaptured" by passing **nil** to SetCaptureControl.

The MouseCapture property uses SetCaptureControl in its implementation. You can find out what control is currently the capture control by calling the GetCaptureControl function.

See also
MouseCapture property
GetCaptureControl function

SetClientSize method

See also

Applies to

TControl

Declaration

```
procedure SetClientSize(Value: TPoint); virtual;
```

Description

The SetClientSize method sets the width and height of the client area of a control to the values passed in the Value parameter. For a control, the client area is the same as the entire area of the control. For a form, the client area is the area inside the border.

See also

GetClientOrigin method

GetClientRect method

SetParent method

See also

Applies to

TControl

Declaration

```
procedure SetParent(AParent: TWinControl); virtual;
```

Description

The SetParent method is a protected access method for the Parent property. If the control already has a parent, SetParent removes the control from that parent's list of controls by calling the parent's RemoveControl method. If AParent is non-**nil**, SetParent calls that control's InsertControl method to add the control to its list of controls.

The Destroy method of all controls calls SetParent(**nil)** to remove the control from its parent's control list before destorying the component.

See also
Parent property

SetZOrder method

Applies to

TControl

Declaration

```
procedure SetZOrder(TopMost: Boolean); dynamic;
```

Description

The SetZOrder method moves the control in its parents control list to make it either the topmost control (if the TopMost parameter is True) or the bottommost control (if TopMost is False). After changing the order, SetZOrder invalidates the control to ensure repainting to reflect the new order.

ShowControl method

See also

Applies to

TWinControl

Declaration

```
procedure ShowControl(AControl: TControl); virtual;
```

Description

The ShowControl method ensures that the control passed in AControl is actually visible. ShowControl is a protected method used in the implementation of the Show method. ShowControl essentially does whatever is necessary to ensure that the specified control is in view. For example, scroll boxes and forms scroll to ensure that the control is in view, if needed. Notebooks ensure that the page containing the control is in front.

See also
Show method

StringToCursor function

See also

Unit

Controls

Declaration

```
function StringToCursor(S: string): TCursor;
```

Description

The StringToCursor function returns a cursor value based on the string passed in the S parameter. StringToCursor first calls IdentToCursor to convert any known cursor names into their equivalent values. If that fails, StringToCursor attempts to convert S into a number using the Val procedure. If an error occurs, StringToCursor raises an exception indicating either that S is not a number or that the number it represents is outside the valid range of cursors.

See also

IdentToCursor function

CursorToString function

TCMActivate type

See also

Unit

Controls

Declaration

```
type
    TCMActivate = TWMNoParams;
```

Description

The TCMActivate type is the message-record type for the CM_ACTIVATE message.

See also

Component messages

TCMDeactivate type

See also

Unit

Controls

Declaration

```
type
    TCMDeactivate = TWMNoParams;
```

Description

The TCMDeactivate type is the message-record type for the CM_DEACTIVATE message.

TCMGotFocus type

See also

Unit

Controls

Declaration

```
type
    TCMGotFocus = TWMNoParams;
```

Description

The TCMGotFocus type is the message-record type for the CM_GOTFOCUS message.

TCMLostFocus type

See also

Unit

Controls

Declaration

```
type
    TCMLostFocus = TWMNoParams;
```

Description

The TCMLostFocus type is the message-record type for the CM_LOSTFOCUS message.

TCMCancelMode type

See also

Unit

Controls

Declaration

```
type
    TCMCancelMode = TWMNoParams;
```

Description

The TCMCancelMode type is the message-record type for the CM_CANCELMODE message.

TCMDialogKey type

See also

Unit

Controls

Declaration

```
type
    TCMDialogKey = TWMKeyDown;
```

Description

The TCMDialogKey type is the message-record type for the CM_DIALOGKEY message.

TCMDialogChar type

See also

Unit

Controls

Declaration

```
type
    TCMDialogChar = TWMSysChar;
```

Description

The TCMDialogChar type is the message-record type for the CM_DIALOGCHAR message.

TCMHitTest type

See also

Unit

Controls

Declaration

```
type
    TCMHitTest = TWMNCHitTest;
```

Description

The TCMHitTest type is the message-record type for the CM_HITTEST message.

TCMEnter type

See also

Unit

Controls

Declaration

```
type
    TCMEnter = TWMNoParams;
```

Description

The TCMEnter type is the message-record type for the CM_ENTER message.

TCMExit type

See also

Unit

Controls

Declaration

```
type
    TCMExit = TWMNoParams;
```

Description

The TCMExit type is the message-record type for the CM_EXIT message.

TCMDesignHitTest type

See also

Unit

Controls

Declaration

```
type
    TCMDesignHitTest = TWMMouse;
```

Description

The TCMDesignHitTest type is the message-record type for the CM_DESIGNHITTEST message.

TCMFocusChanged type

See also

Unit

Controls

Declaration

```
type
    TCMFocusChanged = record
      Msg: Word
      Unused: Word;
      Sender: TWinControl;
      Result: Longint;
    end;
```

Description

The TCMFocusChanged type is the message-record type for the CM_FOCUSCHANGED message.

TCMWantSpecialKey type

See also

Unit

Controls

Declaration

```
type
    TCMWantSpecialKey = TWMKey;
```

Description

The TCMWantSpecialKey type is the message-record type for the CM_WANTSPECIALKEY message.

TControl component

See also Properties Methods Events Components

Unit

Controls

Description

The TControl component is the abstract component type for all controls. Controls are visual components, meaning the user can see them and manipulate them at run time. TControl provides all the properties, methods, and events that all controls need. Those items required for all controls are declared as **public**, and appear in all controls. Many other items that might be useful to controls are declared as **protected**, meaning that you can publish them in components you derive from TControl or one of its descendant types.

Properties

Caption	Font	ParentFont
Color	IsControl	ParentShowHint
DragCursor	MouseCapture	PopupMenu
DragMode	ParentColor	Text

Methods

CancelModes	GetDeviceContext	ReadState
ChangeScale	GetPalette	SetClientSize
Click	HasParent	SetParent
DblClick	MouseDown	SetName
DefaultHandler	MouseMove	SetZOrder
DefineProperties	MouseUp	UpdateBoundsRect
DragCanceled	Notification	VisibleChanging
GetClientOrigin	PaletteChanged	WndProc
GetClientRect	Perform	

Events

OnClick	OnDragOver	OnMouseMove
OnDblClick	OnEndDrag	OnMouseUp
OnDragDrop	OnMouseDown	

See also
TControlClass type

TControlCanvas object

See also Property Methods

Unit

Controls

Description

The TControlCanvas object is a specialized canvas object associated with a windowed control.

Property

Control

Methods

CreateHandle	Destroy	FreeHandle

See also
TCanvas object

TControlClass type

See also

Unit

Controls

Declaration

```
type
    TControlClass = class of TControl;
```

Description

The TControlClass type is the object-type reference for control components.

See also
TControl component

TControlState type

See also

Unit

Controls

Declaration

```
type
   TControlState = set of (csLButtonDown, csClicked, csPalette,
 csReadingState,
     csAlignmentNeeded, csFocusing, csCreating);
```

Description

The TControlState type defines the set of possible values for the ControlState property. The following table describes the meaning of each flag.

Flag	Meaning
csLButtonDown	The left mouse button was clicked and not yet released. This is set for all mouse-down events.
csClicked	The same as csLButtonDown, but only set if ControlStyle contains csClickEvents, meaning that mouse-down events are interpreted as clicks.
csPalette	The control has a palette that requires realization.
csReadingState	The control is reading its state from a stream.
csAlignmentNeeded	The control needs to realign itself when alignment is reenabled.
csFocusing	The application is processing messages intended to give the control focus. This does not guarantee the control will receive focus, but prevents recursive calls.
csCreating	The control and/or its owner and subcontrols is being created. This flag clears when all have finished creating.

See also
ControlState property

TControlStyle type

See also

Unit

Controls

Declaration

```
type
    TControlStyle = set of (csAcceptsControls, csCaptureMouse,
  csDesignInteractive,
      csClickEvents, csFramed, csSetCaption, csOpaque, csDoubleClicks,
  csFixedWidth,
      csFixedHeight);
```

Description

The TControlStyle type defines the set of possible values for the ControlStyle property. The following table describes the meaning of each flag.

Flag	Meaning
csAcceptsControls	The control becomes the parent of any controls dropped on it at design time.
csCaptureMouse	The control captures mouse events.
csDesignInteractive	The control maps right mouse-button clicks at design time into left-button clicks to manipulate the control.
csClickEvents	The control can receive and respond to mouse clicks.
csFramed	The control has a frame.
csSetCaption	The control should change its caption to match the Name property if the caption has not been explicitly set to something else.
csOpaque	The control hides any items behind it, making it unnecessary to draw them.
csDoubleClicks	The control can receive and respond to double-click messages. Otherwise, map double-clicks into clicks.
csFixedWidth	The width of the control does not vary.
csFixedHeight	The height of the control does not vary.

See also
ControlStyle property

TCreateParams type

See also

Unit

Controls

Declaration

```
type
    TCreateParams = record
      Caption: PChar;
      Style: Longint;
      ExStyle: Longint;
      X, Y: Integer;
      Width, Height: Integer;
      WndParent: HWND;
      Param: Pointer
      WindowClass: TWndClass;
      WinClassName: array[0..63] of Char;
    end;
```

Description

The TCreateParams type is a data structure holding information needed when telling Windows to create a window handle. The fields of a TCreateParams record become the parameters to a call to the CreateWindowEx API function.

Delphi components initialize their creation parameters in the CreateParams method.

See also
CreateParams method

TCustomControl component

See also Property Methods Components

Unit

Controls

Description

The TCustomControl component is a specialized windowed control that makes it easier to draw complex visual images. When you create your own windowed controls, you should derive them from TCustomControl, rather than directly from TWinControl.

Standard windowed controls already "know" how to display themselves, because they are part of Windows. Thus, a button or a check box component descends directly from TWinControl.

When creating an original windowed control, however, you will have to define the control's appearance, so in addition to everything inherited from TWinControl, TCustomControl provides a Canvas property that gives you easy access to the drawing surface and a virtual Paint method, called in response to WM_PAINT messages.

If your custom control does not need to recieve focus, you should derive it from TGraphicControl instead of TCustomControl, as your control will not need the overhead associated with being a windowed control.

Property

<u>Canvas</u>

Methods

Create	Paint	PaintWindow
Destroy		

See also

TWinControl component

TGraphicControl component

TGraphicControl component

See also Property Methods Components

Unit

Controls

Description

The TGraphicControl component is the starting point for nonwindowed custom controls. In addition to everything inherited from the abstract TControl component, TGraphicControl provides a Canvas property that provides ready access to the control's drawing surface and a virtual Paint method called in response to WM_PAINT messages.

If your component needs to receive focus, it will have to be a windowed control, descended from TCustomControl instead of from TGraphicControl.

Property

<u>Canvas</u>

Methods

Create	Destroy	Paint

See also

TCustomControl component

TControl component

THintWindow component

See also Properties Methods Components

Unit

Controls

Description

The THintWindow component is a a simple custom control descended from TCustomControl. A hint window is the small popup window that appears over a control at run time if that control has its ShowHint property set to True.

In addition to everything inherited from TCustomControl, THintWindow provides several properties and methods. The hint window makes public the inherited Canvas, Caption, and Color properties. THintWindow overrides the protected CreateParams and Paint methods and introduces three new methods: ActivateHint, which the application uses to display a hint window, IsHintMsg, which allows the hint window to monitor application messages so it can determine when to disappear, and ReleaseHandle, which destroys the hint window's window handle.

If you want to derive a new kind of hint-window component for your applications, derive a new type from THintWindow, override methods as needed, and then assign the new type to the HintWindowClass variable at application startup.

Properties

Canvas	Caption	Color

Methods

ActivateHint	CreateParams	Paint
Create	IsHintMsg	ReleaseHandle

See also
THintWindowClass type

THintWindowClass type

See also

Unit

Controls

Declaration

```
type
    THintWindowClass = class of THintWindow;
```

Description

The THintWindowClass type is the object-type reference type for the THintWindow component. The HintWindowClass variable, of type THintWindowClass, determines what type of component an application creates to show hints.

See also
HintWindowClass variable
THintWindow component

TWinControl component

See also Properties Methods Events Components

Unit

Controls

Description

The TWinControl component is the abstract component type for windowed controls, which are controls with window handles, including the standard Windows controls.

It is unlikely that you will ever derive a new component directly from TWinControl, however. Instead, you will usually derive from TCustomControl, which provides for a canvas and which handles paint messages, or any of several more specialized abstract controls, such as TButtonControl, TCustomComboBox, TCustomEdit, or TCustomListBox.

In addition to everything inherited from the abstract control type, TControl, TWinControl provides numerous properties, methods, and events. Many of these are familiar, as they appear in most of the standard Delphi controls. However, there are also quite a number of protected methods of interest only to component writers.

Properties

Brush	DefWndProc	Showing
Controls	Handle	TabOrder
ControlCount	HelpContext	TabStop
Ctl3D	ParentCtl3D	WindowHandle

Methods

AlignControls
Broadcast
CanFocus
ChangeScale
ContainsControl
ControlAtPos
Create
CreateHandle
CreateParams
CreateSubClass
CreateWindowHandle
CreateWnd
DefaultHandler
Destroy
DestroyHandle
DestroyWindowHandle
DestroyWnd
DisableAlign
DoEnter
DoExit
DoKeyDown
DoKeyPress
DoKeyUp
EnableAlign
FindNextControl
Focused
GetClientOrigin
GetClientRect
GetDeviceContext
GetTabOrderList
HandleAllocated
HandleNeeded
InsertControl
Invalidate
IsControlMouseMsg
KeyDown
KeyPress
KeyUp
MainWndProc
NotifyControls
PaintControls
PaintHandler
PaintWindow
PaletteChanged
ReadState
Realign
RecreateWnd
RemoveControl
Repaint
ScaleBy
ScaleControls
ScrollBy
SelectFirst
SelectNext
SetBounds
SetFocus
SetZOrder
ShowControl
Update

WndProc
WriteComponents

Events

OnEnter	OnKeyDown	OnKeyUp
OnExit	OnKeyPress	

See also
Visual Component Library Windowed Controls

UpdateBoundsRect method

See also

Applies to

TControl

Declaration

```
procedure UpdateBoundsRect(const R: TRect);
```

Description

The UpdateBoundsRect method changes the Left, Top, Width, and Height properties of the control to match the rectangle passed in the R parameter, but without updating the screen image. UpdateBoundsRect is primarily used internally by controls to stay synchronized with changes made by Windows.

See also
BoundsRect property

VisibleChanging method

See also

Applies to

TControl

Declaration

```
procedure VisibleChanging; dynamic;
```

Description

The VisibleChanging method is a protected method called just before the control changes the value of its Visible property. If, for some reason, the component should not change its visible state, it should raise an exception.

See also
Visible property

WindowHandle property

See also

Applies to

TWinControl

Declaration

```
property WindowHandle: HWnd;
```

Description

The WindowHandle property provides access to the same window handle as the Handle property, but it is protected, and therefore only accessible to code inside the component.

The advantage to WindowHandle is that it is writable, where Handle is read-only. You can therefore assign a new handle to an existing windowed control through WindowHandle. In addition, reading the value of WindowHandle does not automatically create a valid handle. The access method for Handle always calls HandleNeeded to generate a handle if there is not one already. Reading WindowHandle can return a zero value.

See also

CreateWindowHandle method

DestroyWindowHandle method

WndProc method

See also

Applies to

TControl

Declaration

```
procedure WndProc(var Message: TMessage); virtual;
```

Description

The WndProc method provides specific message responses for the control. WndProc is the first method that receives messages for a Delphi control.

The WndProc for TControl defines mouse-message responses for basic clicks and drags, and sends all other messages on to the Dispatch method. TWinControl overrides WndProc to define responses for focus, mouse, and keyboard messages, and sends all others to its inherited WndProc.

If you override WndProc in your own components to provide specialized responses to messages, be sure to call the inherited WndProc at the end to dispatch any other messages.

See also

DefWndProc property

MainWndProc method

Delphi Component Writer’s Help Contents

Click a folder icon to expand or collapse a list of topics for an item. You can also click the underlined text to see a list of topics. To expand or collapse all folders, click the icon in the title.

To get Help on Help, press F1.

Shortcut: Use the Search button to look for specific Help topics.

Delphi Component Writer's Help Contents

Click a folder icon to expand or collapse a list of topics for an item. You can also click the underlined text to see a list of topics. To expand or collapse all folders, click the icon in the title.

To get Help on Help, press F1.

Shortcut: Use the Search button to look for specific Help topics.

Delphi Component Writer’s Help Contents

Click a folder icon to expand or collapse a list of topics for an item. You can also click the underlined text to see a list of topics. To expand or collapse all folders, click the icon in the title.

To get Help on Help, press F1.

Shortcut: Use the Search button to look for specific Help topics.

Delphi Component Writer’s Help Contents

Click a folder icon to expand or collapse a list of topics for an item. You can also click the underlined text to see a list of topics. To expand or collapse all folders, click the icon in the title.

To get Help on Help, press F1.

Shortcut: Use the Search button to look for specific Help topics.

Delphi Component Writer's Help Contents

Click a folder icon to expand or collapse a list of topics for an item. You can also click the underlined text to see a list of topics. To expand or collapse all folders, click the icon in the title.

To get Help on Help, press F1.

Shortcut: Use the **Search** button to look for specific Help topics.

AllEqual method

See also Example

Applies to

TPropertyEditor

Declaration

```
function AllEqual: Boolean; virtual;
```

Description

The AllEqual method returns True if all the like-named properties of a group of selected components have equal values. The Object Inspector calls AllEqual to determine how to display the value of common properties of selected components.

TPropertyEditor.AllEqual returns True only if PropCount is 1, meaning it assumes that property values always differ among selected components. You can override AllEqual in your property editor to return True when it determines that all properties being edited have equal values.

Example

TOrdinalProperty overrides AllEqual to establish that ordinal properties are equal if their GetOrdValue methods all return the same value:

```
function TOrdinalProperty.AllEqual: Boolean;
var
  I: Integer;
  V: Longint;
begin
  Result := False; { start by assuming values differ }
  if PropCount > 1 then   { if there are multiple values... }
  begin
    V := GetOrdValue;      { get first value }
    for I := 1 to PropCount - 1 do        { compare with every other value }
      if GetOrdValueAt(I) <> V then Exit; { if any are different, return False }
  end;
  Result := True;  { if we get here, all were equal }
end;
```

See also
PropCount property

Component property

See also

Applies to

TComponentEditor

Declaration

```
property Component: TComponent;
```

Description

The Component property is a read-only reference to the component currently being edited by the component-editor object. One of the parameters passed to the component editor's Create method establishes the value of Component.

See also
Designer property

Copy method

See also

Applies to

TComponentEditor

Declaration

```
procedure Copy; virtual;
```

Description

The Copy method enables the component-editor object to place additional formats on the Clipboard when the application copies the edited component to the clipboard.

Copying a component to the Clipboard at design time automatically places the filed image (the image as written by the component's WriteState method) on the clipboard, then calls Copy. By default, the Copy method does nothing. You can override Copy in your component-editor types to write additional information to the Clipboard, such as additional formats. Delphi ignores such information, but the information might be usable by another application.

See also
WriteState method

Create method

See also

Applies to

TComponentEditor

Declaration

```
constructor Create(AComponent: TComponent; ADesigner: TFormDesigner);
  virtual;
```

Description

The Create constructor constructs and initializes a component-editor object. Whenever you select a component at design time, Delphi constructs a component-editor object for that component by calling the GetComponentEditor function. The type of the component-editor object depends on the type of editor registered for the particular type of component.

The two parameters passed to Create indicate the selected component and the designer currently in use. These become the values of the Component and Designer properties, respectively.

See also

GetComponentEditor function

RegisterComponentEditor procedure

Designer property

See also

Applies to

TPropertyEditor, TComponentEditor

Declaration

```
property Designer: TFormDesigner;
```

Description

The Designer property provides a read-only reference to the form-designer object containing the property or component being edited. One of the parameters passed to the property-editor or component-editor object's Create method becomes the value of Designer.

See also
TFormDesigner object

Edit method

See also

Applies to

TPropertyEditor, TComponentEditor

Declaration

```
procedure Edit; virtual;
```

Description

The Edit method of a property editor or component editor executes a dialog box that allows the user to edit the entire property or component. The Object Inspector calls Edit when the user double-clicks the value column for a property or clicks the '...' button next to the value. Form designers call Edit when the user double-clicks a selected component on a form being designed.

The Edit method inherited from TPropertyEditor does nothing. You can override Edit to display and read a dialog box to provide a property editor.

The Edit method inherited from TComponentEditor calls the ExecuteVerb method for the editor to execute the first verb in the component editor's verb list.

See also
ExecuteVerb method

EditProperty method

See also

Applies to

TDefaultEditor

Declaration

```
procedure EditProperty(PropertyEditor: TPropertyEditor;
    var Continue, FreeEditor: Boolean); virtual;
```

Description

The EditProperty method chooses which property to edit by default for a component that uses the default editor. TDefaultEditor's Edit method calls EditProperty to choose the property from a list of preferred items.

EditProperty looks for the events OnClick, OnChange, or OnCreate, in that order, and edits the property value for that event (using the TMethodProperty editor). If the component has none of those events, PropertyEdit chooses the first property declared in the component and calls its editor instead.

See also
Edit method

EPropertyError object

See also

Unit

DsgnIntf

Declaration

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

Description

The EPropertyError object indicates an error setting a property. The SetValue methods of property-editor objects raise EPropertyError exceptions when they cannot set a requested property value.

See also
SetValue method

ExecuteVerb method

See also

Applies to

TComponentEditor

Declaration

```
procedure ExecuteVerb(Index: Integer); virtual;
```

Description

The ExecuteVerb method executes the verb specified by the Index parameter from the component editor's list of verbs. ExecuteVerb should perform a valid action for every index value from zero to GetVerbCount - 1. The action for each index value should correspond to the string presented in the context menu by GetVerb.

The usual way to execute a verb is by choosing an item from the component's context menu, but the default property editor calls the first verb in the component's verb list when the user double-clicks the component at design time.

See also

GetVerb method

GetVerbCount method

GetAttributes method

See also

Applies to

TPropertyEditor

Declaration

```
function GetAttributes: TPropertyAttributes; virtual;
```

Description

The GetAttributes method returns information the Object Inspector uses to determine what tools to make available for a particular property when selected. The different values available in a set of type TPropertyAttributes can specify that the Object Inspector should include a drop-down list of values, a list of subproperties, a dialog box for editing the property, and so on.

Most property-editor object types override GetAttributes as needed for their particular kinds of data.

See also
TPropertyAttributes type

GetComponent method

See also

Applies to

TPropertyEditor

Declaration

```
function GetComponent(Index: Integer): TComponent;
```

Description

The GetComponent method returns the component specified by the Index parameter from the property editor's list of components being edited. If the GetAttributes method for the property editor returns a value that includes paMultiSelect, meaning that the editor can edit values for several selected objects simultaneously, the editor holds a list of the components for which it is editing.

Methods such as AllEqual need to access the different components to compare their respective property values.

See also

GetAttributes method

AllEqual method

GetComponentEditor function

See also

Unit

DsgnIntf

Declaration

```
function GetComponentEditor(Component: TComponent;
    Designer: TFormDesigner): TComponentEditor;
```

Description

The GetComponentEditor function searches through the list of component editors registered by the RegisterComponentEditor procedure and constructs and returns a component-editor instance of the appropriate type for Component. If there is no specific component editor registered for Component's type, GetComponentEditor constructs and returns an instance of TDefaultEditor.

The Delphi form designer calls GetComponentEditor for each component the user selects in a form at design time, and destroys the returned component editor when the component becomes deselected. The Component parameter is the component for which an editor is needed, and Designer is the form designer requesting the editor.

See also

RegisterComponentEditor procedure

TDefaultEditor object

GetComponentProperties procedure

See also

Unit

DsgnIntf

Declaration

```
procedure GetComponentProperties(Components: TComponentList; Filter:
  TTypeKinds;
    Designer: TFormDesigner; Proc: TGetPropEditProc);
```

Description

The GetComponentProperties procedure fills the Object Inspector with properties and property editors for the components passed in the Components parameter. If Components contains only one component, all its published properties appear. If Components contains multiple components, the resulting property list shows only the properties all of them share.

The Object Inspector calls GetComponentProperties to display properties, as does the GetProperties method of a property inspector that needs to display subproperties. The TDefaultEditor object also calls GetComponentProperties to generate a list of properties from which it chooses the property to edit by default.

The Filter parameter determines what type or types of properties appear in the list (usually this distinguished between method-pointer properties for events and other types for properties). The Designer parameter indicates the form designer that contains the selected components, and is passed on to the property-editor objects.

See also

GetProperties method

TDefaultEditor object

GetEditLimit method

Applies to

TPropertyEditor, TCustomGrid

Description

There are two methods named GetEditLimit. Each specifies the maximum number of characters a user can type into an edit field, either for a property editor object or the in-place editor for a grid.

GetEditLimit method for property editors

GetEditLimit method for grids

GetEditLimit method

See also

Applies to

TPropertyEditor

Declaration

```
function GetEditLimit: Integer; virtual;
```

Description

The GetEditLimit method restricts the number of characters a user can type for a value in the Object Inspector. By default the limit is 255 characters, but certain property-editor types override GetEditLimit to restrict typed names to shorter lengths.

If you can restrict input length to prevent illegal values, you should override GetEditLimit, rather than accepting overlength strings only to have them rejected by the SetValue method.

See also
SetValue method

GetName method

See also

Applies to

TPropertyEditor

Declaration

```
function GetName: string; virtual;
```

Description

The GetName method returns the name of the property edited by the property editor, with all underline characters converted to spaces. The Object Inspector calls GetName to retrieve the name it displays in the property column.

You should rarely have any reason to override GetName, but you can do so to return another name or to format the name string differently.

See also
GetValue method

GetProperties method

See also Example

Applies to

TPropertyEditor

Declaration

```
procedure GetProperties(Proc: TGetPropEditProc); virtual;
```

Description

The GetProperties method provides the list of subproperties and their property editors when a user requests that an object-type property expand to show its subproperties in the Object Inspector. Calling GetProperties is only valid if GetAttributes returns a set including paSubProperties.

The default GetProperties method does nothing and adds no subproperties. Property editors that need to provide subproperties override GetProperties to call the procedure passed in the Proc parameter to provide the list of subproperties and their editors.

TClassProperty overrides GetProperties to drop down the list of published properties in an object-type property. TSetProperty overrides GetProperties to provide a set-element editor for each possible element in a set-type property.

Example

This is the GetProperties method from TSetProperty:

```
procedure TSetProperty.GetProperties(Proc: TGetPropEditProc);
var
  I: Integer;
begin
  with GetTypeData(GetTypeData(GetPropType)^.CompType)^ do
    for I := MinValue to MaxValue do     { for each possible value... }
      Proc(TSetElementProperty.Create(FDesigner, FPropList, FPropCount, I));   { calll Proc }
end;
```

See also
GetAttributes method

GetPropType method

See also

Applies to

TPropertyEditor

Declaration

```
function GetPropType: PTypeInfo;
```

Description

The GetPropType method returns a type-information record pointer for the property or properties being edited. You should never need to override GetPropType. The record pointed to by the return value is a run-time type information (RTTI) record providing information about the type being edited.

See also
ClassInfo method

GetValue method

See also Example

Applies to

TPropertyEditor

Declaration

```
function GetValue: string; virtual;
```

Description

The GetValue method returns a string representation of the current value of the property being edited. GetValue is the implementation for the **read** part of the Value property.

Example

TIntegerProperty overrides GetValue to convert the numeric value of the property into its string representation:

```
function TIntegerProperty.GetValue: string;
begin
  Result := IntToStr(GetOrdValue);
end;
```

See also
SetValue method
Value property
GetName method

GetValues method

See also Example

Applies to

TPropertyEditor

Declaration

`procedure GetValues(Proc: TGetStrProc); virtual;`

Description

The GetValues method builds the list of values displayed in the property editor. The Proc parameter represents a procedure the method should call for each possible value of the property, passing a string representing a value each time. Calling GetValues is only valid if the set returned by GetAttributes includes paValueList.

Example

A property with three possible values, "Yes," "No," and "Maybe," would override GetValues as follows:

```
procedure TSamplePropertyEditor.GetValues(Proc: TGetStrProc);
begin
  Proc('Yes');
  Proc('No');
  Proc('Maybe');
end;
```

See also
GetAttributes method

GetVerb method

See also

Applies to

TComponentEditor

Declaration

```
function GetVerb(Index: Integer): string; virtual;
```

Description

The GetVerb method returns a string for each valid index value passed in the Index parameter. GetVerb should return a valid string for every index value from zero to GetVerbCount - 1.

The returned string appears in the context menu, and should include ampersand characters (&) and ellipses (...) as appropriate. The ExecuteVerb method performs the actions that correspond to the strings returned by GetVerb.

See also

GetVerbCount method

ExecuteVerb method

GetVerbCount method

See also

Applies to

TComponentEditor

Declaration

```
function GetVerbCount: Integer; virtual;
```

Description

The GetVerbCount method returns the number of valid indexes for the GetVerb and ExecuteVerb methods. Each of those methods should perform a valid action given an index in the range 0..GetVerbCount - 1.

See also

GetVerb method

ExecuteVerb method

Initialize method

See also

Applies to

TPropertyEditor

Declaration

```
procedure Initialize; virtual;
```

Description

The Initialize method performs any needed initialization of the property editor. If your property editor has any time-consuming setups, you should place them in Initialize, rather than in Create.

The Object Inspector calls Initialize after constructing the property-editor object, but before using it. Because the Object Inspector often disposes of property editors without ever using them, avoiding long setup procedures can speed up operation for the user.

As a general rule, then, keep property-editor constructors short and quick, and place any lengthy setup procedures in Initialize.

See also
Create method

PrivateDirectory property

Applies to
TPropertyEditor

Declaration
```
property PrivateDirectory: string;
```

Description
The PrivateDirectory property specifies the directory in which the property editor should store any auxiliary or state files. PrivateDirectory is read-only. By default, PrivateDirectory returns the directory that contains DELPHI.EXE or the working directory specified in DELPHI.INI.

PropCount property

See also

Applies to

TPropertyEditor

Declaration

```
property PropCount: Integer;
```

Description

The PropCount property indicates the number of properties currently being edited by the property editor, reflecting the number passed to the Create constructor of the property editor object. PropCount is greater than one only when the user selects multiple components that all share the same property.

See also
Create method

RegisterComponentEditor procedure

See also Example

Unit

DsgnIntf

Declaration

```
procedure RegisterComponentEditor(ComponentClass: TComponentClass;
    ComponentEditor: TComponentEditorClass);
```

Description

The RegisterComponentEditor procedure adds a component-editor type to the list of registered component editors. When the Delphi form designer calls the GetComponentEditor function to generate an editor for a selected component, GetComponentEditor searches the registered component editor list. If there is no specific editor registered for a given type, GetComponentEditor returns an instance of TDefaultEditor.

To have the form designer use a component editor that you create, you must register that editor by calling RegisterComponentEditor, passing the type of the component you want to edit with the editor, and the type of the editor.

Example

To register the component-editor type TMyComponentEditor as the editor type for the component type TMyComponent,

```
procedure Register;
begin
  RegisterComponentEditor(TMyComponent, TMyComponentEditor);
end;
```

See also
GetComponentEditor function

RegisterPropertyEditor procedure

See also

Unit

DsgnIntf

Declaration

```
procedure RegisterPropertyEditor(PropertyType: PTypeInfo; ComponentClass:
  TClass;
    const PropertyName: string; EditorClass: TPropertyEditorClass);
```

Description

The RegisterPropertyEditor procedure associates a property editor with a type of property, optionally restricted to a particular type of component and a specific property in that component..

See also
TPropertyEditor object

SetValue method

See also

Applies to

TPropertyEditor object

Declaration

```
procedure SetValue(const Value: string); virtual;
```

Description

The SetValue method parses the string passed in Value and converts it as necessary to produce the correct property value. SetValue is the implementation of the **write** part of the Value property.

TIntegerProperty overrides SetValue to convert Value to its numeric equivalent.

See also

GetValue method

Value property

TCaptionProperty object

See also Method Objects

Unit

DsgnIntf

Description

The TCaptionProperty object is the default property editor for Caption and Text properties. The editor is a standard string-property editor, but it has the paAutoUpdate attribute, so it updates the property with each keystroke.

In addition to everything inherited from TStringProperty, TCaptionProperty overrides GetAttributes to allow both multiple selection and automatic updating.

Method

GetAttributes

See also

Caption property

Text property

TStringProperty object

TCharProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TCharProperty object is the property editor for character-type properties. If the property type is a subrange of Char, such as 'A'..'Z', the editor restricts input to characters within the range of the property type.

In addition to everything inherited from TOrdinalProperty, TCharProperty overrides both the GetValue and SetValue methods to display only visible characters and accept both individual characters and ASCII values preceded with the '#' character.

Methods

GetValue	SetValue

See also
TOrdinalProperty object

TClassProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TClassProperty object is the default property editor for all properties which are themselves objects. Users cannot modify object-type properties directly, but the editor displays the name of the object type, and allows editing of the object's properties as subproperties of the property.

In addition to everything inherited from TPropertyEditor, TClassProperty overrides GetAttributes to allow subproperties, GetProperties to display those subproperties, and GetValue to display the name of the object type.

Methods

GetAttributes	GetProperties	GetValue

See also
TPropertyEditor object

TColorProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TColorProperty object is the default property editor for Color properties and other properties of type TColor. The color-property editor displays the current color value as one of the standard clXXX color constants if one exists. Otherwise, the color displays as its hexadecimal value. The editor also provides a drop-down list of all the clXXX color constants.

In addition to everything inherited from TIntegerProperty, TColorProperty overrides the Edit method to use the Windows color-selection dialog box, the GetAttributes method to allow multiple types of editing, GetValue and SetValue to handle all the different ways to type color values, and GetValues to display all the standard color constants.

Methods

Edit	GetValue	SetValue
GetAttributes	GetValues	

See also

TColor type

Color property

TIntegerProperty object

TComponentEditor object

See also Properties Methods Objects

Unit

DsgnIntf

Description

The TComponentEditor object is the basis for all component editors. TComponentEditor itself is abstract, and does not actually edit components. The default component editor is TDefaultEditor, a descendant of TComponentEditor, which implements a working Edit method.

There are three reasons for creating a new component editor, as summarized in the following table.

To do this	Override these methods
Add items to the context menu	GetVerb, GetVerbCount
Change the design-time double-click action	Edit
Paste additional formats to the clipboard	Copy

When you derive a new component editor type, you must register it by calling the RegisterComponentEditor procedure.

When a user selects a component in a form at design time, the form designer creates a component editor based on the type of the component. The editor's Component and Designer properties refer to the component instance and the active form designer, respectively.

- If the user double-clicks the component, the form designer calls the component editor's Edit method.
- If the user invokes the context menu (by right-clicking or pressing Alt+F10), the component editor builds the context menu by calling GetVerbCount and then calling GetVerb for each item. If the user chooses one of the context-menu items, the editor calls ExecuteVerb.
- If the user copies the component to the Clipboard, the component editor first copies a description of the component, then calls the Copy method, which allows the component editor to copy a different format to the Clipboard, which the form designer will ignore when pasted, but which another application might recognize.

Note: Whenever a component editor makes a change to its component, it must then call Designer.Modified, which lets the form designer update its onscreen image to reflect the new status of the component.

Properties

<u>Component</u>	<u>Designer</u>

Methods

Copy	Edit	GetVerb
Create	ExecuteVerb	GetVerbCount

See also
TComponentEditorClass type

TComponentEditorClass type

See also

Unit

DsgnIntf

Declaration

```
type
    TComponentEditorClass = class of TComponentEditor;
```

Description

The TComponentEditorClass type is the object-type reference type for component-editor objects.

See also
TComponentEditor object

TComponentList object

See also Objects

Unit

DsgnIntf

Description

The TComponentList object is list object that provides a component-based interface to the standard list object, TList. TComponentList assumes that its items are all descendants of TComponent. In most other ways, the component-list object works like other list objects.

The GetComponentProperties procedure uses a list of components to hold all the currently-selected components when building property lists for the object inspector. You should never need to create a component-list object, as such objects are only used internally by property editors.

See also

TList object

GetComponentProperties procedure

TComponentProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TComponentProperty object is the default property editor for properties that are components. It does not allow modification of the properties of such component properties, but instead allows the user to specify the name of a compatible component in the same form. The ActiveControl property is an example of a property that uses the TComponentProperty object.

In addition to everything inherited from TPropertyEditor, TComponentProperty overrides the GetAttributes method to allow a sorted drop-down list of valid components, the GetEditLimit method to restrict input to 64 characters, the GetValue method to show the name of the specified component, the GetValues method to construct the drop-down list of compatible components, and SetValue to accept component names.

Methods

GetAttributes	GetValue	SetValue
GetEditLimit	GetValues	

See also
TPropertyEditor object

TCursorProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TCursorProperty object is the default property editor for properties of type TCursor. If the cursor value is one of the standard cursors, the editor displays the value as one of the crXXX cursor values. Otherwise, the value appears as a hexadecimal value.

In addition to everything inherited from TIntegerProperty, TCursorProperty overrides the GetAttributes method to allow a drop-down value list, the GetValues method to generate the list, and the GetValue and SetValue methods to display and accept either numeric values or the predefined cursor constants.

Methods

- GetAttributes
- GetValue
- GetValues
- SetValue

See also

TCursor type

TIntegerProperty object

TDefaultEditor object

See also Methods Objects

Unit

DsgnIntf

Description

The TDefaultEditor is the default component editor in the Delphi form designer. That is, unless you register your own customized component editor for a particular type of component, the form designer creates an instance of TDefaultEditor to edit every component in the form.

TDefaultEditor overrides the Edit method inherited from TComponentEditor to generate an event handler for its associated component's OnCreate, OnChanged, or OnClick event, whichever it finds first. If the component editor overrides the GetVerb method to add items to the context menu, the first added item on the menu becomes the default double-click action.

Methods

Edit	EditProperty

See also
TComponentEditor object

TEnumProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TEnumProperty object is the default property editor for all enumerated types. The enumerated type editor supports a drop-down list of all possible values of the enumeration.

In addition to everything inherited from TOrdinalProperty, TEnumProperty overrides the GetAttributes method to allow a drop-down list of values, the GetValues method to generate the list, and the GetValue and SetValue methods to display and accept the predefined values of the enumerated type.

Methods

GetAttributes	GetValues	SetValue
GetValue		

See also
TOrdinalProperty object

TFloatProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TFloatProperty object is the default property editor for all floating-point numeric properties.

In addition to everything inherited from TPropertyEditor, TFloatProperty overrides the AllEqual method to compare floating-point values, and the GetValue and SetValue methods to accept and display floating-point numbers.

Methods

AllEqual	GetValue	SetValue

See also
TPropertyEditor object

TFontProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TFontProperty object is the default property editor for Font properties. In addition to allowing the editing of the individual attributes of the font object as subproperties, the font-property editor provides access to the standard Windows font-selection dialog box.

In addition to everything inherited from TClassProperty, TFontProperty overrides the GetAttributes method to allow a dialog-box editor, then overrides the Edit method to use the standard Windows font-selection dialog box to edit the entire font property.

Methods

Edit	GetAttributes

See also
TClassProperty object

TFontNameProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TFontNameProperty object is the default property editor for font names. The editor provides a drop-down list of the names of all fonts currently available to Windows.

In addition to everything inherited from TStringProperty, TFontNameProperty overrides the GetAttributes method to allow a drop-down list of values and the GetValues method to generate the list.

Methods

GetAttributes	GetValues

See also
TStringProperty object

TFormDesigner object

See also Objects

Unit

DsgnIntf

Description

The TFormDesigner object is an abstract type from which the form designer in Delphi descends. In addition to the properties and methods inherited from TDesigner, TFormDesigner declares seven abstract methods that deal with event handlers in the Code editor.

Methods

Method	Usage
CreateComponent	Creates the specified component with a specified locationand size, with a specified component as its parent.
CreateMethod	Creates and event handler with the specified name and type and returns the method pointer.
GetComponent	Returns the component that matches the name passed as a parameter. Parses names in forms used by the current form.
GetComponentName	Returns the name of the componsnt passed as its parameter. This is the inverse of GetComponent.
GetComponentNames	Calls a passed method for each component that can be assigned to a property of a specified type. This includes items in forms used by the current form.
GetMethodName	Returns the name of the specified event handler.
GetMethods	Creates a list of methods of the specifiec type.
GetPrivateDirectory	Returns the private directory for the form designer.
IsComponentLinkable	Returns True if the specified component is in a form used by the current form.
MakeComponentLinkable	Adds the specified component's unit to the uses clause of the form being designed.
MethodExists	Returns true if the named event handler already exists.
MethodFromAncestor	Returns True if the specified method was introduced by an ancestor form instead of the current form.
RenameMethod	Renames an existing event handler.
Revert	Asks the form designer to revert the specified property to the value of the corresponding component in the ancestor form.
ShowMethod	Activates the Code editor with the input cursor in the named event handler.

See also
TDesigner object

TGetPropEditProc type

See also

Unit

DsgnIntf

Declaration

```
type
    TGetPropEditProc = procedure(Prop: TPropertyEditor) of object;
```

Description

The TGetPropEditProc type is the method-pointer type used for one of the parameters to the GetProperties method and the GetComponentProperties procedure.

See also

GetProperties method

GetComponentProperties procedure

TIntegerProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TIntegerProperty object is the default property editor for all integer types of all sizes. TIntegerProperty restricts values to the range of the particular integer type being edited.

In addition to everything inherited from TOrdinalProperty, TIntegerProperty overrides the GetValue and SetValue methods to display and accept numeric values.

Methods

GetValue	SetValue

See also
TOrdinalProperty object

TMethodProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TMethodProperty object is the default property editor for properties that are method pointers, most notably events.

In addition to everything inherited from TPropertyEditor, TMethodProperty overrides the AllEqual method to compare method-pointer values, the GetAttributes method to allow a drop-down list of values, the GetValues method to generate that list, the Edit method to generate or edit event handlers, and the GetEditLimit, GetValue, and SetValue methods to handle event-handler names.

Methods

AllEqual	GetEditLimit	GetValues
Edit	GetValue	SetValue
GetAttributes		

See also
TPropertyEditor object

TModalResultProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TModalResultProperty object is the default property editor for the ModalResult property.

In addition to everything inherited from TIntegerProperty, TModalResultProperty overrides the GetAttributes method to allow a drop-down list, the GetValues list to provide that list, and the GetValue and SetValue methods to display and accept either numbers or the predefined modal-result constants.

Methods

GetAttributes	GetValues	SetValue
GetValue		

See also

ModalResult property

TIntegerProperty object

TMPFileNameProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TMPFileNameProperty object is the default property editor for the FileName property in the TMediaPlayer component. The editor opens a file-open dialog box to choose the name of a file for the media player to play.

In addition to everything inherited from TStringProperty, TMPFileNameProperty overrides the GetAttributes method to allow a dialog-box for editing and the Edit method to open a standard Windows file-open dialog box to choose a file name.

Methods

Edit	GetAttributes

See also

FileName property

TMediaPlayer component

TStringProperty object

TOrdinalProperty object

See also Method Objects

Unit

DsgnIntf

The TOrdinalProperty object is the base type for all ordinal-type property editors. In addition to everything inherited from TPropertyEditor, TOrdinalProperty overrides the AllEqual method to compare ordinal values and the GetEditLimit to restric typed values to 64 characters.

Useful descendants of the abstract TOrdinalProperty type include TIntegerProperty, TCharProperty, TEnumProperty, TSetProperty, and TShortCutProperty.

Method

AllEqual	GetEditLimit

See also
TPropertyEditor object

TPropertyAttributes type

See also

Unit

DsgnIntf

Declaration

```
TPropertyAttribute = (paValueList, paSubProperties, paDialog, paMultiSelect,
  paAutoUpdate
    paSortList, paReadOnly, paRevertable);
  TPropertyAttributes = set of TPropertyAttribute;
```

Description

The TPropertyAttributes type designates the variations in the way properties appear in the Object Inspector. The GetAtrributes method of a property-inspector object returns a set of property attributes, indicating how that editor will display its values.

The following table indicates the meaning of each of the possible attributes.

Attribute	Meaning
paValueList	The editor can give a list of enumerated values. The GetValues method builds the list.
paSubProperties	The property is an object that has subproperties that can display. The GetProperties method handles the subproperty lists.
paDialog	The editor can display a dialog box for editing the entire property. The Edit method opens the dialog box.
paMultiSelect	The property should appear when the user selects multiple components. Most property editors allow multiple selection. The notable exception is the editor for the Name property.
paAutoUpdate	Calls the SetValue method after each change in the value, rather than after the entire value is approved.
paSortList	The Object Inspector should sort the list of values alphabetically.
paReadOnly	The property is read-only in the Object Inspector. This is used for sets and fonts, which the user cannot type directly.
paRevertable	Allows the property to revert to its inherited value. Nested properties and elements of sets should not be revertable.

See also
GetAttributes method

TPropertyEditor object

See also Properties Methods Objects

Unit

DsgnIntf

Description

The TPropertyEditor type is the base object type for all property editors. TPropertyEditor itself is abstract, but it defines the properties and methods used or overridden by all useful property editors.

In addition to everything inherited from TObject, TPropertyEditor declares four properties and numerous methods.

More useful types descended from the abstract TPropertyEditor include TOrdinalProperty, TFloatProperty, TStringProperty, TSetElementProperty, TClassProperty, TMethodProperty, and TComponentProperty.

Properties

Designer	PropCount	Value
PrivateDirectory		

Methods

AllEqual	GetEditLimit	GetValue
Destroy	GetName	GetValues
Edit	GetProperties	Initialize
GetAttributes	GetPropType	SetValue
GetComponent		

See also
TPropertyEditorClass type

TPropertyEditorClass type

See also

Unit

DsgnIntf

Declaration

```
type
    TPropertyEditorClass = class of TPropertyEditor;
```

Description

The TPropertyEditorClass type is the object-type reference type for all property-editor object types.

See also
TPropertyEditor object

TSetElementProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TSetElementProperty object is the default property editor for individual elements in sets. The Object Inspector treats each element of a set as a Boolean item, True if the set includes the element, False if the set excludes the element.

In addition to everything inherited from TPropertyEditor, TSetElementProperty overrides the GetName method to display the name of the element instead of that of the property and overrides GetValue and SetValue to use Boolean values to represent the status of the individual element.

Methods

AllEqual	GetName	GetValues
Destroy	GetValue	SetValue
GetAttributes		

See also

TSetProperty object

TPropertyEditor object

TSetProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TSetProperty object is the default property editor for all set-type properties. The set-property editor does not edit the set directly, but rather creates set-element editors for each element in the set.

In addition to everything inherited from TOrdinalProperty, TSetProperty overrides the GetAttributes method to allow expansion of the individual set elements as subproperties, the GetProperties method to return a list of set elements and their editors, and the GetValue method to display the set value in Object Pascal notation.

Methods

GetAttributes	GetProperties	GetValue

See also
TSetElementProperty object

TShortCutProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TShortCutProperty object is the default editor for the ShortCut property. The editor allows the user to either choose a shortcut key from a drop-down list or type in a shortcut key value.

In addition to everything inherited from TOrdinalProperty, TShortCutProperty overrides the GetAttributes method to allow a drop-down list of values, the GetValues method to provide that list, and the GetValue and SetValue methods to display and allow shortcuts either selected from the list or typed by the user.

Methods

GetAttributes	GetValues	SetValue
GetValue		

See also

ShortCut property

TOrdinalProperty object

TStringProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TStringProperty object is the default property editor for all string-type properties.

In addition to everything inherited from TPropertyEditor, TStringProperty overrides the AllEqual method to compare string values, the GetEditLimit method to restrict typed values to the maximum length of the string type, and the GetValue and SetValue methods to display and accept those strings.

Methods

AllEqual	GetValue	SetValue
GetEditLimit		

See also
TPropertyEditor object

TTabOrderProperty object

See also Methods Objects

Unit

DsgnIntf

Description

The TTabOrderProperty object is the default property editor for the TabOrder property. The editor is a standard integer-property editor, but it does not appear when the user selects more than one component, since each component must have a unique tab-order value.

In addition to everything inherited from TIntegerProperty, TTabOrderProperty overrides the GetAttributes method to disallow diaplaying the editor for multiple components.

Method

GetAttributes

See also

TabOrder property

TIntegerProperty object

TTypeInfo type

See also

Unit

TypInfo

Declaration

```
type
    PTypeInfo = ^TTypeInfo;
    TTypeInfo = record
      Kind: TTypeKind;
      Name: string;
      TypeData: TTypeData;
    end;
```

Description

The TTypeInfo type (and its associated pointer type, PTypeInfo) represents run-time type information (RTTI) for a Delphi object.

See also
TTypeKind type

TTypeKind type

See also

Unit

TypInfo

Declaration

```
type
   TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration, tkFloat,
 tkString, tkSet,
     tkClass, tkMethod);
```

Description

The TTypeKind type is an enumerated type specifying the different general kinds of property types Delphi and the Object Inspector need to deal with.

See also

TTypeInfo type

TTypeKinds type

TTypeKinds type

See also

Unit

TypInfo

Declaration

```
type
    TTypeKinds = set of TTypeKind;
```

Description

The TTypeKinds type is a set of kinds of types used when a routine needs to specify one or more kinds of types. The GetComponentProperties procedure, for example, takes a parameter of type TTypeKinds to determine which types of properties it should return.

See also

TTypeKind type

GetComponentProperties procedure

Value property

See also

Applies to

TPropertyEditor

Declaration

```
property Value: string read GetValue write SetValue;
```

Description

The Value property reflects the value of the property being edited. The Object Inspector uses Value to pass the property value back and forth to the property editor. You should never need to alter Value itself, but you will generally override the implementation methods GetValue and SetValue.

See also

GetValue method

SetValue method

Component writer's reference

Components Objects Routines

The Component Writer's Reference is a supplement to the regular VCL reference. It contains

- Information on the protected parts of the standard VCL components and objects
- Other information useful to component writers, including objects and routines for manipulating your components

The underlying assumption of this material is that, as a component writer, you are not just using existing objects (including components). You derive new objects of your own, and as such you need information on parts of the existing objects that are not accessible to users of those objects.

If you need more information about deriving objects and accessing their protected parts, see OOP for component writers.

Apply method

See also

Applies to

TFontDialog

Declaration

```
procedure Apply(Wnd: HWND); dynamic;
```

Description

The Apply method is the protected implementation method for the OnApply event of the font-selection dialog box. The font-selection dialog box calls Apply upon receiving a message from Windows indicating that the user clicked the Apply button in the dialog box. The Wnd parameter specifies the window handle of the parent window of the font-selection dialog box.

By default, Apply does nothing except call any event handler attached to the OnApply event. You can override Apply to provide other responses in addition to the inherited event-handler call.

See also
OnApply event

AutoButtonSet method

See also

Applies to

TMediaPlayer

Declaration

```
procedure AutoButtonSet(Btn: TMPBtnType); virtual;
```

Description

The AutoButtonSet method implements the automatic enabling and dsabling of the media player's buttons when the AutoEnable property is True. The Btn parameter indicates which of the media player's buttons the used pressed.

See also
AutoEnable property

CanChange method

See also

Applies to

TTabSet

Declaration

```
function CanChange(NewIndex: Integer): Boolean; virtual;
```

Description

The CanChange method is the protected implementation method for the OnChange event for tab set components.

See also
OnChange event

CanModify method

Applies to
TCustomRadioGroup

Declaration
```
function CanModify: Boolean; virtual;
```

Description
The CanModify method determines whether the user can choose a different item in the radio group. Whenever a user clicks a radio button in a radio group or presses an equivalent key, the group checks whether CanModify allows that change to occur.

By default, CanModify always returns True, but you can override it to return False when users should not be able to make changes.

Change method

See also

Applies to

TCustomEdit, TCustomComboBox, TScrollBar

Declaration

```
procedure Change; dynamic;
```

Description

The Change method for edit boxes, combo boxes, and scroll bars is the protected implementation method for the OnChange event. For each of these components, the default Change method does nothing except call any event handler attached to the OnChange event. You can override Change to perform any other desired response to changes in these components.

The documentation for the OnChange event describes the occurrences that trigger calls to Change.

See also
OnChange event

CheckCursor method

See also

Applies to

TCustomMaskEdit

Declaration

```
procedure CheckCursor;
```

Description

The CheckCursor method resets the selection or the text cursor after changes to the masked edit box text.

See also
SetCursor method
GetSel method
SetSel method

Click method

See also

Applies to

TMediaPlayer

Declaration

```
procedure Click(Button: TMPBtnType; var DoDefault: Boolean); dynamic;
```

Description

The Click method for media-player components is the protected implementation method for the OnClick event. By default, Click does nothing other than call any event handler attached to the media player's OnClick event. You can override Click in descendant types to customize responses to clicks.

See also
OnClick event
PostClick method

ComboWndProc method

See also

Applies to

TCustomComboBox

Declaration

```
procedure ComboWndProc(var Message: TMessage; ComboWnd: HWnd; ComboProc:
  Pointer); virtual;
```

Description

The ComboWndProc method is the window-procedure method for combo box components. A combo-box component is really two controls: the edit box that holds the selected item and the list of items. Because either of those windows might receive messages that affect the combo box as a whole, they share some common message responses in the ComboWndProc method.

See also
WndProc method

Collapse method

See also

Applies to

TCustomOutline

Declaration

```
procedure Collapse(Index: LongInt); dynamic;
```

Description

The Collapse method is the protected implementation method for an outline component's OnCollapse event. Outline-node objects call their associated outline's Collapse method when they hide their expanded items.

By default, the Collapse method does nothing except call any event handler attached to the OnCollapse event. You can override Collapse to modify the response to collapse events.

See also

OnCollapse event

Expand method

ConvertFields method

See also

Applies to

TFindDialog

Description

The ConvertFields method converts the stored data of the find dialog box component into the data structure used by the Windows common dialog box. This primarily involves converting Object Pascal strings to null-terminated strings.

The find dialog box's Execute method calls ConvertFields before executing its search. Upon returning from a successful search, the component converts the data back to its own form by calling the ConvertFieldsForCallback method.

See also
ConvertFieldsForCallback method

ConvertFieldsForCallback method

See also

Applies to

TFindDialog

Declaration

```
procedure ConvertFieldsForCallback; virtual;
```

Description

The ConvertFieldsForCallback method converts the data structure for a Windows common find dialog box into the form used by the find dialog box component. This primarily involves converting null-terminated strings into Object Pascal strings.

The find dialog box component calls ConvertFieldsForCallback when returning from a successful search operation. Before executing that search, the dialog box had converted its data into the Windows format by calling the ConvertFields method.

See also
ConvertFields method

Create method

Applies to
TObject (all objects), TComponent, TFiler, TComponentEditor

Description
The Create method constructs an object or component.

Create method for all objects
Create method for components
Create method for filer objects
Create method for file-stream objects
Create method for handle-stream objects
Create method for resource-stream objects
Create method for component editors

DirectoryExists function

See also

Unit

FileCtrl

Declaration

```
function DirectoryExists(Name: string): Boolean;
```

Description

The DirectoryExists function returns True if the string passed in the Name parameter represents an existing directory. Otherwise, it returns False.

See also
FileExists function

DispatchCommand method

See also

Applies to

TMenu

Declaration

```
function DispatchCommand(ACommand: Word): Boolean;
```

Description

The DispatchCommand method searches the items in the menu for one with an associated command matching the one passed in ACommand. If such an item exists, DispatchCommand calls the Click method for that item and returns True. If there is no such item, DispatchCommand returns False.

See also
DispatchPopup method

DispatchPopup method

See also

Applies to

TMenu

Declaration

```
function DispatchPopup(AHandle: HMENU): Boolean;
```

Description

The DispatchPopup method searches the items in the menu for one with a menu handle matching the one passed in AHandle. If such an item exists, DispatchPopup calls the Click method for that item and returns True. If there is no such item, DispatchPopup returns False.

See also
DispatchCommand method

DoNotify method

See also

Applies to

TMediaPlayer

Declaration

```
procedure DoNotify; dynamic;
```

Description

The DoNotify method is the protected implementation method for a media-player component's OnNotify event. The media player's MMNotify method calls DoNotify after correcting the enabling and disabling of buttons and setting internal flags from the values passed in the notification message parameters.

By default, DoNotify does nothing except call any event handler attached to the media player's OnNotify event. You can override DoNotify to provide other responses in addition to the inherited event-handler call.

See also

MMNotify method

OnNotify event

DrawButtonFace function

See also

Unit

Buttons

Declaration

```
function DrawButtonFace(Canvas: TCanvas; const Client: TRect; BevelWidth:
  Integer;
    Style: TButtonStyle; IsRounded, IsDown, IsFocused: Boolean): TRect;
```

Description

The DrawButtonFace function draws the beveling and corners and other standard aspects of button appearance in the rectangle specified by Client and returns the remaining portion of the client rectangle as its result.

Speed buttons and bitmap buttons both use DrawButtonFace in their Paint and DrawItem methods, respectively, to create their basic button appearance.

See also
Paint method

DrawItem method

See also

Applies to

TCustomComboBox, TCustomListBox

Declaration

```
procedure DrawItem(Index: Integer; Rect: TRect; State: TOwnerDrawState);
  virtual;
```

Description

The DrawItem method is the protected implementation method for the OnDrawItem events of owner-draw list boxes and combo boxes. The list box or combo box calls DrawItem for each visible item in its list, passing the index of the item in the Index parameter.

By default, the DrawItem method for either a combo box or a list box calls any event handler attached to the component's OnDrawItem event. If there is no handler attached, DrawItem fills the rectangle passed in Rect and draws any text associated with the indexed item. The default drawing ignores the State parameter, although State is passed along to attached event handlers.

See also
OnDrawItem event

DrawTab method

See also

Applies to

TTabSet

Declaration

```
procedure DrawTab(TabCanvas: TCanvas; R: TRect; Index: Integer;
    Selected: Boolean); virtual;
```

Description

The DrawTab method is the protected implementation method for an owner-draw tab-set component's OnDrawTab event. A tab set calls DrawTab as part of its Paint method, once for each visible tab, passing a canvas for the tab area, a rectangle to draw in on that canvas, the index of the tab to draw, and an indication of whether that tab is selected.

By default the DrawTab method does nothing except call any event handler attached to the OnDrawTab event. You can override DrawTab to provide other responses in addition to the inherited event-handler call.

See also
OnDrawTab event

DropDown method

See also

Applies to

TCustomComboBox

Declaration

```
procedure DropDown; dynamic;
```

Description

The DropDown method is the protected implementation method for the OnDropDown method of a combo box. The combo box calls DropDown when it receives a notification from Windows that the combo box needs to drop its list (CBN_DROPDOWN).

By default DropDown does nothing except call any event handler attached to the OnDropDown event of the combo box. You can override DropDown to provide other responses in addition to the inherited event-handler call.

See also
OnDropDown event

EditCanModify method

See also

Applies to

TCustomMaskEdit

Declaration

```
function EditCanModify: Boolean; virtual;
```

Description

The EditCanModify method determines whether the masked edit box is in an editable state. Several other methods, including KeyDown, call EditCanModify to determine whether a keystroke such as Del or Backspace should take effect.

By default, EditCanModify always returns True. You can override EditCanModify to return False if there are conditions under which the masked edit box should be read-only.

See also
KeyDown method

Expand method

See also

Applies to

TCustomOutline

Declaration

```
procedure Expand(Index: LongInt); dynamic;
```

Description

The Expand method is the protected implementation method for an outline component's OnExpand event. Outline-node objects call their associated outline's Expand method when they show their collapsed items.

By default, the Expand method does nothing except call any event handler attached to the OnExpand event. You can override Expand to modify the response to expand events.

See also
OnExpand event
Collapse method

Find method

See also

Applies to

TFindDialog

Declaration

```
procedure Find; dynamic;
```

Description

The Find method is the protected implementation method for the OnFind event of a search dialog box. The find and replace dialog boxes call Find when a text search locates the desired text.

By default Find does nothing except call any event handler attached to the OnFind event. You can override Find to provide other responses in addition to the inherited event-handler call.

See also

OnFind event

Replace method

FormatMaskText function

Unit

Mask

Declaration

```
function FormatMaskText(const EditMask: string; const Value: string):
  string;
```

Description

The FormatMaskText function applies the edit-mask string pased in the EditMask parameter and applies it to the string passed in the Value parameter, returning the resulting formatted string.

Frame3D procedure

See also

Unit

ExtCtrls

Declaration

```
procedure Frame3D(Canvas: TCanvas; var Rect: TRect; TopColor, BottomColor:
  TColor;
    Width: Integer);
```

Description

The Frame3D procedure draws 3-D shading rectangles on the canvas object passed in Canvas, inside the rectangle passed in Rect, using the color specified in TopColor for the top and left sides and that in BottomColor for the right and bottom sides. The Width parameter specifies the thickness of the shading rectangle lines. Frame3D adjusts the Rect rectangle to include only that part of the rectangle inside the shading rectangles when it returns.

Panel components use Frame3D to draw their bevels.

See also
DrawButtonFace function

GetFirstEditChar method

See also

Applies to

TCustomMaskEdit

Declaration

```
function GetFirstEditChar: Integer;
```

Description

The GetFirstEditChar method returns the position of the first editable character in the masked edit control's EditText string (excluding characters added by mask formatting).

See also

GetLastEditChar method

GetNextEditChar method

GetPriorEditChar method

GetLastEditChar method

See also

Applies to

TCustomMaskEdit

Declaration

```
function GetLastEditChar: Integer;
```

Description

The GetLastEditChar method returns the position of the last editable character in the masked edit control's EditText string (excluding characters added by mask formatting).

See also

GetFirstEditChar method

GetNextEditChar method

GetPriorEditChar method

GetMaxChars method

See also

Applies to

TCustomMaskEdit

Declaration

```
function GetMaxChars: Integer;
```

Description

The GetMaxChars function returns the maximu number of characters a user can type into the masked edit box. If the IsMasked property is True, GetMaxChars returns the value of the MaxLength property. Otherwise, GetMaxChars returns the value of the inherited GetTextLen method.

See also
MaxLength property

, GetTextLen method

GetNextEditChar method

See also

Applies to

TCustomMaskEdit

Declaration

```
function GetNextEditChar(Offset: Integer): Integer;
```

Description

The GetNextEditChar method returns the position of the next editable character in the masked edit control's EditText string (excluding characters added by mask formatting) after the character at position Offset.

See also

GetFirstEditChar method

GetLastEditChar method

GetPriorEditChar method

GetNodeDisplayWidth method

See also

Applies to

TCustomOutline

Declaration

```
function GetNodeDisplayWidth(Node: TOutlineNode): Integer;
```

Description

The GetDisplayNodeWidth method returns the width in pixels required to display the specified node. The width includes the item's text in the current font, indentation for the item's level in the outline, and space for any graphics such as lines, plus signs and minus signs, depending on the outline style.

If the outline is an owner-draw outline with a handler attached to the OnDrawItem event, GetNodeDisplayWidth returns zero for all nodes.

See also
SetDisplayWidth method

GetPriorEditChar method

See also

Applies to

TCustomMaskEdit

Declaration

```
function GetPriorEditChar(Offset: Integer): Integer;
```

Description

The GetPriorEditChar method returns the position of the previous editable character in the masked edit control's EditText string (excluding characters added by mask formatting) before the character at position Offset.

See also

GetFirstEditChar method

GetLastEditChar method

GetNextEditChar method

GetSel method

See also

Applies to

TCustomMaskEdit

Declaration

```
procedure GetSel(var SelStart, SelStop: Integer);
```

Description

The GetSel method sets the SelStart and SelStop **var** parameters to the indexes of the first and last selected characters in the masked edit box, respectively.

See also
SetSel method

IsShortCut method

See also

Applies to

TMenu

Declaration

```
function IsShortCut(var Message: TWMKey): Boolean;
```

Description

The IsShortCut method decodes the key message passed in the Message parameter and if the keystroke in that message is the shortcut key for an item in the menu, calls that item's Click method and returns True. If no item in the menu has a shortcut corresponding to the message, IsShortCut returns False.

See also
Click method for menu items

MaskGetFldSeparator function

See also

Unit

Mask

Declaration

```
function MaskGetFldSeparator(const EditMask: string): Integer;
```

Description

The MaskGetFldSeparator function locates the field-separator character in the string passed in EditMask and returns the position of that character in the string. If there is no field-separator character in the string, MaskGetFldSeparator returns -1.

See also
MaskGetMaskBlank function

MaskGetMaskBlank function

See also

Unit

Mask

Declaration

```
function MaskGetMaskBlank(const EditMask: string): Char;
```

Description

The MaskGetMaskBlank function returns the default blank character from the edit-mask string passed in EditMask. The blank character is the last character in the edit-mask string, and must follow a field-separator character.

See also

MaskGetFldSeparator function

MaskGetMaskSave function

See also

Unit

Mask

Declaration

```
function MaskGetMaskSave(const EditMask: string): Boolean;
```

Description

The MaskGetMaskSave function returns True if the edit-mask string passed in EditMask does not contain the MaskNoSave character in its mask-save field. If the mask-save field contains the MaskNoSave character, MaskGetMaskSave returns False.

See also
EditMask property

MeasureItem method

See also

Applies to

TCustomComboBox, TCustomListBox

Declaration

```
procedure MeasureItem(Index: Integer; var Height: Integer); virtual;
```

Description

The MeasureItem method is the protected implementation method for an owner-draw combo box or list box component's OnMeasureItem event. Before drawing its items , a variable owner-draw combo box or list box calls MeasureItem once for each visible item, passing the index of the item to measure, and the default height of the item. Since Height is a **var** parameter, MeasureItem can increase or reduce the height of each item as needed.

By default the MeasureItem method does nothing except call any event handler attached to the OnMeasureItem event. You can override MeasureItem to provide other responses in addition to the inherited event-handler call.

See also

OnMeasureItem event

DrawItem method

MeasureTab method

See also

Applies to

TTabSet

Declaration

```
procedure MeasureTab(Index: Integer; var TabWidth: Integer); virtual;
```

Description

The MeasureTab method is the protected implementation method for an owner-draw tab-set component's OnMeasureTab event. Before drawing its tabs, a tab set calls MeasureTab once for each visible tab, passing the index of the tab to measure, and the default width of the tab. Since TabWidth is a **var** parameter, MeasureTab can increase or reduce the width as needed.

By default the MeasureTab method does nothing except call any event handler attached to the OnMeasureTab event. You can override MeasureTab to provide other responses in addition to the inherited event-handler call.

See also

OnMeasureTab event

DrawTab method

MenuChanged method

See also

Applies to

TMenuItem

Declaration

```
procedure MenuChanged(Rebuild: Boolean); dynamic;
```

Description

The MenuChanged method is the protected implementation method for the OnChange event for menu-item components. The Rebuild parameter indicates whether the change requires rebuilding the menu structure. Menu items call MenuChanged whenever the menu sturcture or content changes, such as when adding, deleting, or renaming items.

The MenuChanged method defined by TMenuItem does nothing except call any event handler attached to the OnChange event. You can override MenuChanged to provide other responses in addition to the inherited event-handler call.

See also
OnChange event

Message method

See also

Applies to

TCommonDialog

Declaration

```
function Message(var Msg: TMessage): Boolean; virtual; export;
```

Description

The Message method is the window-procedure method for common dialog box components. TCommonDialog defines Message to handle Help-request messages if the dialog box's HelpContext property holds a nonzero value. Find and replace dialog boxes override Message to also handle find and replace messages.

See also

Find method

Replace method

MMNotify method

See also

Applies to

TMediaPlayer

Declaration

```
procedure MMNotify(var Message: TMessage); message MM_MCINOTIFY;
```

Description

The MMNotify method responds to a notification message from Windows that a multimedia command completed by updating button states if the AutoEnable property is True, setting internal flags to reflect the outcome of the command, then calling the DoNotify method to trigger the OnNotify event and any other special responses.

See also
DoNotify method

Move method

See also

Applies to

TCustomOutline

Declaration

```
procedure Move(Destination, Source: LongInt; AttachMode: TAttachMode);
```

Description

The Move method moves the outline node at the index position specified in Source to the index passed in Destination, adding, inserting or adding as a child depending on the value of the AttachMode parameter.

See also
MoveTo method

NewItem function

See also

Unit

Menus

Declaration

```
function NewItem(const ACaption: string; AShortCut: TShortCut; AChecked,
  AEnabled: Boolean;
    AOnClick: TNotifyEvent; hCtx: Word; const AName: string): TMenuItem;
```

Description

The NewItem function creates a menu-item component with the attributes specified by the parameters passed and returns the new menu item. By using NewItem and the related functions NewLine, NewMenu, NewPopupMenu, and NewSubMenu, you can create entire menu structures at run time, instead of or in addition to using the Menu designer in Delphi.

See also

NewLine function

NewMenu function

NewPopupMenu function

NewSubMenu function

NewLine function

See also

Unit

Menus

Declaration

```
function NewLine: TMenuItem;
```

Description

The NewLine function creates a menu-item component used as a dividing line between other items and returns that new menu item. By using NewLine and the related functions NewItem, NewMenu, NewPopupMenu, and NewSubMenu, you can create entire menu structures at run time, instead of or in addition to using the Menu designer in Delphi.

See also

NewItem function

NewMenu function

NewPopupMenu function

NewSubMenu function

NewMenu function

See also

Unit

Menus

Declaration

```
function NewMenu(Owner: TComponent; const AName: string;
    Items: array of TMenuItem): TMainMenu;
```

Description

The NewMenu function creates a new main-menu component with the owner specified by the Owner parameter and the name passed in AName and the menu items passed in the Items array parameter and returns the new main menu. By building the Items array with calls to the NewItem, NewLine, and NewSubMenu functions, you can construct an entire main menu at run time.

See also
NewItem function
NewLine function
NewPopupMenu function
NewSubMenu function

NewPopupMenu function

See also

Unit

Menus

Declaration

```
function NewPopupMenu(Owner: TComponent; const AName: string; Alignment:
  TPopupAlignment;
    AutoPopup: Boolean; Items: array of TMenuitem): TPopupMenu;
```

Description

The NewPopupMenu function creates a new popup-menu component with the owner specified by the Owner parameter and the name passed in AName and the menu items passed in the Items array parameter and returns the new popup menu. By building the Items array with calls to the NewItem, NewLine, and NewSubMenu functions, you can construct an entire popup menu at run time.

See also

NewItem function

NewLine function

NewMenu function

NewSubMenu function

NewSubMenu function

See also

Unit

Menus

Declaration

```
function NewSubMenu(const ACaption: string; hCtx: Word; const AName: string;
    Items: array of TMenuItem): TMenuItem;
```

Description

The NewSubMenu function creates a menu-item component with the caption, help context, and name specified by the parameters passed and the subitems specified in the Items array parameter and returns the new menu item. By using NewSubMenu and the related functions NewLine, NewMenu, NewPopupMenu, and NewItem, you can create entire menu structures at run time, instead of or in addition to using the Menu designer in Delphi.

See also

NewItem function

NewLine function

NewMenu function

NewPopupMenu function

PostClick method

See also

Applies to

TMediaPlayer

Declaration

```
procedure PostClick(Button: TMPBtnType); dynamic;
```

Description

The PostClick method is the protected implementation method for the OnPostClick event. By default, PostClick does nothing other than call any event handler attached to the media player's OnPostClick event. You can override PostClick in descendant types to customize responses to clicks.

See also

OnPostClick event

Click method

Replace method

See also

Applies to

TReplaceDialog

Declaration

```
procedure Replace; dynamic;
```

Description

The Replace method is the protected implementation method for the OnReplace event of a find-and-replace dialog box. The replace dialog boxes calls Replace when a text search locates the desired text and replaces it.

By default Replace does nothing except call any event handler attached to the OnReplace event. You can override Replace to provide other responses in addition to the inherited event-handler call.

See also

OnReplace event

Find method

Scroll method

See also

Applies to

TScrollBar

Declaration

```
procedure Scroll(ScrollCode: TScrollCode; var ScrollPos: Integer); dynamic;
```

Description

The Scroll method is the protected implementation method for the OnScroll event in scroll-bar components. TScrollBar calls Scroll in response to scrolling messages from Windows.

The Scroll method defined in TScrollBar does nothing except call any event handler attached to the OnScroll event. You can override Scroll to provide other responses in addition to the inherited event-handler call.

See also
OnScroll event

SetButtonStyle method

Applies to
TButton

Declaration
```
procedure SetButtonStyle(ADefault: Boolean); virtual;
```

Description
The SetButtonStyle method toggles the style of the button control between the default and nondefault styles, depending on whether the button has focus.

SetCursor method

See also

Applies to

TCustomMaskEdit

Declaration

```
procedure SetCursor(Pos: Integer);
```

Description

The SetCursor method sets the text cursor to the position in the masked edit box text specified by the Pos parameter.

See also
CheckCursor method

SetDisplayWidth method

See also

Applies to

TCustomOutline

Declaration

```
procedure SetDisplayWidth(Value: Integer);
```

Description

The SetDisplayWidth method sets the DefaultColWidth property for the outline to the value passed in Value. Since the outline shows only one column, this sets the width of the outline display. The outline calls SetDisplayWidth to set the display width to the size of the widest item in the outline when changing outline styles and scroll bars.

See also
DefaultColWidth property

SetLevel method

See also

Applies to

TCustomOutline

Declaration

```
procedure SetLevel(Node: TOutlineNode; CurLevel, NewLevel: Cardinal);
```

Description

The SetLevel method assigns the specified node from its current level (passed in CurLevel) to a new level (passed in NewLevel, which can be either higher or lower than the current level).

See also
Level property

SetSel method

See also

Applies to

TCustomMaskEdit

Declaration

```
procedure SetSel(SelStart, SelStop: Integer);
```

Description

The SetSel method selects the characters in the masked edit box starting with the character at the SelStart index and ending with the character at SelStop.

See also
GetSel method

Sized method

See also

Applies to

THeader

Declaration

```
procedure Sized(ASection, AWidth: Integer); dynamic;
```

Description

The Sized method is the protected implementation method for the OnSized event in header components. The header component calls Sized when the user completes a resizing operation.

The Sized method defined by THeader does nothing except call any event handler attached to the OnSized event, except at design time, in which case it also calls the Modified method of the form designer that contains the header component. You can override Sized to provide responses in addition to the inherited behavior.

See also

OnSized event

Sizing method

Sizing method

Description

There are two methods called Sizing. Each deals with a control being dragged to resize, but since the controls differ, so do the parameters of their Sizing methods.

Sizing Method for Headers

Sizing Method for Custom Grids

Sizing method

See also

Applies to

THeader

Declaration

```
procedure Sizing(ASection, AWidth: Integer); dynamic;
```

Description

The Sizing method is the protected implementation method for the OnSizing event in header components. The header component calls Sizing for every mouse-move message while the user resizes a section of the header.

The Sizing method defined by THeader does nothing except call any event handler attached to the OnSizing event. You can override Sizing to provide responses in addition to the inherited event-handler call.

See also

OnSizing event

Sized method

TBevel component (protected)

See also Method Components

Unit

ExtCtrls

Description

The TBevel component is a simple descendant of TGraphicControl. In addition to the published parts explained in the VCL reference on using Bevel components, TBevel overrides the protected Paint method to draw its beveled appearance.

Method

<u>Paint</u>

See also
TGraphicControl component
TBevel component VCL reference

TButton component (protected)

See also Methods Components

Unit

StdCtrls

Description

The TButton component is a straightforward descendant of TButtonControl, providing a component interface to the standard Windows push button control.

In addition to everything inherited from TButtonControl and the public and published items explained in the VCL reference for TButton, TButton overrides the protected CreateParams and CreateWnd methods to create the association with the Windows button control and introduces the protected SetButtonStyle method, which handles changing the default and nondefault styles of the button as focus changes among forms.

Methods

CreateParams	CreateWnd	SetButtonStyle

See also

TButtonControl component

TButton component VCL reference

TButtonControl component

See also Methods Components

Unit

StdCtrls

Description

The TButtonControl component is a simple refinement of TWinControl that serves as the basis for all button components, including TButton, check boxes, and radio buttons. TButtonControl overrides one protected method, WndProc, to perform special handling of focus changes for buttons.

Methods

WndProc

See also
TWinControl component

TCommonDialog component

See also Properties Methods Components

Unit

Dialogs

Description

The TCommonDialog component is the abstract component type, derived from TComponent, for all the components that represent Windows common dialog boxes, such as the file-open, find, replace, color, and font-selection dialog boxes. Although you can derive new types from TCommonDialog, it is only designed to handle encapsulations of the Windows common dialog boxes. When you create your own dialog boxes, you will normally start from TForm.

The two properties introduced by TCommonDialog, Ctl3D and HelpContext, are published in all common dialog box components. TCommonDialog also introduces a protected Message method, which serves as the window procedure for the common dialog box.

Properties

Ctl3D	HelpContext

Methods

Create	Message

See also
TComponent component

TCustomCheckBox component

See also Properties Methods Components

Unit

StdCtrls

Description

The TCustomCheckBox component is the abstract base type for all check-box controls, including TCheckBox, TDBCheckBox, and any user-defined check box components.

The properties defined by TCustomCheckBox are all published in the standard Delphi check box components. TCustomCheckBox also overrides three protected methods and introduces a protected Toggle method that switches among the valid check-box states.

Properties

Alignment	Checked	TabStop
AllowGrayed	State	

Methods

Create	CreateWnd	Toggle
CreateParams		

See also
TButtonControl component

TCustomComboBox component

See also Properties Methods Events Components

Unit

StdCtrls

Description

The TCustomComboBox component is the abstract base type for all combo-box controls, including TComboBox, TDBComboBox, and TDriveComboBox.

All the properties declared in TCustomComboBox are published in TComboBox. In addition to the methods inherited from TWinControl, TCustomComboBox overrides several protected methods and introduces four new protected methods. The ComboWndProc method serves as the window procedure for the combo box component. The Change, DropDown, DrawItem, and MeasureItem methods trigger the OnChange, OnDropDown, OnDrawItem, and OnMeasureItem events, respectively.

Properties

DropDownCount	MaxLength	Sorted
ItemHeight	ParentColor	Style

Methods

Change	CreateWnd	DropDown
ComboWndProc	DestroyWnd	MeasureItem
CreateParams	DrawItem	WndProc

Events

OnChange	OnDrawItem	OnMeasureItem
OnDropDown		

See also
TWinControl component

TCustomEdit component

See also Properties Methods Events Components

Unit

StdCtrls

Description

The TCustomEdit component is the abstract base component type for all edit-box and memo components in Delphi, including TEdit, TDBEdit, TMaskEdit, and TMemo.

All the properties declared in TCustomEdit are published in TEdit and explained in the VCL reference for edit-box components. In addition to all the methods it inherits from TWinControl, TCustomEdit overrides two protected methods and introduces a new protected method, Change, which triggers the OnChange event, also introduced.

Properties

AutoSelect	HideSelection	PasswordChar
AutoSize	MaxLength	ParentColor
BorderStyle	OEMConvert	ReadOnly
CharCase		

Methods

Change	CreateParams	CreateWnd

Events

OnChange

See also
TWinControl component

TCustomGroupBox component

See also Methods Components

Unit

StdCtrls

Description

The TCustomGroupBox component is the abstract base component type for all group-box controls in Delphi, including TGroupBox and TRadioGroup.

TCustomGroupBox introduces one new protected method in addition to the methods inherited from TCustomControl. The AlignControls method handles the positioning of owned controls. It also overrides the Paint method.

Methods

AlignControls	Create	Paint

See also
TCustomControl component

TCustomLabel component

See also Properties Methods Components

Unit

StdCtrls

Description

The TCustomLabel component is the abstract base component type for all label components in Delphi, including TLabel and TDBText.

All the properties introduced by TCustomLabel are published in TLabel, and are explained in the VCL reference for TLabel. In addition to the methods it inherits from TGraphicControl, TCustomLabel overrides two protected methods.

Properties

Alignment	FocusControl	Transparent
AutoSize	ShowAccelChar	WordWrap
Canvas		

Methods

Create	Notification	Paint

See also
TGraphicControl component

TCustomListBox component

See also Properties Methods Events Components

Unit

StdCtrls

Description

The TCustomListBox component is the abstract base type for all list-box controls, including TListBox, TDBListBox, and TFileListBox.

All the properties and many of the methods declared in TCustomListBox are published in TListBox. In addition to the methods inherited from TWinControl, TCustomListBox overrides several protected methods and introduces two new protected methods. The DrawItem and MeasureItem methods trigger the OnDrawItem and OnMeasureItem events, respectively.

Properties

BorderStyle	ItemIndex	Selected
Canvas	Items	Sorted
Columns	MultiSelect	Style
ExtendedSelect	ParentColor	TabStop
IntegralHeight	SelCount	TopIndex
ItemHeight		

Methods

Clear	Destroy	ItemAtPos
Create	DestroyWnd	ItemRect
CreateParams	DragCanceled	MeasureItem
CreateWnd	DrawItem	WndProc

Events

OnDrawItem	OnMeasureItem

See also
TWinControl component

TCustomMaskEdit component

See also Properties Methods Components

Unit

Mask

Description

The TCustomMaskEdit component is an abstract component type derived from TCustomEdit that provides the mechanism for creating useful masked edit-box components such as TMaskEdit. Most of the properties and methods declared in TCustomMaskEdit are published by TMaskEdit, and are explained in the VCL reference for masked edit box users.

In addition, there are a number of protected methods you can override in your descendant masked edit boxes.

Properties

EditMask	IsMasked	Text
EditText	MaxLength	

Methods

CheckCursor	GetMaxChars	KeyPress
Clear	GetNextEditChar	KeyUp
Create	GetPriorEditChar	SetCursor
Destroy	GetSel	SetSel
EditCanModify	GetTextLen	ValidateEdit
GetFirstEditChar	KeyDown	ValidateError
GetLastEditChar		

See also
TCustomEdit component

TCustomMemo component

See also Properties Methods Components

Unit

StdCtrls

Description

The TCustomMemo component is the abstract base type for memo components, which are multiline edit boxes, including TMemo and TDBMemo.

In addition to everything it inherits from TCustomEdit, TCustomMemo introduces several new properties that TMemo publishes. They are all explained in the VCL reference for the TMemo component. TCustomMemo also overrides several of its inherited protected methods.

Properties

Alignment	ScrollBars	WantTabs
Lines	WantReturns	WordWrap

Methods

Create	CreateWindowHandle	KeyPress
CreateParams	Destroy	

See also
TCustomEdit component

TCustomOutline component

See also Properties Methods Events Components

Unit

Outline

Description

The TCustomOutline component is a special abstract grid component derived from TCustomGrid. TCustomOutline is the basis for useful grid components such as TOutline. Most of the properties and methods defined in TCustomOutline are protected, so that you can choose whether to publish them in your own outline components. TOutline publishes nearly all of them, so you can find their explanations in the VCL reference for TOutline.

In addition to the items published in TOutline, TCustomOutline declares a number of protected implementation methods you might need to override in your outline components.

Properties

Canvas	Options	PictureOpen
ItemCount	OutlineStyle	PicturePlus
ItemHeight	PictureClosed	Row
Items	PictureLeaf	SelectedItem
ItemSeparator	PictureMinus	Style
Lines		

Methods

Add	DrawCell	KeyDown
AddChild	EndUpdate	KeyPress
AddChildObject	Expand	Loaded
AddObject	FullCollapse	LoadFromFile
BeginUpdate	FullExpand	LoadFromStream
Clear	GetDataItem	Move
Collapse	GetItem	SaveToFile
Create	GetNodeDisplayWidth	SaveToStream
DblClick	GetTextItem	SetDisplayWidth
DefineProperties	Insert	SetLevel
Delete	InsertObject	SetUpdateState
Destroy		

Events

OnCollapse	OnExpand	OnDrawItem

See also
TCustomGrid component

TCustomPanel component

See also Properties Methods Events Components

Unit

ExtCtrls

Description

The TCustomPanel component is the abstract base component type for all panel components in Delphi, including TPanel.

In addition to everything inherited from TCustomControl, TCustomPanel introduces a number of properties, all of which TPanel publishes, so you can find them explained in the VCL reference for TPanel. TCustomPanel also overrides a number of inherited protected methods and introduces one new protected method, Resize, which triggers the OnResize event, also introduced.

Properties

Alignment	BevelWidth	Color
BevelInner	BorderWidth	Locked
BevelOuter	BorderStyle	ParentColor

Methods

AlignControls	CreateParams	Paint
Create	DefineProperties	Resize

Events

OnResize

See also
TCustomControl component

TCustomRadioGroup component

See also Properties Methods Components

Unit

ExtCtrls

Description

The TCustomRadioGroup component is a specialized descendant of TCustomGroupBox that serves as the base component type for radio-group components.

The properties introduced by TCustomRadioGroup are all published in TRadioGroup, so you can find explanations for them in the VCL reference for TRadioGroup. TCustomRadioGroup overrides several inherited protected methods and introduces one new protected method, CanModify.

Properties

Columns	ItemIndex	Items

Methods

CanModify	Destroy	ReadState
Create		

See also
TCustomGroupBox component

TFindDialog component (protected)

See also Methods Components

Unit

Dialogs

Description

The TFindDialog component is a descendant of TCommonDialog. In addition to everything it inherits, TFindDialog overrides the Message method and introduces three new protected methods: Find, which triggers the OnFind event, and ConvertFields and ConvertFieldsForCallback, which convert data types for communication with the Windows common dialog box.

Methods

ConvertFields	Find	Message
ConvertFieldsForCallback		

See also

TCommonDialog component

TFindDialog component VCL reference

TFontDialog component (protected)

See also Method Components

Unit

Dialogs

Description

The TFontDialog component is a descendant of TCommonDialog. In addition to everything it inherits and the published items explained in the VCL reference for TFontDialog, TFontDialog introduces one new protected method called Apply, which applies the current font dialog box values.

Method

<u>Apply</u>

See also

TCommonDialog component

TFontDialog component VCL reference

THeader component (protected)

See also Methods Components

Unit

ExtCtrls

Description

The THeader component is a descendant of TCustomControl. In addition to everything it inherits and the published parts explained in the VCL reference for THeader, it overrides two protected methods and introduces the protected Sized and Sizing methods, which trigger the OnSized and OnSizing events, respectively.

Methods

CreateParams	Sized	Sizing
Paint		

See also

TCustomControl component

THeader component VCL reference

TImage component (protected)

See also   Methods   Components

Unit

ExtCtrls

Description

The TImage component is a descendant of TGraphicControl. In addition to everything it inherits and the published items explained in the VCL reference for TImage, TImage also overrides the GetPalette and Paint methods.

Methods

GetPalette	Paint

See also

TGraphicControl component

TImage component VCL reference

Timer method

See also

Unit

ExtCtrls

Applies to

TTimer

Declaration

```
procedure Timer; dynamic;
```

Description

The Timer method is the protected implementation method for the OnTimer event in Timer components. The Timer component calls Timer every time it receives a timer message (WM_TIMER) from Windows.

The Timer method defined by TTimer does nothing except call any event handler attached to the OnTimer event. You can override Timer to provide other responses in addition to the inherited event-handler call.

See also
OnTimer event

TMediaPlayer component (protected)

See also Methods Components

Unit

MPlayer

Description

The TMediaPlayer component is a descendant of TCustomControl. In addition to eveything it inherits and the published items explained in the VCL reference for TMediaPlayer, it overrides the Paint method to display the media-player buttons and overrides the Loaded method to implement the AutoOpen property. TMediaPlayer also introduces several new protected methods for handling clicks and Windows notifications.

Methods

AutoButtonSet	Loaded	Paint
Click	MMNotify	PostClick
DoNotify		

See also

TCustomControl component

TMediaPlayer component VCL reference

TMemo component (protected)

See also Methods Components

Unit

StdCtrls

Description

The TMemo component inherits everything from TCustomMemo, and overrides one protected method, DefineProperties. Everything else in TMemo is public or published and explained in the VCL reference for the TMemo component.

Methods

DefineProperties

See also

TCustomMemo component

TMemo component VCL reference

TMenu component

See also Properties Methods Components

Unit

Menus

Description

The TMenu component is the abstract base type for usable menu components such as TMainMenu and TPopupMenu. In addition to the items it inherits from its ancestor, TComponent, TMenu introduces three properties and several methods.

Properties

Handle	Items	WindowHandle

Methods

Create	DispatchPopup	IsShortCut
Destroy	FindItem	WriteComponents
DispatchCommand	GetHelpContext	

See also
TComponent component

TMenuItem component (protected)

See also Methods Components

Unit

Menus

Description

The TMenuItem component represents an item in either a main menu or a popup menu. Most parts of TMenuItem are either public or published, and therefore explained in the VCL reference for users of menu items. TMenuItem declares one additional protected method, MenuChanged, which implements the OnChange event, and overrides four others: DefineProperties, HasParent, ReadState, and WriteComponents.

Methods

DefineProperties	MenuChanged	WriteComponents
HasParent	ReadState	

See also

TComponent component

TMenuItem component VCL reference

TNotebook component (protected)

See also Methods Components

Unit

ExtCtrls

Description

The TNotebook component is a custom control. In additin to everything it inherits and the published items explained in the VCL reference for TNotebook, it overrides several inherited protected methods.

Methods

CreateParams	ShowControl	WriteComponents
ReadState		

See also

TCustomControl component

TNotebook component VCL reference

Toggle method

See also

Applies to

TCustomCheckBox

Declaration

```
procedure Toggle; virtual;
```

Description

The Toggle method toggles among the different values of the State property for the check box component. Check-box components call Toggle when clicked.

The following table shows how the Toggle method defined by TCustomCheckBox sets the value of state based on its previous value.

State before Toggle	State after Toggle
csChecked	csUnchecked
csGrayed	csChecked
csUnchecked	csChecked, unless AllowGrayed is True, in which case csGrayed

You can override Toggle to change the order of the toggle states in your check box components.

See also
State property

TRadioButton component (protected)

See also Methods Components

Unit

StdCtrls

Description

The TRadioButton component is a button control. In addition to everything it inherits and the published items explained in the VCL reference for TRadioButton, it overrides two inherited protected methods, CreateParams and CreateWnd.

Methods

CreateParams	CreateWnd

See also

TButtonControl component

TRadioButton component VCL reference

TReplaceDialog component (protected)

See also Methods Components

Unit

Dialogs

Description

The TReplaceDialog component is a customized version of TFindDialog that both finds and replaces text. In addition to everything it inherits and the published items explained in the VCL reference for TReplaceDialog, it overrides the protected ConvertFields and ConvertFieldsForCallback methods and introduces a protected Replace method that triggers the OnReplace event.

Methods

ConvertFields	ConvertFieldsForCallback	Replace

See also

TFindDialog component (protected)

TReplaceDialog component VCL reference

TShape component (protected)

See also Method Components

Unit

ExtCtrls

Description

The TShape component is a simple descendant of TGraphicControl. In addition to the published parts explained in the VCL reference on using Shape components, TShape overrides the protected Paint method to draw its geometric shapes.

Method

Paint

See also

TGraphicControl component

TShape component VCL reference

TPaintBox component (protected)

See also Method Components

Unit

ExtCtrls

Description

The TPaintBox component is a simple descendant of TGraphicControl. In addition to the published parts explained in the VCL reference on using Paint Box components, TPaintBox overrides the protected Paint method to trigger its OnPaint event.

Method

Paint

See also

TGraphicControl component

TPaintBox component VCL reference

TScrollBar component (protected)

See also   Methods   Components

Unit

StdCtrls

Description

The TScrollBar object is a windowed control. In addition to everything it inherits and the published items explained in the VCL reference for TScrollBar, it overides two protected methods and introduces two more, Change and Scroll, which trigger the OnChange and OnScroll methods, respectively.

Methods

Change	CreateWnd	Scroll
CreateParams		

See also

TWinControl component

TScrollBar component VCL reference

TTabSet component (protected)

See also Methods Components

Unit

Tabs

Description

The TTabSet component is a custom control. In addition to everything it inherits and the published items explained in the VCL reference for TTabSet, it overrides the protected DefineProperties and Paint methods, and introduces three more protected methods. DrawTab and MeasureTab trigger the OnDrawTab and OnMeasureTab events, respectively. CanChange determines whether the control can change selected tabs, and triggers the OnChange event if it can.

Methods

CanChange	DrawTab	Paint
DefineProperties	MeasureTab	

See also

TCustomControl component

TTabSet component VCL reference

TTimer component (protected)

See also Methods Components

Unit

ExtCtrls

Description

The TTimer component is a simple nonvisual component derived directly from TComponent. In addition to the public and published items explained in the VCL reference for using timer components, TTimer declares a single protected method called Timer, which triggers the OnTimer event.

Methods

Timer

See also

TComponent component

TTimer component VCL reference

ValidateError method

See also

Applies to

TCustomMaskEdit

Declaration

```
procedure ValidateError; virtual;
```

Description

The ValidateError method is called by the ValidateEdit method if an error occurs when validating the masked edit box text. By default, ValidateError beeps and then raises an EDBEditError exception. You can override ValidateError to provide your own error handling.

See also
ValidateEdit method

Components

The following are the components documented in the Delphi Component Writer's Reference. Many of them are also covered in the regular VCL reference. In those cases, the entries for component writers show only the protected parts of the components. For components that do not appear in the documentation for component users, all parts of the component are shown.

TApplication	TCustomMaskEdit	TMenu
TBevel	TCustomMemo	TMenuItem
TButton	TCustomOutline	TNotebook
TButtonControl	TCustomPanel	TPaintBox
TCommonDialog	TCustomRadioGroup	TRadioButton
TComponent	TFindDialog	TReplaceDialog
TControl	TFontDialog	TScrollBar
TCustomCheckBox	TForm	TScrollBox
TCustomComboBox	TGraphicControl	TScrollingWinControl
TCustomControl	THeader	TShape
TCustomEdit	THintWindow	TTabSet
TCustomGrid	TImage	TTimer
TCustomGroupBox	TInplaceEdit	TWinControl
TCustomLabel	TMediaPlayer	
TCustomListBox	TMemo	

Objects

Components Routines VCL Objects

The following are the objects documented in the Delphi Component Writer's Reference:

TBitmap	TFiler	TMPFileNameProperty
TCanvas	TFileStream	TObject
TCaptionProperty	TFloatProperty	TOrdinalProperty
TCharProperty	TFontProperty	TPersistent
TClassProperty	TFontNameProperty	TPicture
TColorProperty	TFormDesigner	TPropertyEditor
TComponentEditor	TGraphic	TReader
TComponentList	THandleStream	TResourceStream
TComponentProperty	TIcon	TSetElementProperty
TControlCanvas	TIntegerProperty	TSetProperty
TCursorProperty	TList	TShortCutProperty
TCustomMemoryStream	TMemoryStream	TStream
TDefaultEditor	TMetafile	TStringProperty
TDesigner	TMethodProperty	TTabOrderProperty
TEnumProperty	TModalResultProperty	TWriter

Routines

Components Objects

The following procedures and function are of interest primarily to component writers.

DrawButtonFace	GetFixupRoots	RegisterComponentEditor
FindDragTarget	IdentToCursor	RegisterComponents
FormatMaskText	InitComponentRes	RegisterPropertyEditor
GetComponentEditor	MaskGetFldSeparator	SetCaptureControl
GetComponentProperties	MaskGetMaskBlank	StringToCursor
GetCursorValues	MaskGetMaskSave	

Ancestor property

See also

Applies to

TFiler

Declaration

```
property Ancestor: TPersistent;
```

Description

The Ancestor property is used when writing components in inherited forms. Since the writer object only needs to write the values of properties that differ from those inherited, it tracks each inherited component in Ancestor and compares properties before writing.

If Ancestor is **nil**, there is no corresponding inherited component, and the writer object should write the component out to the stream completely. Ancestor is always **nil**, except during calls to WriteDescendant and WriteDescendantRes.

When writing or overriding a DefineProperties method, you need to be aware that Ancestor might be set, and therefore you might need to write or not write properties, as appropriate.

See also

RootAncestor property

DefineProperties method

AssignTo method

See also

Applies to

TPersistent

Declaration

```
procedure AssignTo(Dest: TPersistent); virtual;
```

Description

The AssignTo method provides a generic mechanism for assigning the data associated with an object to another. AssignTo is the protected implementation of the public Assign method. By default, AssignTo raises an EConvertError exception. The Clipboard object overrides AssignTo to permit different kinds of graphic objects to copy each others' data from the clipboard.

TClipboard.AssignTo copies the contents of the Clipboard to the object specified by Dest. Dest must be a picture, a bitmap, or a metafile. Passing any other type of object causes a conversion error.

See also
Assign method

BeginReferences method

See also Example

Applies to

TReader

Declaration

```
procedure BeginReferences;
```

Description

The BeginReferences method starts a block of commands that read components that might contain references to each other. You'll always use BeginReferences together with FixupReferences and EndReferences in a **try..finally** block.

Following the call to BeginReferences, the reader object creates a list of all the objects read and their names. After all the interdependent objects are read, call FixupReferences to resolve the named references from the stream into instance references. Finally, call EndReferences to dispose of the fixup list.

Example

A block that reads components that reference each other always takes this form:

```
BeginReferences;   { create the fixup list }
try
  { read all the components, adding their names to the fixup list}
  ...
  FixupReferences; { resolve names into pointers }
finally
  EndReferences;   { dispose of the fixup list }
end;
```

See also

EndReferences method

FixupReferences method

Capacity property

See also

Applies to

TMemoryStream

Declaration

```
property Capacity: Longint;
```

Description

The Capacity property determines the size of the memory pool allocated to the memory stream, as opposed to Size, which is the size of the data in the stream. You can set Capacity to a large initial value if you know you'll be adding large amounts of data in small increments, to prevent frequent reallocations.

See also

Size property

Realloc method

ChangeName method

See also

Applies to

TComponent (all components)

Declaration

```
procedure ChangeName(const NewName: TComponentName);
```

Description

The ChangeName method sets the private, internal storage for the Name property to the string passed in NewName. The SetName method uses ChangeName to actually change the component's name. If you override SetName, you must either call the inherited SetName, call ChangeName directly, or provide your own internal storage for the Name property.

See also

Name property

SetName methodFor all components

Clear method

See also

Applies to

TMemoryStream

Declaration

```
procedure Clear;
```

Description

The Clear method disposes of the memory pool assigned to the memory stream and sets the Memory property to **nil**. After a call to Clear, both the Size and Position properties of the memory stream are 0.

See also
Memory property

ComponentState property

See also

Applies to

TComponent (all components)

Declaration

```
property ComponentState: TComponentState;
```

Description

The ComponentState property is a read-only property that describes the current state of the component. The individual component-state flags are explained under the TComponentState type.

Components use the ComponentState property to detect states in which they need to avoid certain kinds of actions. For example, if a component needs to avoid certain behaviors at design time that it will perform at run time, it can check for the csDesigning flag.

See also
TComponentState type

ComponentStyle property

See also

Applies to

TComponent (all components)

Declaration

```
property ComponentStyle: TComponentStyle;
```

Description

The ComponentStyle property is a set of possible styles that govern the behavior of the component.

The csInheritable style, if present, indicates that the component can be inherited by a descendant form type. If any of the components in a form do not have the csInheritable style, the form cannot be used as the ancestor of an inherited form. TComponent sets this style by default, and most of the standard components are inheritable.

The csCheckPropAvail style indicates that a component needs to check its properties for readability. This is only used for OLE controls, where the Object Inspector cannot tell directly that a property is readable, and therefore displayable. Native Delphi components should not use this style, as the readability test is very time-consuming.

See also
TComponentStyle type

CopyFrom method

Applies to
TStream

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

Description
The CopyFrom method copies Count bytes from the stream specified by Source into the stream, moves the current position by Count bytes, and returns the number of bytes copied.

Create method

See also

Applies to

TFiler

Declaration

```
constructor Create(Stream: TStream; BufSize: Cardinal);
```

Description

The Create method constructs a new filer object and associates it with the stream passed in the Stream parameter, with a buffer of size BufSize.

See also
Create method

Create method

See also

Applies to

TComponent

Declaration

```
constructor Create(AOwner: TComponent); virtual;
```

Description

The Create method constructs and initializes a new component object and inserts the newly-constructed component into its owner, as specified by the AOwner parameter, by calling that owner's InsertComponent method.

Nearly every kind of component overrides Create to initialize its unique properties. When you override Create in your components, you must always call the inherited Create method first, then proceed with your component's initialization.

If your component's Create method allocates resources or memory, you should also override the Destroy method to free those resources.

See also

Create Method for All Objects

Destroy method

Create method

See also

Applies to

TFileStream

Declaration

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

Description

The Create method constructs a new file-stream object associated with the file specified in FileName. Create opens the associated file using the file mode specified by Mode. Mode must be one of the Delphi file-mode constants.

See also
Create method

Create method

See also

Applies to

THandleStream

Declaration

```
constructor Create(AHandle: Integer);
```

Description

The Create method constructs a new handle-stream object associated with the file handle passed in AHandle. AHandle becomes the value of the stream's Handle property.

See also

Create method

Handle property

Create method

See also

Applies to

TResourceStream

Declaration

```
constructor Create(Instance: THandle; const ResName: string; ResType:
  PChar);
```

Description

The Create method for resource streams constructs a memory stream containing the resource named by ResName, of the type indicated by ResType, from the instance passed in Instance.

See also
CreateFromID method

CreateFromID method

See also

Applies to

TResourceStream

Declaration

```
constructor CreateFromID(Instance: THandle; ResID: Integer; ResType: PChar);
```

Description

The CreateFromID method constructs a memory stream on the resource in the Instance passed in Instance, of the type passed in ResType, with the resource ID matching ResID.

See also
Create method for resource streams

DefineBinaryProperty method

See also

Applies to

TFiler, TReader, TWriter

Declaration

```
procedure DefineBinaryProperty(const Name: string; ReadData, WriteData:
  TStreamProc;
    HasData: Boolean); virtual; abstract;
```

Description

The DefineBinaryProperty method defines binary data the filer object will store as if the data were a property. The Name parameter specifies the name of this "fake" property. The ReadData and WriteData parameters specify the methods in the storing object that read and write the desired data, respectively. The HasData parameter determines at run time whether the "fake" property has data to store.

DefineBinaryProperty and its nonbinary counterpart, DefineProperty, should only be called in the DefineProperties method of an object that has data it needs to store. DefineProperties has a filer object as its parameter, and it is that filer object's DefineProperty and DefineBinaryProperty methods you call.

The difference between DefineBinaryProperty and DefineProperty is that the binary property is written directly a stream object, rather than going through a filer object. The methods passed in ReadData and WriteData read or write a binary representation of the object's data directly to the stream passed to them.

Defined binary properties are quite rare. The only standard VCL objects that define binary properties are the custom outline (for storing its internal bitmaps) and the graphic and pricture objects.

See also

DefineProperty method

DefineProperties method

DefineProperties method

See also Examples

Applies to

TPersistent

Declaration

```
procedure DefineProperties(Filer: TFiler); virtual;
```

Description

The DefineProperties method designates methods for storing an object's unpublished data on a stream such as a form file. By default, writing an object to a stream writes the values of all its published properties, and reading the object in reads those values and assigns them to the properties. Objects can also specify methods that read and write data other than properties by overriding the DefineProperties method.

When you override DefineProperties, you include some or all of the following:

- A call to the inherited method
- Calls to the filer object's DefineProperty method
- Calls to the filer object's DefineBinaryProperty method

The default DefineProperties method defined by TPersistent does nothing, but many other objects and components override the method to store otherwise unpublished data.

Examples

One of the simplest examples of a DefineProperties method is that for TComponent. Although TComponent doesn't publish or even declare Left and Top properties, but it does store the information on where the user positions the component. Nonvisual components derived directly from TComponent can therefore save their positions because TComponent overrides DefineProperties to save the component's position:

```
procedure TComponent.DefineProperties(Filer: TFiler);
begin
  Filer.DefineProperty('Left', ReadLeft, WriteLeft, LongRec(FDesignInfo).Lo <> 0);
  Filer.DefineProperty('Top', ReadTop, WriteTop, LongRec(FDesignInfo).Hi <> 0);
end;
```

TControl, on the other hand, publishes real Left and Top properties, so when it overrides DefineProperties, it does not call its inherited method, which would define the "fake" Left and Top:

```
procedure TControl.DefineProperties(Filer: TFiler);
begin
  { Ignore inherited defined properties since they are redefined with real properties }
  Filer.DefineProperty('IsControl', ReadIsControl, WriteIsControl, FIsControl);
end;
```

See also

DefineProperty method

DefineBinaryProperty method

DefineProperty method

See also Example

Applies to

TFiler, TReader, TWriter

Declaration

```
procedure DefineProperty(const Name: string; ReadData: TReaderProc;
  WriteData: TWriterProc;
    HasData: Boolean); virtual; abstract;
```

Description

The DefineProperty method defines data the filer object will store as if the data were a property. The Name parameter specifies the name of this "fake" property. The ReadData and WriteData parameters specify the methods in the storing object that read and write the desired data, respectively. The HasData parameter determines at run time whether the "fake" property has data to store.

DefineProperty and its binary counterpart, DefineBinaryProperty, should only be called in the DefineProperties method of an object that has data it needs to store. DefineProperties has a filer object as its parameter, and it is that filer object's DefineProperty and DefineBinaryProperty methods you call.

Note that when defining properties, a writer object should be aware of the Ancestor property, which if non-**nil** indicates that the writer should only write the values of properties that differ from those inherited from Ancestor.

Example

One of the simplest examples of a DefineProperties method is that for TComponent. Although TComponent doesn't publish or even declare Left and Top properties, but it does store the information on where the user positions the component. Nonvisual components derived directly from TComponent can therefore save their positions because TComponent overrides DefineProperties to save the component's position:

```
procedure TComponent.DefineProperties(Filer: TFiler);
begin
  Filer.DefineProperty('Left', ReadLeft, WriteLeft, LongRec(FDesignInfo).Lo <> 0);
  Filer.DefineProperty('Top', ReadTop, WriteTop, LongRec(FDesignInfo).Hi <> 0);
end;
```

See also
DefineBinaryProperty method
DefineProperties method
Ancestor property

DesignInfo property

Applies to

TComponent (all components)

Declaration

```
property DesignInfo: Longint;
```

Description

The DesignInfo property contains information used by the form designer. You should never need to access DesignInfo.

DestroyComponents method

See also

Applies to

TComponent (all components)

Declaration

```
procedure DestroyComponents;
```

Description

The DestroyComponents method iterates through the components owned by the component, removing each from the list of owned components and destroying it. The Destroy method of TComponent calls DestroyComponents after calling Destroying and before disposing of itself.

See also

Destroy method

Destroying method

Destroying method

See also

Applies to

TComponent (all components)

Declaration

```
procedure Destroying;
```

Description

The Destroying method sets the csDestroying flag in the ComponentState property, then calls the Destroying method for each owned component so that their csDestroying flags will also be set. If csDestroying is already set, Destroying does nothing.

The Destroy method of TComponent calls Destroying before actually destroying anything. Other components, such as TWinControl, also call Destroying directly to notify owned components of their impending destruction.

See also

Destroy method

ComponentState property

EndOfList method

See also Example

Applies to

TReader

Declaration

```
function EndOfList: Boolean;
```

Description

The EndOfList method returns True if the reader object has read to the end of a list of items. To read a list of items, call the reader object's ReadListBegin method, then repeatedly read the list items until EndOfList returns True, then call ReadListEnd.

Example

The following code shows the ReadData method of the TStrings object, which reads a list of strings from a reader object into a string-list object:

```
procedure TStrings.ReadData(Reader: TReader);
begin
  Reader.ReadListBegin;
  Clear;
  while not Reader.EndOfList do
    Add(Reader.ReadString);
  Reader.ReadListEnd;
end;
```

See also

ReadListBegin method

ReadListEnd method

EndReferences method

See also Example

Applies to

TReader

Declaration

```
procedure EndReferences;
```

Description

The EndReferences method terminates a block of code that reads components from the reader object's stream that might contain references to each other. EndReferences is always used in the **finally** part of a **try..finally** block that also includes calls to BeginReferences and FixupReferences.

See also

BeginReferences method

FixupReferences method

Error method

Applies to
TReader, TList

Description
There are two methods called Error of interest to component writers. One is called when a reader object encounters an error. The other is called when a list object index is out of bounds.

Error method for reader objects

Error method for list objects

Error method

See also Example

Applies to

TReader

Declaration

```
function Error(const Message: string): Boolean; virtual;
```

Description

The Error method is the protected implementation method for the reader object's OnError event. The return value indicates whether to continue error processing. A return value of True indicates that the application should continue to treat the condition as an error. A return value of False indicates that the error condition is either fixed or should be ignored.

The reader object calls Error whenever it encounters a problem reading a component or a property. By default, Error sets its return value to False, then calls any handler attached to the OnError event, which can change the return value.

Example

The TReader object always calls Error in the **except** part of a **try..except** block, providing the user with a chance to ignore the error. The syntax looks like this:

```
try
  { statements reading components }
except
  on E: Exception do
  begin
    ... { perform some cleanup }
    if not Error(E.Message) then raise;  { reraise exception if Error returns True }
  end;
end;
```

You might have seen one example of this processing in the Delphi environment. When reading a form file, if Delphi encounters an unknown property name or an unregistered component, a reader error occurs, displaying a message box to the user asking whether to ignore the error and continue reading.

See also
OnError event

Error method

See also

Applies to

TList

Declaration

```
procedure Error; virtual;
```

Description

The Error method is called when a list object attempts to act on an illegal index value (such as a negative index, or one beyond the current size of the list). By default, TList.Error raises an EListError exception.

You can override Error to provide additional or alternative error handling and reporting.

See also
EListError object

File-mode constants

See also Example

Unit

Classes

Description

The Classes unit defines a number of file-mode constants used as parameters to the Create constructor of file-stream objects. The constant passed determines the file mode used for opening the stream's associated file.

The following table shows the defined file-mode constants and their meanings.

Constant	Value	Meaning
fmOpenRead	$0000	Open the file for reading only
fmOpenWrite	$0001	Open the file for writing only
fmOpenReadWrite	$0002	Open the file for reading or writing
fmShareExclusive	$0010	Open the file, disallowing other applications to open it for reading or writing.
fmShareDenyWrite	$0020	Open the file, disallowing other applications to open it for writing.
fmShareDenyRead	$0030	Open the file, disallowing other applications to open it for reading.
fmShareDenyNone	$0040	Open the file, disallowing other applications to open it for exclusive use.
fmCreate	$FFFF	Create a new file, replacing any existing file with the same name.

You can combine file-mode constants using the **or** operator.

Example

To open a file stream with its associated file set as read-only, allowing other applications to also read the file, you would combine the file-mode constants as follows:

```
   FileStream := TFileStream.Create('EXAMPLE.STM', fmOpenRead or fmShareDenyWrite);
```

See also
Create method

FindClass function

See also

Unit

Classes

Declaration

```
function FindClass(const ClassName: string): TPersistentClass;
```

Description

The FindClass function searches the list of registered object types and returns an object-type reference to the registered type with the name or alias matching ClassName. If no registered type matches ClassName, FindClass raises an EClassNotFound exception.

The GetClass function performs the same search, but returns **nil** if no registered type matches the specified name.

See also

GetClass function

RegisterClasses procedure

FindMethod method

See also

Applies to

TReader

Declaration

```
function FindMethod(Root: TComponent; const MethodName: string): Pointer;
  virtual;
```

Description

The FindMethod method returns a pointer to the method in the component passed in Root that has the name passed in MethodName, then calls the event handler attached to the OnFindMethod event, if any. If FindMethod cannot locate the named method and the OnFindMethod handler does not clear its error flag, FindMethod raises an EReadError exception.

The reader object calls FindMethod when it reads a property that is a method pointer (that is, usually an event). After reading the name of the assigned method (the event handler), the reader calls FindMethod to get the address of the method in the root object (the form).

You can customize the processing of method pointers read from the reader object by attaching a handler to the OnFindMethod event.

See also
OnFindMethod event

FixupReferences method

See also Example

Applies to

TReader

Declaration

```
procedure FixupReferences;
```

Description

The FixupReferences method resolves the references among various mutually dependent components read from the reader object's stream after all of the objects have been read. FixupReferences is always used inside a **try..finally** block between calls to BeginReferences and EndReferences.

See also

BeginReferences method

EndReferences method

FlushBuffer method

See also

Applies to

TFiler, TReader, TWriter

Declaration

```
procedure FlushBuffer; virtual; abstract;
```

Description

The FlushBuffer method synchronizes the filer object's buffer with the associated stream, either by rereading the buffer (for reader objects) or by writing the current buffer (for writer objects).

TFiler introduces FlushBuffer as an abstract method. TReader and TWriter override it to provide practical implementations.

See also
Create method

FreeNotification method

See also

Applies to

TComponent (all components)

Declaration

```
procedure FreeNotification(AComponent: TComponent);
```

Description

The FreeNotification method ensures that AComponent is notified that the component is going to be destroyed. For components in the same form with the component, the Notification method will be called automatically, but for components in other forms that have references to the component, it is necessary to call FreeNotification.

See also
Notification method

GetChildOwner method

See also

Applies to

TComponent (all components)

Declaration

```
function GetChildOwner: TComponent; dynamic;
```

Description

The GetChildOwner method returns the owner of a child component being read from a stream. If GetChildOwner returns **nil**, the owner is assumed to be the root component currently being read (that is usually a form).

See also
Owner property

GetChildParent method

See also

Applies to

TComponent (all components)

Declaration

```
function GetChildParent: TComponent; dynamic;
```

Description

The GetChildParent method returns the parent of a child component being read from a stream. If GetChildParent returns **nil**, the parent is assumed to be the root component currently being read (that is usually a form).

See also
Parent property

GetChildren method

See also Example

Applies to

TComponent

Declaration

```
procedure GetChildren(Proc: TGetChildProc); dynamic;
```

Description

The GetChildren method returns the "child" components of the component; that is, those who return the component from their GetParentComponent method.

By default, GetChildren returns no children. TWinControl overrides GetChildren to return all the controls that have it as parent. For example, a panel component returns all the controls it contains, and a menu item returns all its subitems.

For each child item, GetChildren should call the method passed in Proc. The children must be returned in creation order, the order they will appear in a form file.

Note: GetChildren replaces the WriteComponents method used in Delphi 1.0. GetChildren supports form inheritance.

Example

Here is the GetChildren method from TWinControl. Note that it iterates through its list of controls and passes each to the Proc routine:

```
procedure TWinControl.GetChildren(Proc: TGetChildProc);
var
  I: Integer;
  Control: TControl;
  Form: TForm;
begin
  Form := GetParentForm(Self);    { find the form we're in }
  for I := 0 to ControlCount - 1 do      { iterate through child controls }
  begin
    Control := Controls[I];       { take each control...}
    if Control.Owner = Form then  { ...and if it's in this form...}
      Proc(Control);      { ...pass it to Proc }
  end;
end;
```

See also

GetClass function

See also

Unit

Classes

Declaration

```
function GetClass(const ClassName: string): TPersistentClass;
```

Description

The GetClass function searches the list of registered object types and returns an object-type reference to the registered type with the name or alias matching ClassName. If no registered type matches ClassName, GetClass returns **nil**.

The FindClass function performs the same search, but raises an exception if no registered type matches the specified name.

See also

FindClass function

RegisterClasses procedure

GetFixupRoots procedure

Unit
Classes

Declaration
```
procedure GetFixupRoots(Names: TStrings; Root: TComponent);
```

Description
The GetFixupRoots procedure fills the Names string list with the names of any root components (that is, in general, forms) that still need to be loaded to resolve pointers in the already-loaded component, Root. Such pointers are a result of form linking.

If Root is **nil**, GetFixupRoots fills Names with the names of all pending root components.

GetParentComponent method

See also

Applies to

TComponent

Declaration

```
function GetParentComponent: TComponent; dynamic;
```

Description

The GetParentComponent method returns the parent of the component. Controls and menu items return the value of their Parent property. Field components return their associated data sets. Components that don't have an actual "parent" return the value of their Owner properties.

See also
Parent property

Grow method

See also Example

Applies to

TList, TStringList

Declaration

```
procedure Grow; virtual;
```

Description

The Grow method increases the capacity of the list object. By default, Grow increases the capacity by differing amounts, depending on the current size of the list, as shown in the following table.

Current capacity	Increment
0..4	4
5..8	8
9+	16

The list object calls Grow whenever adding or inserting an item causes the list to fill its current capacity. You can override Grow to change the size of the capacity increment, or to limit the size of the list.

Example

The following code shows an override to the Grow method that always increases the capacity of the list by 42 items.

```
procedure TMyList.Grow;
begin
  Capacity := Capacity + 42;
end;
```

See also
Capacity property

Handle property

See also

Applies to

THandleStream

Declaration

```
property Handle: Integer;
```

Description

The Handle property provides read-only access to the file handle passed to the handle stream's Create constructor. You can then pass the handle to file-management routines that require a file handle.

See also
Create method

HasParent method

See also

Applies to

TComponent, TControl, TMenuItem

Declaration

```
function HasParent: Boolean; dynamic;
```

Description

The HasParent function indicates whether an object has a parent that is responsible for writing the object. By default, TComponent's HasParent method returns False. TControl and TMenuItem both override HasParent to always return True.

A return value of True indicates that some other object will be responsible for writing the object to a stream in its WriteComponents method. Most commonly that other object is a form or panel component that contains a control.

Note: Any child that returns True from HasParent must now also implement the GetParentComponent and SetParentComponent methods.

You should rarely need to override HasParent.

See also

Parent property

GetParentComponent method

SetParentComponent method

IgnoreChildren property

See also

Applies to

TFiler

Declaration

```
property IgnoreChildren: Boolean;
```

Description

The IgnoreChildren property enables a writer object to save a component without saving the components it owns. If IgnoreChildren is True, the writer writes a component without writing any child components it owns. Otherwise, the writer object writes all owned objects to the stream.

See also
WriteComponent method

InitComponentRes function

See also

Unit

Classes

Declaration

```
function InitComponentRes(const ResName: string; Instance: TComponent):
  Boolean;
```

Description

The InitComponentRes function is identical to the ReadComponentRes function, except that

- Instance must be non-**nil**
- Returns False if ResName is not found (rather than raising an exception)

See also
ReadComponentRes function

LineStart function

Unit
Classes

Declaration
```
function LineStart(Buffer, BufPos: PChar): PChar;
```

Description
The LineStart function returns a pointer to the first character of the next line of text in the text buffer passed in Buffer. The BufPos parameter indicates the current or starting position of the search. LineStart begins searching the buffer at the character indicated by BufPos and returns a pointer to the first character following a linefeed (#10).

LineStart is a utility function used for finding the separate lines in a large text buffer, such as that used by a Memo component.

Loaded method

See also Example

Applies to

TComponent

Declaration

```
procedure Loaded; virtual;
```

Description

The Loaded method provides an opportunity for a component to initialize itself after all its parts have loaded from a stream. When a Delphi application loads a form from its form file, for example, it first constructs the form component by calling its Create constructor, then reading its property values from the form file, which is a stream. After reading all the property values for all the components, Delphi calls the Loaded methods of each component in the order the components were created. This gives the components a chance to initialize any data that depends on the values of other components or other parts of itself.

Example

The Media Player component overrides Loaded to automatically open its associated media file if its AutoOpen property is True. By opening the file in the Loaded method instead of in the Create constructor, the media player can be certain that all its properties are properly initialized before opening.

```
procedure TMediaPlayer.Loaded;
begin
  inherited Loaded;        { always call the inherited Loaded first! }
  if (not (csDesigning in ComponentState)) and FAutoOpen then
    Open;
end;
```

See also
Create method

LoadFromFile method

See also

Applies to

TMemoryStream

Declaration

```
procedure LoadFromFile(const FileName: string);
```

Description

The LoadFromFile method reads the entire contents of the file specified by FileName into the memory stream, replacing any existing contents. The memory stream becomes an in-memory copy of the file.

See also
SaveToFile method, LoadFromStream method

LoadFromStream method

See also

Applies to

TMemoryStream

Declaration

```
procedure LoadFromStream(Stream: TStream);
```

Description

The LoadFromStream method copies the entire contents of the stream specified by Stream into the memory stream, replacing any existing contents. The memory stream becomes a copy of the source stream.

See also

SaveToStream method

LoadFromFile method

MaxListSize constant

See also

Unit

Classes

Declaration

```
const
    MaxListSize = MaxInt div 16;
```

Description

The MaxListSize constant represents the largest number of items a list object can contain.

See also
Count property

Memory property

See also

Applies to

TCustomMemoryStream

Declaration

```
property Memory: Pointer;
```

Description

The Memory property provides direct access to the memory pool allocated for the memory stream. Note that Memory is a read-only property. If you need to change the memory allocated, you can allocate the memory and then assign it to Memory by calling the SetPointer method.

See also
Clear method
SetPointer method

NextValue method

See also

Applies to

TReader

Declaration

```
function NextValue: TValueType;
```

Description

The NextValue method returns the type of the next item in the reader object's stream, and returns with the stream position still before the value-type indicator.

See also
ReadValue method

Notification method

See also Example

Applies to

TComponent

Declaration

```
procedure Notification(AComponent: TComponent; Operation: TOperation);
  virtual;
```

Description

The Notification method notifies the component that the component specified by AComponent is about to be inserted or removed, as specified by Operation. By default, components pass along the notification to their owned components, if any.

A component can, if needed, act on the notification that a component is being inserted or removed. In particular, if a component has object fields or properties that contains references to other components, it might check the notifications of component removals and invalidate those references as needed.

Example

The Batch Move component overrides Notification. If Operation is opRemove, the batch-move component compares AComponent to its Source and Destination properties. If either of those is the component being removed, the batch-move component sets the corresponding property to **nil**.

See also

InsertComponent method

RemoveComponent method

TOperation type

ObjectBinaryToText procedure

See also

Unit

Classes

Declaration

```
procedure ObjectBinaryToText(Input, Output: TStream);
```

Description

The ObjectBinaryToText procedure converts components stored in the binary stream (such as one written by the WriteRootComponent method) passed in Input into a text-based representation in the stream passed in Output. You can then process the text version as you would any text file, such as making global search-and-replace changes with a text editor, then reconvert the text file to a binary form by calling ObjectTextToBinary.

See also

ObjectTextToBinary procedure

ObjectResourceToText procedure

See also

Unit

Classes

Declaration

```
procedure ObjectResourceToText(Input, Output: TStream);
```

Description

The ObjectResourceToText procedure converts components stored in a Windows resource file format (such as a Delphi form file) passed in Input into a test-based representation in the stream passed in Output. You can then process the text version as you would any text file, such as making global search-and-replace changes with a text editor, then reconvert the text file to a binary form by calling ObjectTextToResource.

ObjectResourceToText reads a resource-file header from Input by calling the ReadResHeader method, the calls ObjectBinaryToText to convert the binary data contained in the resource file.

See also
ObjectTextToResource procedure

ObjectTextToBinary procedure

See also

Unit

Classes

Declaration

```
procedure ObjectTextToBinary(Input, Output: TStream);
```

Description

The ObjectTextToBinary procedure converts a text-based representation of stored components passed in Input into a binary form (equivalent to that written by the WriteRootComponent method) in the stream passed in Output.

ObjectTextToBinary is most useful for reconverting text files generated by the ObjectBinaryToText procedure, but can also generate binary files from text written by another application.

See also

ObjectBinaryToText procedure

ObjectTextToResource procedure

ObjectTextToResource procedure

See also

Unit

Classes

Declaration

```
procedure ObjectTextToResource(Input, Output: TStream);
```

Description

The ObjectTextToResource procedure converts the text-based representation of stored components (such as a form file saved as text from the Delphi Code editor) passed in Input to a binary Windows resource-file format in the stream passed in Output.

ObjectTextToResource is most useful for generating form files from modified text versions of form files, but it can also convert compatible text files written by other applications or by the ObjectResourceToText procedure.

ObjectTextToResource calls the ObjectTextToBinary procedure to convert the text file into a binary format, then writes a resource-file header and stores the binary data in the resource file.

See also

ObjectResourceToText procedure

ObjectTextToBinary procedure

OnError event

See also

Applies to

TReader

Declaration

```
property OnError: TReaderError;
```

Description

The OnError event occurs whenever a reader object encounters an error reading its data, such as reading the name of an undeclared property or an illegal value. By handling OnError events, you can selectively choose to process or ignore errors.

The last parameter passed to the OnError event handler is a **var** parameter called Handled. By default, the Error method passes False in Handled, but a handler that corrects the error or chooses to ignore the error can set Handled to True, which prevents further processing of the error. If the event handler returns with Handler still set to False, the reader object raises an EReadError exception.

See also
Error methodFor reader objects

OnFindMethod event

See also

Applies to

TReader

Declaration

```
property OnFindMethod: TFindMethodEvent;
```

Description

The OnFindMethod event occurs each time the reader object reads a method-pointer property for an object. Properties that are method pointers are almost always events.

The most common reason to handle the OnFindMethod event is to ask the user how to handle missing methods. If the FindMethod method does not locate the named method assigned to the method pointer, it sets the Error parameter to the OnFindMethod handler to True. Error is a **var** parameter, however, so the handler can set Error to False, which prevents FindMethod from raising an exception when the handler returns.

See also
FindMethod method

OnSetName event

See also Example

Applies to

TReader

Declaration

```
property OnSetName: TSetNameEvent;
```

Description

The OnSetName event occurs just before the reader object sets the Name property of a component it reads from its stream. The Name parameter to the OnSetName event handler is a **var** parameter, so the handler can change the name before the reader assigns it to the component. This is useful if you want to filter all the component names in a form, for instance, to add or change part of the string.

Example

The following handler for the OnSetName event renames all components that contain the word "Button" in their names to contain "PushButton" instead:

```
procedure TForm1.ReaderSetName(Reader: TReader; Component: TComponent;
  var Name: string);
var
  ButtonPos: Integer;
begin
  ButtonPos := Pos('Button', Name);
  if ButtonPos <> 0 then
    Name := Copy(Name, 1, ButtonPos - 1) + 'PushButton' +
      Copy(Name, ButtonPos + 6, Length(Name));
end;
```

See also
SetName methodFor reader objects

Owner property

See also

Applies to

TReader

Declaration

```
property Owner: TComponent;
```

Description

The Owner property for a reader object stores the component that will be assigned as the Owner property of all components read from the reader object's stream.

See also

Owner property for components

Parent property

Parent property

See also

Applies to

TReader

Declaration

```
property Parent: TComponent;
```

Description

The Parent property for a reader object stores the component that will be assigned as the Parent property of all controls read from the reader object's stream.

See also

Parent property for controls

Owner property

Position property

Applies to
TStream, TReader, TWriter

Description
There are two related properties called Position used in the Delphi streaming and filing system. One is the current position of the stream, the other is the current reading or writing position in the reader or writer object's buffer.

Position property for stream objects

Position property for reader and writer objects

Position property

See also

Applies to

TStream

Declaration

```
property Position: Longint;
```

Description

The Position property indicates the current offset into the stream for reading and writing.

See also

Seek method

Size property

Position property

See also

Applies to

TReader, TWriter

Declaration

```
property Position: Longint;
```

Description

The Position property for a filer object represents the current reading or writing position in the associated stream. The value of Position will generally be inside the most recent buffer block read or the next block to be written. Thus, for writer objects, Position will generally be less than the stream's Position, and for reader objects, it will be greater.

Setting Position to a location outside the current buffer causes a call to FlushBuffer.

See also

Position property for streams

FlushBuffer method

Read method

See also

Applies to

TStream

Declaration

```
function Read(var Buffer; Count: Longint): Longint; virtual; abstract;
```

Description

The Read method reads Count bytes from the stream into Buffer, advances the current position of the stream by Count bytes, and returns the number of bytes actually read. If the return value is less than Count, it means that reading reached the end of the stream data before reading the requested number of bytes.

TStream.Read is abstract. Each descendant stream type defines a Read method that reads data from its particular storage medium (such as memory or a disk file). All the other data-reading methods of a stream (ReadBuffer, ReadComponent) call Read to do their actual reading.

See also
Write methodFor stream objects

ReadBoolean method

See also

Applies to

TReader

Declaration

```
function ReadBoolean: Boolean;
```

Description

The ReadBoolean method reads a Boolean value from the reader object's stream and returns its value.

See also
Read method
WriteBoolean method

ReadBuffer method

See also

Applies to

TStream

Declaration

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

Description

The ReadBuffer method reads Count bytes from the stream into Buffer and advances the current position of the stream by Count bytes. If reading attempts to read past the end of the stream, ReadBuffer raises an EReadError exception.

See also

Read method

WriteBuffer method

ReadChar method

See also

Applies to

TReader

Declaration

```
function ReadChar: Char;
```

Description

The ReadChar method reads a character from the reader object's stream and returns that character value.

See also

Read method

WriteChar method

ReadComponent method

See also

Applies to

TStream

Declaration

```
function ReadComponent(Instance: TComponent): TComponent;
```

Description

The ReadComponent method reads the component specified by Instance from the stream and returns the component. ReadComponent constructs a reader object and calls its ReadRootComponent method to read Instance and any objects owned by Instance.

If Instance is **nil**, ReadComponent constructs a component based on the type information in the stream and returns the newly-constructed component.

See also

ReadRootComponent method

WriteComponent methodFor stream objects

ReadComponentRes method

See also

Applies to

TStream

Declaration

```
function ReadComponentRes(Instance: TComponent): TComponent;
```

Description

The ReadComponentRes method reads the component specified by Instance from the stream. The current position of the stream must point to a component written using the WriteComponentRes method.

ReadComponentRes first calls ReadResHeader to read a resource header from the stream, then calls ReadComponent to read Instance. If the stream does not contain a resource header at the current position, ReadResHeader will raise an EInvalidImage exception.

Note: There is also a function in the Classes unit called ReadComponentRes that performs much the same operation, but which creates its own stream based on the application's resource.

See also
WriteComponentRes method
ReadResHeader method
ReadComponent method
ReadComponentRes function

ReadComponentRes function

See also

Unit

Classes

Declaration

```
function ReadComponentRes(const ResName: string; Instance: TComponent):
  TComponent;
```

Description

The ReadComponentRes function locates the resource named by ResName by calling the AccessResource API function, then opens a handle-stream object using the resulting handle. ReadComponentRes then calls the stream's ReadComponent method to read Instance from the resource and returns the value returned by ReadComponent.

You can use ReadComponentRes as a convenient way to read a resourced component without having to manually create and dispose of a stream object.

See also

ReadComponent method

ReadComponentResFile function

ReadComponentRes method

InitComponentRes function

ReadComponentResFile function

See also

Unit

Classes

Declaration

```
function ReadComponentResFile(const FileName: string; Instance: TComponent):
  TComponent;
```

Description

The ReadComponentResFile function reads a component from the file specified by FileName and returns the component read, if any. Delphi uses ReadComponentResFile to read forms and their components from form files. You can therefore write utilities that read Delphi form files, modify their components, and write the restults back to the form file.

ReadComponentResFile constructs a file-stream object associated with FileName, then calls the stream's ReadComponentRes method to load Instance.

See also

WriteComponentResFile procedure

ReadComponentRes method

ReadComponents method

See also

Applies to

TReader

Declaration

```
procedure ReadComponents(AOwner, AParent: TComponent; Proc:
  TReadComponentsProc);
```

Description

The ReadComponents method reads a list of components from the reader object's associated stream.

ReadComponents first sets the reader object's Root and Owner properties to the component passed in the AOwner parameter and sets its Parent property to AParent. Next ReadComponents reads components by calling the ReadComponent method, passing each returned component to the method passed in Proc.

After loading all the components in the list, ReadComponents calls the Loaded method of each loaded component in the order it read them.

See also

ReadComponent method

WriteComponents method

ReadFloat method

See also

Applies to

TReader

Declaration

```
function ReadFloat: Extended;
```

Description

The ReadFloat method reads a floating-point number from the reader object's stream and returns its value.

See also

Read method

WriteFloat method

ReadIdent method

See also

Applies to

TReader

Declaration

```
function ReadIdent: string;
```

Description

The ReadIdent method reads an identifier from the reader object's stream and returns the identifier.

See also

Read method

WriteIdent method

ReadInteger method

See also

Applies to

TReader

Declaration

```
function ReadInteger: Longint;
```

Description

The ReadInteger method reads an integer-type number from the reader object's stream and returns its value.

See also
Read method, WriteInteger method

ReadListBegin method

See also Example

Applies to

TReader

Declaration

```
procedure ReadListBegin;
```

Description

The ReadListBegin method reads a start-of-list marker from the reader object's associated stream. If the next item in the stream is not a start-of list marker as written by the WriteListBegin method, ReadListBegin raises an EReadError exception.

A call to ReadListBegin is generally followed by a reading loop that terminates when the EndOfList method returns True, indicating that an end-of-list marker is next on the stream, at which point a call to ReadListEnd is required.

Example

The TStrings object uses ReadListBegin and ReadListEnd when reading its list of strings from a reader object. The following ReadData method is the method TStrings registers for reading its string data in its DefineProperties method:

```
procedure TStrings.ReadData(Reader: TReader);
begin
  Reader.ReadListBegin;    { read the start-of-list marker }
  Clear;{ clear any existing strings }
  while not Reader.EndOfList do   { as long as there is still data... }
    Add(Reader.ReadString);       { ...read a string and add it to the list }
  Reader.ReadListEnd;      { get past the end-of-list marker }
end;
```

See also

Read method

ReadListEnd method

WriteListBegin method

ReadListEnd method

See also Example

Applies to

TReader

Declaration

```
procedure ReadListEnd;
```

Description

The ReadListEnd method reads and end-of-list marker from the reader object's associated stream. If the item read is not an end-of-list marker, ReadListEnd raises an EReadError exception. A call to ReadListEnd must correspond to a preceeding call to ReadListBegin.

See also
Read method
EndOfList method
ReadListBegin method
WriteListEnd method

ReadPrefix method

See also

Applies to

TReader

Declaration

```
procedure ReadPrefix(var Flags: TFilerFlags; var AChildPos: Integer);
```

Description

The ReadPrefix method reads the component prefix from the reader object's stream. When a writer object writes a component to its stream, it prefaces the component itself with two values. The first is a flag that indicates whether the component is inherited from an ancestor form and whether its position in its form is important. The second indicates the position in the ancestor form's creation order.

The ReadComponent method calls ReadPrefix automatically, but you can also call it directly if you need to read a component's prefix separately.

See also
ReadComponent method

ReadResHeader method

See also

Applies to

TStream

Declaration

```
procedure ReadResHeader;
```

Description

The ReadResHeader method reads a Windows resource-file header from the stream and moves the current position of the stream to just beyond the header. If the stream does not contain a valid resource-file header, ReadResHeader raises an EInvalidImage exception.

The ReadComponentRes method calls ReadResHeader automatically before reading a component from a resource file. You should generally not need to call it yourself.

See also
ReadComponentRes method

ReadRootComponent method

See also

Applies to

TReader

Declaration

```
function ReadRootComponent(Root: TComponent): TComponent;
```

Description

The ReadRootComponent method reads a component and all its owned components from the reader object's stream. ReadRootComponent first calls the ReadSignature method to ensure that it is reading a proper component, then reads the component's properties and owned objects. The component passed in Root becomes the value of the Root property for the reader object, which assigns as the owner of each owned object read.

See also

Root property

ReadComponents method

WriteRootComponent method

ReadSignature method

See also

Applies to

TReader

Declaration

```
procedure ReadSignature;
```

Description

The ReadSignature method reads a Delphi filer signature from the reader object's associated stream. The ReadRootComponent method calls ReadSignature before reading its component from the stream. By checking for the signature before loading objects, the reader object can guard against inadvertantly reading invalid or corrupted data.

The filer signature is a four-character sequence. For this version of Delphi, the signature is 'TPF0'.

See also

ReadRootComponent method

WriteSignature method

ReadState method

See also

Applies to

TComponent, TControl, TWinControl

Declaration

```
procedure ReadState(Reader: TReader); virtual;
```

Description

The ReadState method reads the value of the component's published propeties, stored data, and owned components from the reader object passed in Reader.

See also
WriteState method

ReadStr method

See also

Applies to

TReader

Declaration

```
function ReadStr: string;
```

Description

The ReadStr method reads a string from the reader object's stream and returns its contents. ReadStr is used internally by some of Delphi's standard components, but your components should not use it.

Caution: Always use ReadString and WriteString for reading and writing component strings to streams. The similarly-named ReadStr and WriteStr methods are for internal use by certain VCL components and will likely corrupt your data if you attempt to use them.

See also

Read method

ReadString method

WriteStr method

ReadString method

See also

Applies to

TReader

Declaration

```
function ReadString: string;
```

Description

The ReadString method reads a string written by WriteString from the reader object's stream and returns its contents.

Caution: Always use ReadString and WriteString for reading and writing component strings to streams. The similarly-named ReadStr and WriteStr methods are for internal use by certain VCL components and will likely corrupt your data if you attempt to use them.

See also

Read method

WriteString method

ReadValue method

See also

Applies to

TReader

Declaration

```
function ReadValue: TValueType;
```

Description

The ReadValue method reads the type of the next item on the reader object's stream and returns with the stream positioned after the value-type indicator.

See also
NextValue method

Realloc method

See also

Applies to

TMemoryStream

Declaration

```
function Realloc(var NewCapacity: Longint): Pointer; virtual;
```

Description

The Realloc method reallocates the specified amount of memory, rounded up to the nearest 8K, and returns a pointer to it.

See also
Memory property
SetPointer method

RegisterClass procedure

See also Example

Unit

Classes

Declaration

```
procedure RegisterClass(AClass: TPersistentClass);
```

Description

The RegisterClass procedure registers the object type passed in the AClass parameter with the Delphi streaming system.

In order to store objects on streams and read them back, the object types of those objects must be registered. You can also register an object type with an alternate name, called an alias, by calling RegisterClassAlias instead of RegisterClass.

Example

Each of the following two procedure calls registers an object type:

```
   RegisterClass(TAnObject);
   RegisterClass(TAnotherObject);
```

See also

RegisterClassAlias procedure

RegisterClasses procedure

UnRegisterClass procedure

RegisterClassAlias procedure

See also Example

Unit

Classes

Declaration

```
procedure RegisterClassAlias(AClass: TPersistentClass; const Alias: string);
```

Description

The RegisterClassAlias procedure registers the object type passed in the AClass parameter with the Delphi streaming system and also adds the name passed in Alias as an alternate name for the registered type.

In order to store objects on streams and read them back, the object types of those objects must be registered.

Example

The following procedure call registers an object type and also specifies an alternate name for that type:

```
   RegisterClassAlias(TAnObject, 'TAnAliasForTAnObject');
```

See also

RegisterClass procedure

GetClass function

FindClass function

RegisterClasses procedure

See also Example

Unit

Classes

Declaration

```
procedure RegisterClasses(AClasses: array of TPersistentClass);
```

Description

The RegisterClasses procedure registers each of the object types passed in the AClasses array parameter with the Delphi streaming system by calling the RegisterClass procedure for each element in the array. RegisterClasses is a somewhat more compact way to register multiple object types.

In order to store objects on streams and read them back, the object types of those objects must be registered.

Example

The following procedure call registers two object types with a single call:

```
   RegisterClasses([TAnObject, TAnotherObject]);
```

See also

RegisterClass procedure

RegisterClassAlias procedure

UnRegisterClasses procedure

RegisterComponents procedure

See also Example

Unit

Classes

Declaration

```
procedure RegisterComponents(const Page: string; ComponentClasses: array of
  TComponentClass);
```

Description

The RegisterComponents procedure registers components with the Delphi component library, allowing them to appear on the Component palette. The Page parameter specifies the name of the Component-palette page the component types passed in ComponentClasses will appear on. If the specified page does not already exist in the Component palette, Delphi creates one.

Calls to RegisterComponents always appear in a procedure called Register in a unit that contains component declarations and implementations. When you install a component unit into the component library, Delphi looks for the Register procedure for that unit (and reports and error if it does not find one). Using RegisterComponents you can register one or more components on any given page with a single call.

Example

The following code shows a typical Register procedure for a unit that declares three new component types, named TFirst, TSecond, and TThird:

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TFirst, TSecond]);       { two on this page }
  RegisterComponents('Assorted', [TThird]);      { and one on another }
end;
```

See also
Registering components with Delphi

RegisterIntegerConsts procedure

See also

Unit

Classes

Declaration

```
procedure RegisterIntegerConsts(IntegerType: Pointer; IdentToInt:
  TIdentToInt;
    IntToIdent: TIntToIdent);
```

Description

The RegisterIntegerConsts procedure registers equivalent constant identifiers and values for use by the streaming system. Registered types can store identifiers instead of numeric values. For example, forms can write their color values into form files using the clXXX constants instead of numeric values, making it easier to modify the form file by hand.

Delphi registers constants for three standard types to make form files more useful to human readers: TColor, TCursor, and TShortCut. Registering other types can slow down the filing process, so register new types judiciously.

In addition to the type to register constants for, RegisterIntegerConstants takes two functions as parameters, specifying how to convert from identifiers into integers and back. For example, the TCursor type uses the IdentToCursor and CursorToIdent functions.

See also

TIdentToInt type

TIntToIdent type

RegisterNoIcon procedure

Unit
Classes

Declaration
```
procedure RegisterNoIcon(ComponentClasses: array of TComponentClass);
```

Description
The RegisterNoIcon procedure adds the component types passed in the ComponentClasses array parameter to a list of nonvisual components that should not have icons appear for them at design time. Creating such components still adds corresponding fields to the form's type declaration, but the individual components do not have visible representation on the form.

Nonvisual components without icons are useful for components that are always subcomponents associated with other components. Delphi defines two of these: menu-item components, which are always associated with either a Main Menu or a Popup Menu, and fiel components, which are always associated with a Table component.

Root property

See also

Applies to

TFiler

Declaration

```
property Root: TComponent;
```

Description

The Root property indicates to the filer object which object is the root, or ultimate owner, of the objects read or written by the filer. The ReadRootComponent and WriteRootComponent methods set Root before reading or writing their components and owned components.

See also

ReadRootComponent method

WriteRootComponent method

RootAncestor property

See also

Applies to

TWriter

Declaration

```
property RootAncestor: TComponent;
```

Description

The RootAncestor property represents the ancestor of the component in the Root property. If Root is an inherited form, the writer object iterates through each of the form's owned components, comparing each to the corresponding component in the ancestor form, and writes only those that differ in some way from the ancestor component. The Ancestor property tracks each component in the ancestor form, which is always RootAncestor.

See also
Ancestor property
Root property

SaveToFile method

See also

Applies to

TCustomMemoryStream

Declaration

```
procedure SaveToFile(const FileName: string);
```

Description

The SaveToFile method writes the entire contents of the memory stream to the file specified by FileName. If the specified file already exists, SaveToFile overwrites it. The file becomes a binary copy of the memory stream contents.

See also

LoadFromFile method

SaveToStream method

SaveToStream method

See also

Applies to

TCustomMemoryStream

Declaration

```
procedure SaveToStream(Stream: TStream);
```

Description

The SaveToStream method writes the entire contents of the memory stream to the stream specified by Stream.

See also

LoadFromStream method

SaveToFile method

Seek method

See also

Applies to

TStream

Declaration

```
function Seek(Offset: Longint; Origin: Word): Longint; virtual; abstract;
```

Description

The Seek method moves the current position of the stream by Offset bytes, relative to the origin specified by Origin. If Offset is a negative number, the seek is backward from the specified origin. The following table shows the different values of Origin and their meanings for seeking.

Constant	Value	Seek origin
soFromBeginning	0	Beginning of the stream
soFromCurrent	1	Current position
soFromEnd	2	End of the stream

See also
Position propertyFor stream objects

SetAncestor method

See also

Applies to

TComponent (all components)

Declaration

```
procedure SetAncestor(Value: Boolean);
```

Description

The SetAncestor method sets or clears the csAncestor flag in the component's ComponentState property and all components owned by the component.

See also
ComponentState property

SetChildOrder method

See also

Applies to

TComponent (all components)

Declaration

```
procedure SetChildOrder(Child: TComponent; Order: Integer); dynamic;
```

Description

The SetChildOrder method changes the order in which Child appears the the child components returned by GetChildren. The child moves as if it were an item in a list object: items previously below the child's old position move up, and those below the new position move down.

See also

GetChildren method

SetDesigning method

See also

Applies to

TComponent (all components)

Declaration

```
procedure SetDesigning(Value: Boolean);
```

Description

The SetDesigning method sets the csDesigning option in the ComponentState property if Value is True, otherwise, it removes csDesigning. SetDesigning then calls the SetDesigning methods of any owned components, passing Value, so that the owned components' ComponentState properties will be synchronized with the owner's.

The InsertComponent and RemoveComponent methods call SetDesigning for inserted or removed components to ensure that their design-mode flags are set properly.

See also

ComponentState property

InsertComponent method

RemoveComponent method

SetName method

Applies to
TReader object, TComponent (all components)

Description
There are two different methods called SetName. One is for reader objects, and allows changing the names of components as they are read from a stream. The other is the protected property-implementation for the Name property in all components.

SetName method for reader objects

SetName method for all components

SetName method

See also

Applies to

TReader

Declaration

```
procedure SetName(Component: TComponent; var Name: string); virtual;
```

Description

The SetName method allows a reader object to change the names of components being read from the associated stream before assigning them to the Name properties of the components. The ReadComponent method reads the type and name of a component before reading the component's property values and other data. After reading the name, ReadComponent passes the name read from the stream in the Name parameter to SetName. Name is a **var** parameter, so SetName can change the string before returning.

SetName also calls any event handler attached to the OnSetName event, passing the name string in a **var** parameter, so the attached handler can also modify the string.

See also
OnSetName event

SetName method

See also

Applies to

TComponent (all components)

Declaration

```
procedure SetName(const NewName: TComponentName); virtual;
```

Description

The SetName method is the property-access method for setting the value of the Name property.

See also
Name property

SetParentComponent method

See also

Applies to

TComponent (all components)

Declaration

```
procedure SetParentComponent(Value: TComponent); dynamic;
```

Description

The SetParentComponent method changes the value that GetParentComponent returns to match Value. In most cases, that value is the Parent property, but not always.

See also

GetParentComponent method

Parent property

SetPointer method

See also

Applies to

TCustomMemoryStream

Declaration

```
procedure SetPointer(Ptr: Pointer; Size: Longint);
```

Description

The SetPointer method assigns the pointer passed in Ptr to the internal storage for the Memory property and sets the stream's Size property to the value passed in Size.

Call SetPointer in your memory streams when you have allocated or reallocated the memory pool for the stream's storage.

See also

Memory property

Size property

SetSize method

See also

Applies to

TMemoryStream

Declaration

```
procedure SetSize(Size: Longint);
```

Description

The SetSize method clears the memory stream contents and sets the size of the memory pool for the memory stream to Size bytes. If Size is zero, SetSize disposes of the existing memory pool and sets the Memory property to **nil**. Otherwise, SetSize adjusts the size of the memory pool to Size.

See also
Size property

Size property

See also

Applies to

TStream

Declaration

```
property Size: Longint;
```

Description

The Size property indicates the size in bytes of the stream. Size is read-only.

See also

Position propertyFor stream objects

TComponent component

See also Properties Methods Components

Unit

Classes

Description

The TComponent component is the most basic, abstract starting point for all components. Every item the user can manipulate in the form designer is a descendant of TComponent, and all components inherit some basic behavior from TComponent.

In addition to everything it inherits from its ancestor, TPersistent, TComponent declares a number of properties and methods common to all components. Most of these are public or published, and are described in the VCL reference for all components.

TComponent also declares several protected methods that descendant components can override to customize their behavior. These methods fall into three categories:

- Methods for managing the Name property

  The SetName method is the virtual method used as the write part of the Name property. It calls the ChangeName method, which performs the actual name change.

- Methods for streaming and filing

  The ReadState and WriteState methods respectively read and write the component's property values to a filer object. The HasParent method determines whether the component has a parent to handle its filing for it. The Loaded method performs initialization after loading from a filer object. TComponent also overrides the DefineProperties method to define "fake" Left and Right properties.

- Methods for managing owned components

  The ValidateRename method ensures that renaming an owened component does not create name conflict. The WriteComponents method writes owned components to a filer object. The Notification method forwards notification messages to all owned components. The SetDesigning method ensures that components inserted at design time have their design-mode flag set.

Properties

ComponentCount	ComponentState	Name
ComponentIndex	ComponentStyle	Owner
Components	DesignInfo	Tag

Methods

ChangeName	GetChildParent	SetAncestor
Create	GetChildren	SetChildOrder
DefineProperties	GetParentComponent	SetDesigning
Destroy	HasParent	SetName
DestroyComponents	InsertComponent	SetParentComponent
Destroying	Loaded	Updated
FindComponent	Notification	Updating
FreeNotification	ReadState	ValidateRename
GetChildOwner	RemoveComponent	WriteState

See also
TComponentClass type

TComponentClass type

See also

Unit

Classes

Declaration

```
type
    TComponentClass = class of TComponent;
```

Description

The TComponentClass type is the class reference type for all components.

See also
TComponent component

TComponentState type

See also

Unit

Classes

Declaration

```
type
    TComponentState = set of (csLoading, csReading, csWriting, csDestroying,
  csDesigning,
      csAncestor, csUpdating, csFixups);
```

Description

The TComponentState type defines the set of possible state flags for the ComponentState property. The following table describes the state corresponding to each flag.

Flag	Component state
csAncestor	Set if the component was introduced in an ancestor form. Only set if csDesigning is also set.
csDesigning	Design mode, meaning it is in a form being manipulated by a form designer.
csDestroying	The component is about to be destroyed.
csFixups	Set if the component is linked to a component in another form that has not yet been loaded. This flag is cleared when all pending fixups are resolved.
csLoading	Loading from a filer object.
csReading	Reading its property values from a stream.
csUpdating	The component is being updated to reflect changes in an ancestor form. Only set if csAncestor is also set.
csWriting	Writing its property values to a stream.

See also
ComponentState property

TComponentStyle type

See also

Unit

Classes

Declaration

```
type
    TComponentStyle = set of (csInheritable, csCheckPropAvail);
```

Description

The TComponentStyle type defines a set of values for the ComponentStyle property. ComponentStyle determines whether a component (and therefore the forms that contain it) is inheritable, and whether it needs to check its properties for readability.

See also
ComponentStyle property

TCustomMemoryStream object

See also Properties Methods Objects

Unit

Classes

Description

The TCustomMemoryStream object is an abstract base class used as the common ancestor for useful memory streams, such as TMemoryStream and TResourceStream.

In addition to the items it inherits from TStream, TCustomMemoryStream defines a Memory property that provides a pointer to the stream's internal storage, overrides the inherited Read and Seek methods to implement access within that memory, and adds methods for saving the memory to files and streams.

Property

<u>Memory</u>

Methods

Read	SaveToStream	SetPointer
SaveToFile	Seek	

See also

TStream object

TMemoryStream object

TFiler object

See also Properties Methods Objects

Unit

Classes

Description

The TFiler object is the abstract base object for the reader objects and writer objects that Delphi uses for saving forms and components in form files. Among other things, filer objects provide buffering of data to speed read and write operations.

Every filer has an associated stream object passed as a parameter to its Create method. In addition, it defines a Root property, which specifies the root object read or written.

Filers also have two methods for defining properties, DefineProperty and DefineBinaryProperty. These enable an object to store hidden or complex data in a different form from its internal representation, as if they were actual properties of the object.

Properties

Ancestor	IgnoreChildren	Root

Methods

Create	DefineProperty	FlushBuffer
DefineBinaryProperty	Destroy	

See also

TStream object

TReader object

TWriter object

TFilerFlags type

See also

Unit

Classes

Declaration

```
type
    TFilerFlag = (ffInherited, ffChildPos);
    TFilerFlags = set of TFilerFlag;
```

Description

The TFilerFlags type indicates whether a component in a filer object's stream is a component in an inherited form and whether the component's creation order is important to its form. The ReadPrefix method gets a parameter of type TFilerFlags.

See also
ReadPrefix method

TFileStream object

See also Methods Objects

Unit

Classes

Description

The TFileStream object is a stream object that stores its data in a disk file. The file stream overrides only two methods it inherits from the basic stream type, TStream. The Create constructor takes two parameters, one which specifies the name of the file associated with the stream and one which specifies the file mode. The other overridden method is the Destroy destructor, which closes the associated file before disposing of the stream object.

Methods

Create	Destroy

See also

TStream object

THandleStream object

TFindMethodEvent type

See also

Unit

Classes

Declaration

```
type
    TFindMethodEvent = procedure(Reader: TReader; const MethodName: string;
      var Address: Pointer; var Error: Boolean) of object;
```

Description

The TFindMethodEvent type is the method-pointer type used as the type of the OnFindMethod event.

See also

OnFindMethod event

FindMethod method

TGetChildProc type

See also

Unit

Classes

Declaration

```
type
    TGetChildProc = procedure (Child: TComponent) of object;
```

Description

The TGetChildProc type defines a method-pointer type for the methods to be passed as parameters to the GetChildren method.

See also
GetChildren method

TGetStrProc type

See also Examples

Unit

Classes

Declaration

```
type
    TGetStrProc = procedure(const S: string) of object;
```

Description

The TGetStrProc type is the method-pointer type used as a parameter type to various procedures that generate lists of strings at run time. In general, the procedure declaration includes a single parameter named Proc, of type TGetStrProc. The procedure then iteratively calls Proc, passing each string to add to the list.

Examples

The GetCursorValues procedure in the Controls unit takes a parameter of type TGetStrProc, then calls the passed method to add the name of each cursor in the list of available cursors for the application.

Here is the declaration of GetCursorValues:

```
procedure GetCursorValues(Proc: TGetStrProc);
var
  I: Integer;
begin
  for I := Low(Cursors) to High(Cursors) do Proc(StrPas(Cursors[I].Name));
end;
```

The method for Proc is generally assigned by a property editor or other code. Your only interaction with such a method is to call the method once each for each string you want to add.

Another example of a TGetStrProc parameter is in the GetValues method of a property-editor object. Set elements appear in the Object Inspector as Boolean values, so the values defined by TSetElementProperty.GetValues are the strings 'True' and 'False':

```
procedure TSetElementProperty.GetValues(Proc: TGetStrProc);
begin
  Proc('False');
  Proc('True');
end;
```

See also
GetValues method
GetCursorValues procedure
GetColorValues procedure

THandleStream object

See also Properties Methods Objects

Unit

Classes

Description

The THandleStream object is a stream object that works much like a file stream object, but instead of specifying the name of a file to use to store the stream's data, you specify the file handle of an already-assigned file.

The handle-stream object defines a Handle property that provides read-only access to the file handle, which you can pass as a parameter to RTL functions that operate on files based on a handle. The handle stream also overrides the Create constructor to take a handle as a parameter, specifying the file to associate with the handle-stream object.

Property

Handle

Methods

Create	Seek	Write
Read		

See also

TStream object

TFileStream object

TIdentToInt type

See also

Unit

Classes

Declaration

```
type
    TIdentToInt = function (const Ident: string; var Int: Longint): Boolean;
```

Description

The TIdentToInt type defines a function type used when registering integer constants and their values with the streaming system.

See also

TIntToIdent type

IdentToColor function

IdentToCursor function

TIntToIdent type

See also

Unit

Classes

Declaration

```
type
    TIntToIdent = function (Int: Longint; var Ident: string): Boolean;
```

Description

The TIntToIdent type defines a function type used when registering integer constants and their values with the streaming system.

See also

TIdentToInt type

ColorToIdent function

CursorToIdent function

TList object (protected)

Methods Objects

Unit

Classes

Description

The TList object is a list of pointers. In addition to the items in the VCL reference for using the list object, there are several methods of interest to component writers.

You can derive your own, specialized list objects that change the behavior of the standard list. TList includes several protected methods that you can call or override in your descendant list objects.

Methods

<u>Error</u>   <u>Grow</u>

TMemoryStream object

See also Methods Objects

Unit

Classes

Description

The TMemoryStream object is a stream object that stores its data in dynamic memory. In addition to the items it inherits from TCustomMemoryStream, the memory stream has methods that allow it to load or replace its contents from either a disk file or another stream object. It also provides methods for writing to and clearing its memory, and for managing a dynamic memory pool.

Methods

Capacity	LoadFromFile	SetSize
Clear	LoadFromStream	Write
Destroy	Realloc	

See also
TStream object
TCustomMemoryStream object

TOperation type

See also

Unit

Classes

Declaration

```
type
    TOperation = (opInsert, opRemove);
```

Description

The TOperation type defines the two types of component operations flagged by the Notification method: inserting and removing. Notification takes a parameter of type TOperation that indicates whether the component passed in its other parameter is being inserted or removed.

See also
Notification method

TPersistent object

See also Methods Objects

Unit

Classes

Declaration

The TPersistent object is the abstract base object for all objects stored and loaded on Delphi stream objects. In addition to the methods it inherits from its ancestor, TObject, TPersistent defines three new methods: AssignTo and DefineProperties, which are protected, and Assign, which is public.

Methods

Assign	AssignTo	DefineProperties

See also

TPersistentClass type

TStream object

TPersistentClass type

See also

Unit

Classes

Declaration

```
type
    TPersistentClass = class of TPersistent;
```

Description

The TPersistentClass type is the class reference type for persistent objects.

See also
TPersistent object

TReadComponentsProc type

See also

Unit

Classes

Declaration

```
type
    TReadComponentsProc = procedure(Component: TComponent) of object;
```

Description

The TReadComponentsProc type is a method-pointer type used for the Proc parameter to the ReadComponents method of reader objects.

See also
ReadComponents method

TReader object

See also Properties Methods Events Objects

Unit

Classes

Description

The TReader object is a specialized filer object which reads data from its associated stream. In addition to the items it inherits from TFiler, TReader declares properties, methods, and events of its own.

The Owner and Parent properties serve as the owner and parent, respectively, of components read from the reader object's stream. The OnError, OnFindMethod, and OnSetName events allow your application to customize responses to certain occurrences when reading items.

In addition to overriding some of the methods inherited from TFiler, TReader defines numerous methds for reading different kinds of items from the associated stream and to trigger the reader object's events.

Properties

Owner	Parent	Position

Methods

BeginReferences	FlushBuffer	ReadListBegin
DefineBinaryProperty	NextValue	ReadListEnd
DefineProperty	Read	ReadPrefix
Destroy	ReadBoolean	ReadRootComponent
EndOfList	ReadChar	ReadSignature
EndReferences	ReadComponents	ReadStr
Error	ReadFloat	ReadString
FindMethod	ReadIdent	ReadValue
FixupReferences	ReadInteger	SetName

Events

OnError	OnFindMethod	OnSetName

See also

TFiler object

TWriter object

TReaderError type

See also

Unit

Classes

Declaration

```
type
    TReaderError = procedure(Reader: TReader; const Message: string;
      var Handled: Boolean) of object;
```

Description

The TReaderError type is the method-pointer type of the OnError event of reader objects.

See also

OnError event

TReader object

TReaderProc type

See also

Unit

Classes

Declaration

```
TReaderProc = procedure(Reader: TReader) of object;
```

Description

The TReaderProc type is the method-pointer type used for the ReadData parameter of the DefineProperty method. TReaderProc defines a method that reads a property value from the reader object passed in the Reader parameter.

See also

DefineProperty method

TReader object

TWriterProc type

TResourceStream object

See also Methods Objects

Unit

Classes

Description

The TResourceStream object is a memory stream that provides access to the resources in a Windows application. In addition to the items it inherits from TCustomMemoryStream, the resource stream defines constructors that associate the stream with the resources of a particular module or resource ID, and override the Write method to write to the resource file.

Methods

Create	Destroy	Write
CreateFromID		

See also
TCustomMemoryStream object

TSetNameEvent type

See also

Unit

Classes

Declaration

```
type
    TSetNameEvent = procedure(Reader: TReader; Component: TComponent;
      var Name: string) of object;
```

Description

The TSetNameEvent type is the method-pointer type used for the OnSetName event of reader objects.

See also

OnSetName event

SetName methodFor reader objects

TStream object

See also Properties Methods Objects

Unit

Classes

Description

The TStream object is an abstract object type representing a medium that can store binary data. There are several useful descendants of TStream that store their data in memory, in disk files, and so on.

TStream defines two properties, Size and Position, which represent the size of the stream in bytes and the current position, respectively. The methods defined by TStream provide for reading, writing, and copying bytes into and out of the stream.

Properties

Position	Size

Methods

CopyFrom	ReadResHeader	WriteComponent
Read	Seek	WriteComponentRes
ReadBuffer	Write	WriteDescendant
ReadComponent	WriteBuffer	WriteDescendantRes
ReadComponentRes		

See also

TFileStream object

THandleStream object

TMemoryStream object

TStreamProc type

See also

Unit

Classes

Declaration

```
TStreamProc = procedure(Stream: TStream) of object;
```

Description

The TStreamProc type is the method-pointer type used for the ReadData and WriteData parameters of the DefineBinaryProperty method of reader objects. TStreamProc defines a method that reads or writes a property value in binary form to the stream passed in the Stream parameter.

See also

DefineBinaryProperty method

TReader object

TValueType type

See also

Unit

Classes

Declaration

```
type
    TValueType = (vaNull, vaList, vaInt8, vaInt16, vaInt32, vaExtended,
  vaString, vaIdent,
      vaFalse, vaTrue, vaBinary, vaSet, vaLString, vaNil, vaCollection);
```

Description

The TValueType type defines the kinds of values written to and read from filer objects. Delphi's filers use a tagged value system, where each value on the stream is preceded by a prefix indicating its type. Those prefixes are of type TValueType.

See also

NextValue method

ReadValue method

TWriter object

See also Properties Methods Objects

Unit

Classes

Description

The TWriter object is a specialized filer object which writes data to its associated stream.

In addition to overriding some of the methods inherited from TFiler, TWriter defines numerous methds for writing different kinds of items to the associated stream.

Properties

Position	RootAncestor

Methods

DefineBinaryProperty	WriteChar	WriteListEnd
DefineProperty	WriteFloat	WriteRootComponent
Destroy	WriteIdent	WriteSignature
Write	WriteInteger	WriteStr
WriteBoolean	WriteListBegin	WriteString
WriteComponent		

See also

TFiler object

TReader object

TWriterProc type

See also

Unit

Classes

Declaration

```
type
    TWriterProc = procedure(Writer: TWriter) of object;
```

Description

The TWriterProc type is the method-pointer type used for the WriteData parameter of the DefineProperty method. TWriterProc defines a method that writes a property value to the writer object passed in the Writer parameter.

See also

DefineProperty method

TWriter object

TReaderProc type

UnRegisterClass procedure

See also

Unit

Classes

Declaration

```
procedure UnRegisterClass(AClass: TPersistentClass);
```

Description

The UnRegisterClass procedure removes the object type specified by the AClass parameter from the list of object types registered with the Delphi streaming system.

See also

RegisterClass procedure

UnRegisterClasses procedure

UnRegisterClasses procedure

See also

Unit

Classes

Declaration

```
procedure UnRegisterClasses(AClasses: array of TPersistentClass);
```

Description

The UnRegisterClasses procedure removes the object types specified by the AClasses array parameter from the list of object types registered with the Delphi streaming system. UnRegisterClasses calls UnRegisterClass once for each element in the array.

See also

RegisterClasses procedure

UnRegisterClass procedure

Updated method

See also

Applies to

TComponent (all components)

Declaration

```
procedure Updated; dynamic;
```

Description

The Updated method clears the csUpdating state in the component's ComponentState property to indicate that the component has finished updating. A call to Updated always follows a call to Updating, which sets the flag.

See also

ComponentState property

Updating method

Updating method

See also

Applies to

TComponent (all components)

Declaration

```
procedure Updating; dynamic;
```

Description

The Updating method sets the csUpdating state in the component's ComponentState property to indicate that the component is about to be updated. A call to Updating should always be followed by a call to Updated, which clears the flag.

See also

ComponentState property

Updated method

ValidateRename method

See also Example

Applies to

TComponent

Declaration

```
procedure ValidateRename(AComponent: TComponent; const CurName, NewName:
  string); virtual;
```

Description

The ValidateRename method checks to see if a component can rename one of its owned components, passed in AComponent, from its current name (CurName) to the string passed in NewName. If AComponent is **nil** or NewName is already the name of a component in the Components list, ValidateRename raises an EComponentError exception.

Example

The InsertComponent method of TComponent uses ValidateRename to determine whether it can insert a component with a particular name. ValidateRename raises an exception if the component already owns another component with the name of the component being inserted.

```
procedure TComponent.InsertComponent(AComponent: TComponent);
begin
  ValidateRename(AComponent, '', AComponent.FName^);    { validate the new name }
  Insert(AComponent);     { never executed if name was already in use }
  ...
end;
```

See also

Name property

FindComponent method

Write method

Applies to

TStream, TWriter

Description

There are two methods called Write that both perform similar functions. The one for stream objects writes data to the stream. The one for writer objects passes data to the writer object's associated stream.

Write method for stream objects

Write method for writer objects

Write method

See also

Applies to

TStream

Declaration

```
function Write(const Buffer; Count: Longint): Longint; virtual; abstract;
```

Description

The Write method writes Count bytes from Buffer onto the stream, advances the current position of the stream by Count bytes, and returns the number of bytes written.

TStream.Write is abstract. Each descendant stream type defines a Write method that writes data to its particular storage medium (such as memory or a disk file). All the other data-writing methods of a stream (WriteBuffer, WriteComponent) call Write to do their actual writing.

See also
Read method

Write method

See also

Applies to

TWriter

Declaration

```
procedure Write(const Buf; Count: Longint);
```

Description

The Write method writes Count bytes from Buf to the writer object's associated stream.

See also
Read method

WriteBoolean method

See also

Applies to

TWriter

Declaration

```
procedure WriteBoolean(Value: Boolean);
```

Description

The WriteBoolean method writes the Boolean value passed in Value to the writer object's stream.

See also

Write methodFor stream objects

ReadBoolean method

WriteBuffer method

See also

Applies to

TStream

Declaration

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

Description

The WriteBuffer method writes Count bytes from Buffer onto the stream and advances the current position of the stream by Count bytes. If the stream fails to write all the requested bytes, WriteBuffer raises an EWriteError exception.

See also
Write methodFor stream objects
ReadBuffer method

WriteChar method

See also

Applies to

TWriter

Declaration

```
procedure WriteChar(Value: Char);
```

Description

The WriteChar method writes the character passed in Value to the writer object's stream.

See also

Write methodFor stream objects

ReadChar method

WriteCollection method

See also

Applies to

TWriter

Declaration

```
procedure WriteCollection(Value: TCollection);
```

Description

The WriteCollection method writes the collection of objects passed in Value to the writer object's stream.

See also
Write methodFor writer objects

WriteComponent method

Applies to

TStream, TWriter

There are two different methods called WriteComponent. The WriteComponent for stream objects writes a specified component and any components it owns to the stream. The WriteComponent for writer objects has the specified component write the values of each of its properties to the writer object.

WriteComponent method for stream objects

WriteComponent method for writer objects

WriteComponent method

See also

Applies to

TStream

Declaration

```
procedure WriteComponent(Instance: TComponent);
```

Description

The WriteComponent method writes the component specified by Instance and any objects owned by Instance to the stream. WriteComponent constructs a writer object and calls its WriteRootComponent method to write Instance and its owned objects.

See also

WriteRootComponent method

ReadComponent method

WriteComponent method

See also

Applies to

TWriter

Declaration

```
procedure WriteComponent(Component: TComponent);
```

Description

The WriteComponent method calls the WriteState method of the component passed in Component, passing itself as the Writer parameter. That is, it calls Component.WriteState(Self). Before calling WriteState, WriteComponent sets the csWriting state in Component's ComponentState property, then clears it when WriteState returns.

See also

WriteState method

ReadComponent method

WriteComponentRes method

See also

Applies to

TStream

Declaration

```
procedure WriteComponentRes(const ResName: string; Instance: TComponent);
```

Description

The WriteComponentRes method writes a standard Windows resource-file header to the stream followed by the component specified by Instance. To read a component written with WriteComponentRes, call the ReadComponentRes method.

WriteComponentRes uses the string passed in ResName as the resource name for the resource-file header, then calls the WriteComponent method to write the component and its owned components.

See also

WriteComponent methodFor stream objects

ReadComponentRes method

WriteComponentResFile procedure

See also

Unit

Classes

Declaration

```
procedure WriteComponentResFile(const FileName: string; Instance:
  TComponent);
```

Description

The WriteComponentResFile procedure writes the component passed in Instance and any components it owns in Windows resource-file format to the file specified by FileName. For example, Delphi writes forms and their components into form files with WriteComponentResFile. Similarly, you can write your own applications that generate Delphi-readable form files by constructing form components and their components and writing them to a form file with WriteComponentResFile.

WriteComponentResFile creates a file-stream object with the name passed in FileName, then calls that stream's WriteComponentRes method to write Instance to the file.

See also

ReadComponentResFile function

WriteComponentRes method

WriteComponents method

See also

Description

The WriteComponents method used by Delphi 1.0 components has been replaced by the GetChildren method, which supports form inheritance.

Any component that had used WriteComponents must now implement GetChildren and SetChildOrder. Any child that returns True from HasParent must now implement GetParentComponent and SetParentComponent.

See also

GetChildren method

SetChildOrder method

HasParent method

GetParentComponent method

SetParentComponent method

WriteDescendant method

Applies to

TStream, TWriter

There are two different methods called WriteDescendant. The WriteDescendant for stream objects writes a specified component and any components it owns to the stream. The WriteDescendant for writer objects has the specified component write the values of each of its properties to the writer object's stream.

WriteDescendant method for stream objects

WriteDescendant method for writer objects

WriteDescendant method

See also

Applies to

TStream

Declaration

```
procedure WriteDescendant(Instance, Ancestor: TComponent);
```

Description

The WriteDescendant method for stream objects constructs a writer object, then calls the writer object's WriteDescendant method to write the component passed in Instance to the stream. Instance is either an inherited form descended from Ancestor or a component in an inherited form corresponding to the component Ancestor in the ancestor form.

See also

WriteDescendantRes method

WriteDescendant method for writer objects

WriteDescendant method

See also

Applies to

TWriter

Declaration

```
procedure WriteDescendant(Root: TComponent; AAncestor: TComponent);
```

Description

The WriteDescendant method sets the writer object's Ancestor and RootAncestor properties to the value passed in AAncestor, sets the Root property to the value passed in Root, then calls WriteSignature and WriteComponent to write Root and any components it owns to the writer object's stream.

WriteDescendant differs from WriteComponent in that, by setting Ancestor, it uses Ancestor's property values as the defaults, rather than those defined by Root's type.

See also

WriteComponent methodFor writer objects

WriteSignature method

Ancestor property

Root property

RootAncestor property

WriteDescendantRes method

See also

Applies to

TStream

Declaration

```
procedure WriteDescendantRes(const ResName: string; Instance, Ancestor:
  TComponent);
```

Description

The WriteDescendantRes method for stream objects writes a Windows resource header to the stream, using the resource name passed in ResName as the name of the resource. It then calls WriteDescendant to write Instance to the stream as a descendant of Ancestor.

See also
WriteDescendant method for stream objects

WriteFloat method

See also

Applies to

TWriter

Declaration

```
procedure WriteFloat(Value: Extended);
```

Description

The WriteFloat method writes the floating-point value passed in Value to the writer object's stream.

See also

Write methodFor stream objects

ReadFloat method

WriteIdent method

See also

Applies to

TWriter

Declaration

```
procedure WriteIdent(const Ident: string);
```

Description

The WriteIdent method writes the identifier passed in Ident to the writer object's stream.

See also

Write methodFor stream objects

ReadIdent method

WriteInteger method

See also

Applies to

TWriter

Declaration

```
procedure WriteInteger(Value: Longint);
```

Description

The WriteInteger method writes the integer value passed in Value to the writer object's stream.

See also

Write methodFor stream objects

ReadInteger method

WriteListBegin method

See also Example

Applies to

TWriter

Declaration

```
procedure WriteListBegin;
```

Description

The WriteListBegin method writes a start-of-list marker to the writer object's associated stream. Every call to WriteListBegin must have a corresponding call to WriteListEnd.

Example

The TStrings object uses WriteListBegin and WriteListEnd to store its strings. The following WriteData method is the one TStrings registers for storing its strings in its DefineProperties method:

```
procedure TStrings.WriteData(Writer: TWriter);
var
  I: Integer;
begin
  Writer.WriteListBegin;  { write start-of-list marker }
  for I := 0 to Count - 1 do      { iterate through the strings... }
    Writer.WriteString(Get(I));   ( ...writing each to the stream }
  Writer.WriteListEnd;    { write end-of-list marker }
end;
```

See also

Write methodFor stream objects

ReadListBegin method

WriteListEnd method

WriteListEnd method

See also Example

Applies to

TWriter

Declaration

```
procedure WriteListEnd;
```

Description

The WriteListEnd method writes an end-of-list marker to the writer object's associated stream. Calls to WriteListEnd must follow matching WriteListBegin calls.

See also

Write methodFor stream objects

ReadListEnd method

WriteListBegin method

WriteRootComponent method

See also

Applies to

TWriter

Declaration

```
procedure WriteRootComponent(Root: TComponent);
```

Description

The WriteRootComponent method sets the writer object's Root property to the component passed in Root, then calls WriteSignature to write the Delphi filer signature bytes to the stream, and finally calls the writer object's WriteComponent method to store Root on the stream.

See also
Root property
WriteSignature method
ReadRootComponent method

WriteSignature method

See also

Applies to

TWriter

Declaration

```
procedure WriteSignature;
```

Description

The WriteSignature method writes a Delphi filer signature to the writer object's associated stream. The WriteRootComponent method calls WriteSignature before writing its component to the stream. By checking for the signature before loading objects, reader objects can guard against inadvertantly reading invalid or corrupted data.

The filer signature is a four-character sequence. For this version of Delphi, the signature is 'TPF0'.

See also

Write methodFor stream objects

ReadSignature method

WriteState method

See also

Applies to

TComponent

Declaration

```
procedure WriteState(Writer: TWriter); virtual;
```

Description

The WriteState method writes the values of all the component's published properties, other stored data, and any owned components to the writer object passed in Writer.

See also
ReadState method

WriteStr method

See also

Applies to

TWriter

Declaration

```
procedure WriteStr(const Value: string);
```

Description

The WriteStr method writes the string passed in Value to the writer object's stream.

Caution: Always use ReadString and WriteString for reading and writing component strings to streams. The similarly-named ReadStr and WriteStr methods are for internal use by certain VCL components and will likely corrupt your data if you attempt to use them.

See also

Write methodFor stream objects

ReadStr method

WriteString method

WriteString method

See also

Applies to

TWriter

Declaration

```
procedure WriteString(const Value: string);
```

Description

The WriteString method writes the string passed in Value to the writer object's stream.

Caution: Always use ReadString and WriteString for reading and writing component strings to streams. The similarly-named ReadStr and WriteStr methods are for internal use by certain VCL components and will likely corrupt your data if you attempt to use them.

See also

Write methodFor stream objects

ReadString method

WriteStr method

Message record types

See also

Unit

Messages

Delphi provides message-record types for all the standard Windows messages.

TMessage	TWMHScrollClipboard	TWMNCRButtonUp
TWMActivate	TWMIconEraseBkgnd	TWMNextDlgCtl
TWMActivateApp	TWMInitDialog	TWMNoParams
TWMAskCBFormatName	TWMInitMenu	TWMPaint
TWMCancelMode	TWMInitMenuPopup	TWMPaintClipboard
TWMChangeCBChain	TWMKey	TWMPaintIcon
TWMChar	TWMKeyDown	TWMPaletteChanged
TWMCharToItem	TWMKeyUp	TWMPaletteIsChanging
TWMChildActivate	TWMKillFocus	TWMParentNotify
TWMChooseFont_GetLogFont	TWMLButtonDblClk	TWMPaste
TWMClear	TWMLButtonDown	TWMPower
TWMClose	TWMLButtonUp	TWMQueryDragIcon
TWMCommand	TWMMButtonDblClk	TWMQueryEndSession
TWMCommNotify	TWMMButtonDown	TWMQueryNewPalette
TWMCompacting	TWMMButtonUp	TWMQueryOpen
TWMCompareItem	TWMMDIActivate	TWMQueueSync
TWMCopy	TWMMDICascade	TWMQuit
TWMCreate	TWMMDICreate	TWMRButtonDblClk
TWMCtlColor	TWMMDIDestroy	TWMRButtonDown
TWMCut	TWMMDIGetActive	TWMRButtonUp
TWMDDE_Ack	TWMMDIIconArrange	TWMRenderAllFormats
TWMDDE_Advise	TWMMDIMaximize	TWMRenderFormat
TWMDDE_Data	TWMMDINext	TWMScroll
TWMDDE_Execute	TWMMDIRestore	TWMSetCursor
TWMDDE_Initiate	TWMMDISetMenu	TWMSetFocus
TWMDDE_Poke	TWMMDITile	TWMSetFont
TWMDDE_Request	TWMMeasureItem	TWMSetRedraw
TWMDDE_Terminate	TWMMenuChar	TWMSetText
TWMDDE_Unadvise	TWMMenuSelect	TWMShowWindow
TWMDeadChar	TWMMouse	TWMSize
TWMDeleteItem	TWMMouseActivate	TWMSizeClipboard
TWMDestroy	TWMMouseMove	TWMSpoolerStatus
TWMDestroyClipboard	TWMMove	TWMSysChar
TWMDevModeChange	TWMNCActivate	TWMSysColorChange
TWMDrawClipboard	TWMNCCalcSize	TWMSysCommand

TWMDrawItem	TWMNCCreate	TWMSysDeadChar
TWMDropFiles	TWMNCDestroy	TWMSysKeyDown
TWMEnable	TWMNCHitTest	TWMSysKeyUp
TWMEndSession	TWMNCHitMessage	TWMSystemError
TWMEnterIdle	TWMNCLButtonDblClk	TWMTimeChange
TWMEraseBkgnd	TWMNCLButtonDown	TWMTimer
TWMFontChange	TWMNCLButtonUp	TWMUndo
TWMGetDlgCode	TWMNCMButtonDblClk	TWMVKeyToItem
TWMGetFont	TWMNCMButtonDown	TWMVScroll
TWMGetMinMaxInfo	TWMNCMButtonUp	TWMVScrollClipboard
TWMGetText	TWMNCMouseMove	TWMWindowPosChanged
TWMGetTextLength	TWMNCPaint	TWMWindowPosChanging
TWMHelp	TWMNCRButtonDblClk	TWMWindowPosMsg
TWMHScroll	TWMNCRButtonDown	TWMWinIniChange

See also
What's In a Windows Message?

TMessage type

See also

Unit

Messages

Declaration

```
TMessage = record
    Msg: Word;
    case Integer of
      0: (
        WParam: Word;
        LParam: Longint;
        Result: Longint);
      1: (
        WParamLo: Byte;
        WParamHi: Byte;
        LParamLo: Word;
        LParamHi: Word;
        ResultLo: Word;
        ResultHi: Word);
  end;
```

Description

The TMessage type is the generic message-record type. All standard Windows messages and Delphi-defined messages have specific message-record types with named parameters. You can typecast any message record into type TMessage and use the generic parameter names to refer to the message's parameters if you prefer.

See also
<u>Message record types</u>

TWMActivate type

See also

Unit

Messages

Declaration

```
type
    TWMActivate = record
      Msg: Cardinal;
      Active: WORD;     { WA_INACTIVE, WA_ACTIVE, WA_CLICKACTIVE }
      ActiveWindow: HWND;
      Minimized: BOOL;
      Result: Longint;
    end;
```

Description

The TWMActivate type is the message record for the WM_ACTIVATE message.

TWMActivateApp type

See also

Unit

Messages

Declaration

```
type
    TWMActivateApp = record
      Msg: Cardinal;
      Active: BOOL;
      Task: THANDLE;
      Unused: Word;
      Result: Longint;
    end;
```

Description

The TWMActivateApp type is the message record for the WM_ACTIVATEAPP message.

TWMAskCBFormatName type

See also

Unit

Messages

Declaration

```
type
    TWMAskCBFormatName = record
      Msg: Cardinal;
      NameLen: Word;
      FormatName: PChar;
      Result: Longint;
    end;
```

Description

The TWMAskCBFormatName type is the message record for the WM_ASKCBFORMATNAME message.

TWMCancelMode type

See also

Unit

Messages

Declaration

```
type
    TWMCancelMode = TWMNoParams;
```

Description

The TWMCancelMode type is the message record for the WM_CANCELMODE message.

TWMChangeCBChain type

See also

Unit

Messages

Declaration

```
type
    TWMChangeCBChain = record
      Msg: Cardinal;
      Remove: HWND;
      Next: HWND;
      Unused: Word;
      Result: Longint;
    end;
```

Description

The TWMCancelMode type is the message record for the WM_CHANGECBCHAIN message.

TWMChar type

See also

Unit

Messages

Declaration

```
type
    TWMChar = TWMKey;
```

Description

The TWMChar type is the message record for the WM_CHAR message.

TWMCharToItem type

See also

Unit

Messages

Declaration

```
type
    TWMCharToItem = record
      Msg: Cardinal;
      Key: Word;
      ListBox: HWND;
      CaretPos: Word;
      Result: Longint;
    end;
```

Description

The TWMCharToItem type is the message record for the WM_CHARTOITEM message.

TWMChildActivate type

See also

Unit

Messages

Declaration

```
type
    TWMChildActivate = TWMNoParams;
```

Description

The TWMChildActivate type is the message record for the WM_CHILDACTIVATE message.

TWMChooseFont_GetLogFont type

See also

Unit

Messages

Declaration

```
type
    TWMChooseFont_GetLogFont = record
      Msg: Cardinal;
      Unused: Word;
      LogFont: PLogFont;
      Result: Longint;
    end;
```

Description

The TWMChooseFont_GetLogFont type is the message record for the WM_CHOOSEFONT_GETLOGFONT message.

TWMClear type

See also

Unit

Messages

Declaration

```
type
    TWMClear = TWMNoParams;
```

Description

The TWMClear type is the message record for the WM_CLEAR message.

TWMClose type

See also

Unit

Messages

Declaration

```
type
    TWMClose = TWMNoParams;
```

Description

The TWMClose type is the message record for the WM_CLOSE message.

TWMCommand type

See also

Unit

Messages

Declaration

```
type
    TWMCommand = record
      Msg: Cardinal;
      ItemID: Word;
      Ctl: HWND;
      NotifyCode: Word;
      Result: Longint;
    end;
```

Description

The TWMCommand type is the message record for the WM_COMMAND message.

TWMCommNotify type

See also

Unit

Messages

Declaration

```
type
    TWMCommNotify = record
      Msg: Cardinal;
      Device: Word;
      NotifyStatus: Word;
      Unused: Word;
      Result: Longint;
    end;
```

Description

The TWMCommNotify type is the message record for the WM_COMMNOTIFY message.

TWMCompacting type

See also

Unit

Messages

Declaration

```
type
    TWMCompacting = record
      Msg: Cardinal;
      CompactRatio: Word;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMCompacting type is the message record for the WM_COMPACTING message.

TWMCompareItem type

See also

Unit

Messages

Declaration

```
type
    TWMCompareItem = record
      Msg: Cardinal;
      Ctl: Word;
      CompareItemStruct: PCompareItemStruct;
      Result: Longint;
    end;
```

Description

The TWMCompareItem type is the message record for the WM_COMPAREITEM message.

TWMCopy type

See also

Unit

Messages

Declaration

```
type
    TWMCopy = TWMNoParams;
```

Description

The TWMCopy type is the message record for the WM_COPY message.

TWMCreate type

See also

Unit

Messages

Declaration

```
type
    TWMCreate = record
      Msg: Cardinal;
      Unused: Integer;
      CreateStruct: PCreateStruct;
      Result: Longint;
    end;
```

Description

The TWMCreate type is the message record for the WM_CREATE message.

TWMCtlColor type

See also

Unit

Messages

Declaration

```
type
    TWMCtlColor = record
      Msg: Cardinal;
      ChildDC: HDC;
      ChildWnd: HWND;
      CtlType: Integer;
      Result: Longint;
    end;
```

Description

The TWMCtlColor type is the message record for the WM_CTLCOLOR message.

TWMCut type

See also

Unit

Messages

Declaration

```
type
    TWMCut = TWMNoParams;
```

Description

The TWMCut type is the message record for the WM_CUT message.

TWMDDE_Ack type

See also

Unit

Messages

Declaration

```
type
    TWMDDE_Ack = record
      Msg: Cardinal;
      PostingApp: HWND;
      case Word of
        WM_DDE_INITIATE: (
          App: Word;
          Topic: Word;
          Result: Longint);
        WM_DDE_EXECUTE {and all others }: (
          Status: Word;
          case Word of
            WM_DDE_EXECUTE: (
              Commands: THANDLE);
            0 { all others }: (
              Item: Word));
    end;
```

Description

The TWMDDE_Ack type is the message record for the WM_DDE_ACK message.

TWMDDE_Advise type

See also

Unit

Messages

Declaration

```
type
    TWMDDE_Advise = record
      Msg: Cardinal;
      PostingApp: HWND;
      Options: THANDLE;
      Item: Word;
      Result: Longint;
    end;
```

Description

The TWMDDE_Advise type is the message record for the WM_DDE_ADVISE message.

TWMDDE_Data type

See also

Unit

Messages

Declaration

```
type
    TWMDDE_Data = record
      Msg: Cardinal;
      PostingApp: HWND;
      Data: THANDLE;
      Item: Word;
      Result: Longint;
    end;
```

Description

The TWMDDE_Data type is the message record for the WM_DDE_DATA message.

TWMDDE_Execute type

See also

Unit

Messages

Declaration

```
type
    TWMDDE_Execute = record
      Msg: Cardinal;
      PostingApp: HWND;
      Commands: THANDLE;
      Unused: Word;
      Result: Longint;
    end;
```

Description

The TWMDDE_Execute type is the message record for the WM_DDE_EXECUTE message.

TWMDDE_Initiate type

See also

Unit

Messages

Declaration

```
type
    TWMDDE_Initiate = record
      Msg: Cardinal;
      PostingApp: HWND;
      App: Word;
      Topic: Word;
      Result: Longint;
    end;
```

Description

The TWMDDE_Initiate type is the message record for the WM_DDE_INITIATE message.

TWMDDE_Poke type

See also

Unit

Messages

Declaration

```
type
  TWMDDE_Poke = record
    Msg: Cardinal;
    PostingApp: HWND;
    Data: THANDLE;
    Item: Word;
    Result: Longint;
  end;
```

Description

The TWMDDE_Poke type is the message record for the WM_DDE_POKE message.

TWMDDE_Request type

See also

Unit

Messages

Declaration

```
type
    TWMDDE_Request = record
      Msg: Cardinal;
      PostingApp: HWND;
      Format: Word;
      Item: Word;
      Result: Longint;
    end;
```

Description

The TWMDDE_Request type is the message record for the WM_DDE_REQUEST message.

TWMDDE_Terminate type

See also

Unit

Messages

Declaration

```
type
    TWMDDE_Terminate = record
      Msg: Cardinal;
      PostingApp: HWND;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMDDETerminate type is the message record for the WM_DDE_TERMINATE message.

TWMDDE_Unadvise type

See also

Unit

Messages

Declaration

```
type
    TWMDDE_Unadvise = TWMDDE_Advise;
```

Description

The TWMDDE_Unadvise type is the message record for the WM_DDE_UNADVISE message.

TWMDeadChar type

See also

Unit

Messages

Declaration

```
type
    TWMDeadChar = TWMChar;
```

Description

The TWMDeadChar type is the message record for the WM_DEADCHAR message.

TWMDeleteItem type

See also

Unit

Messages

Declaration

```
type
    TWMDeleteItem = record
      Msg: Cardinal;
      Ctl: HWND;
      DeleteItemStruct: PDeleteItemStruct;
      Result: Longint;
    end;
```

Description

The TWMDeleteItem type is the message record for the WM_DELETEITEM message.

TWMDestroy type

See also

Unit

Messages

Declaration

```
type
    TWMestroy = TWMNoParams;
```

Description

The TWMDestroy type is the message record for the WM_DESTROY message.

TWMDestroyClipboard type

See also

Unit

Messages

Declaration

```
type
    TWMDestroyClipboard = TWMNoParams;
```

Description

The TWMDestroyClipboard type is the message record for the WM_DESTROYCLIPBOARD message.

TWMDevModeChange type

See also

Unit

Messages

Declaration

```
type
    TWMDevModeChange = record
      Msg: Cardinal;
      Unused: Integer;
      Device: PChar;
      Result: Longint;
    end;
```

Description

The TWMDevModeChanged type is the message record for the WM_DEVMODECHANGE message.

TWMDrawClipboard type

See also

Unit

Messages

Declaration

```
type
    TWMDrawClipboard = TWMNoParams;
```

Description

The TWMDrawClipboard type is the message record for the WM_DRAWCLIPBOARD message.

TWMDrawItem type

See also

Unit

Messages

Declaration

```
type
    TWMDrawItem = record
      Msg: Cardinal;
      Ctl: HWND;
      DrawItemStruct: PDrawItemStruct;
      Result: Longint;
    end;
```

Description

The TWMDrawItem type is the message record for the WM_DRAWITEM message.

TWMDropFiles type

See also

Unit

Messages

Declaration

```
type
    TWMDropFiles = record
      Msg: Cardinal;
      Drop: THANDLE;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMDropFiles type is the message record for the WM_DROPFILES message.

TWMEnable type

See also

Unit

Messages

Declaration

```
type
    TWMEnable = record
      Msg: Cardinal;
      Enabled: BOOL;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMEnable type is the message record for the WM_ENABLE message.

TWMEndSession type

See also

Unit

Messages

Declaration

```
type
    TWMEndSession = record
      Msg: Cardinal;
      EndSession: BOOL;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMEndSession type is the message record for the WM_ENDSESSION message.

TWMEnterIdle type

See also

Unit

Messages

Declaration

```
type
    TWMEnterIdle = record
      Msg: Cardinal;
      Source: Word;
      IdleWnd: HWND;
      Unused: Word;
      Result: Longint;
    end;
```

Description

The TWMEnterIdle type is the message record for the WM_ENTERIDLE message.

TWMEraseBkgnd type

See also

Unit

Messages

Declaration

```
type
    TWMEraseBkgnd = record
      Msg: Cardinal;
      DC: HDC;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMEraseBkgnd type is the message record for the WM_ERASEBKGND message.

TWMFontChange type

See also

Unit

Messages

Declaration

```
type
    TWMFontChange = TWMNoParams;
```

Description

The TWMFontChange type is the message record for the WM_FONTCHANGE message.

TWMGetDlgCode type

See also

Unit

Messages

Declaration

```
type
    TWMGetDlgCode = TWMNoParams;
```

Description

The TWMGetDlgCode type is the message record for the WM_GETDLGCODE message.

TWMGetFont type

See also

Unit

Messages

Declaration

```
type
    TWMGetFont = TWMNoParams;
```

Description

The TWMGetFont type is the message record for the WM_GETFONT message.

TWMGetMinMaxInfo type

See also

Unit

Messages

Declaration

```
type
    TWMGetMinMaxInfo = record
      Msg: Cardinal;
      Unused: Integer;
      MinMaxInfo: PMinMaxInfo;
      Result: Longint;
    end;
```

Description

The TWMGetMinMaxInfo type is the message record for the WM_GETMINMAXINFO message.

TWMGetText type

See also

Unit

Messages

Declaration

```
type
    TWMGetText = record
      Msg: Cardinal;
      TextMax: Integer;
      Text: PChar;
      Result: Longint;
    end;
```

Description

The TWMGetText type is the message record for the WM_GETTEXT message.

TWMGetTextLength type

See also

Unit

Messages

Declaration

```
type
    TWMGetTextLength = TWMNoParams;
```

Description

The TWMGetTextLength type is the message record for the WM_GETTEXTLENGTH message.

TWMHelp type

See also

Unit

Messages

Declaration

```
type
    TWMHelp = record
      Msg: Cardinal;
      Unused: Integer;
      HelpInfo: PHelpInfo;
      Result: Longint;
    end;
```

Description

The TWMHelp type is the message record for the WM_HELP message.

TWMHScroll type

See also

Unit

Messages

Declaration

```
type
    TWMHScroll = TWMScroll;
```

Description

The TWMHScroll type is the message record for the WM_HSCROLL message.

TWMHScrollClipboard type

See also

Unit

Messages

Declaration

```
type
    TWMHScrollClipboard = record
      Msg: Cardinal;
      Viewer: HWND;
      ScrollCode: Word;
      Pos: Word;
      Result: Longint;
    end;
```

Description

The TWMHScrollClipboard type is the message record for the WM_HSCROLLCLIPBOARD message.

TWMIconEraseBkgnd type

See also

Unit

Messages

Declaration

```
type
    TWMIconEraseBkgnd = TWMEraseBkgnd;
```

Description

The TWMIconEraseBkgnd type is the message record for the WM_ICONERASEBKGND message.

TWMInitDialog type

See also

Unit

Messages

Declaration

```
type
    TWMInitDialog = record
      Msg: Cardinal;
      Focus: HWND;
      InitParam: Longint;
      Result: Longint;
    end;
```

Description

The TWMInitDialog type is the message record for the WM_INITDIALOG message.

TWMInitMenu type

See also

Unit

Messages

Declaration

```
type
    TWMInitMenu = record
      Msg: Cardinal;
      Menu: HMENU;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMInitMenu type is the message record for the WM_INITMENU message.

TWMInitMenuPopup type

See also

Unit

Messages

Declaration

```
type
    TWMInitMenuPopup = record
      Msg: Cardinal;
      MenuPopup: HMENU;
      Pos: SmallInt;
      SystemMenu: BOOL;
      Result: Longint;
    end;
```

Description

The TWMInitMenuPopup type is the message record for the WM_INITMENUPOPUP message.

TWMKey type

See also

Unit

Messages

Declaration

```
type
    TWMKey = record
      Msg: Cardinal;
      CharCode: Word;
      KeyData: Longint;
      Result: Longint;
    end;
```

Description

The TWMKey type is the message record for all Windows keyboard messages. Both TWMKeyDown and TWMKeyUp are declared as exactly TWMKey.

TWMKeyDown type

See also

Unit

Messages

Declaration

```
type
    TWMKeyDown = TWMKey;
```

Description

The TWMKeyDown type is the message record for the WM_KEYDOWN message.

TWMKeyUp type

See also

Unit

Messages

Declaration

```
type
    TWMKeyUp = TWMKey;
```

Description

The TWMKeyUp type is the message record for the WM_KEYUP message.

TWMKillFocus type

See also

Unit

Messages

Declaration

```
type
    TWMKillFocus = record
      Msg: Cardinal;
      FocusedWnd: HWND;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMKillFocus type is the message record for the WM_KILLFOCUS message.

TWMLButtonDblClk type

See also

Unit

Messages

Declaration

```
type
    TWMLButtonDblClk = TWMMouse;
```

Description

The TWMLButtonDblClk type is the message record for the WM_LBUTTONDBLCLK message.

TWMLButtonDown type

See also

Unit

Messages

Declaration

```
type
    TWMLButtonDown = TWMMouse;
```

Description

The TWMLButtonDown type is the message record for the WM_LBUTTONDOWN message.

TWMLButtonUp type

See also

Unit

Messages

Declaration

```
type
    TWMLButtonUp = TWMMouse;
```

Description

The TWMLButtonUp type is the message record for the WM_LBUTTONUP message.

TWMMButtonDblClk type

See also

Unit

Messages

Declaration

```
type
    TWMMButtonDblClk = TWMMouse;
```

Description

The TWMMButtonDblClk type is the message record for the WM_MBUTTONDBLCLK message.

TWMMButtonDown type

See also

Unit

Messages

Declaration

```
type
    TWMMButtonDown = TWMMouse;
```

Description

The TWMMButtonDown type is the message record for the WM_MBUTTONDOWN message.

TWMMButtonUp type

See also

Unit

Messages

Declaration

```
type
    TWMMButtonUp = TWMMouse;
```

Description

The TWMMButtonUp type is the message record for the WM_MBUTTONUP message.

TWMMDIActivate type

See also

Unit

Messages

Declaration

```
type
    TWMMDIActivate = record
      Msg: Cardinal;
      Activate: BOOL;
      ActiveWnd: HWND;
      DeactiveWnd: HWND;
      Result: Longint;
    end;
```

Description

The TWMMDIActivate type is the message record for the WM_MDIACTIVATE message.

TWMMDICascade type

See also

Unit

Messages

Declaration

```
type
    TWMMDICascade = record
      Msg: Cardinal;
      Cascade: Word;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMMDICascade type is the message record for the WM_MDICASCADE message.

TWMMDICreate type

See also

Unit

Messages

Declaration

```
type
    TWMMDICreate = record
      Msg: Cardinal;
      Unused: Integer;
      MDICreateStruct: PMDICreateStruct;
      Result: Longint;
    end;
```

Description

The TWMMDICreate type is the message record for the WM_MDICREATE message.

TWMMDIDestroy type

See also

Unit

Messages

Declaration

```
type
    TWMMDIDestroy = record
      Msg: Cardinal;
      Child: HWND;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMMDIDestroy type is the message record for the WM_MDIDESTROY message.

TWMMDIGetActive type

See also

Unit

Messages

Declaration

```
type
    TWMMDIGetActive = TWMNoParams;
```

Description

The TWMMDIGetActive type is the message record for the WM_MDIGETACTIVE message.

TWMMDIIconArrange type

See also

Unit

Messages

Declaration

```
type
    TWMMDIIconArrange = TWMNoParams;
```

Description

The TWMMDIIconArrange type is the message record for the WM_MDIICONARRANGE message.

TWMMDIMaximize type

See also

Unit

Messages

Declaration

```
type
    TWMMDIMaximize = record
      Msg: Cardinal;
      Maximize: HWND;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMMDIMaximize type is the message record for the WM_MDIMAXIMIZE message.

TWMMDINext type

See also

Unit

Messages

Declaration

```
type
    TWMMDINext = record
      Msg: Cardinal;
      Child: HWND;
      Next: Longint;
      Result: Longint;
    end;
```

Description

The TWMMDINext type is the message record for the WM_MDINEXT message.

TWMMDIRestore type

See also

Unit

Messages

Declaration

```
type
    TWMMDIRestore = record
      Msg: Cardinal;
      IDChild: HWND;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMMDIRestore type is the message record for the WM_MDIRESTORE message.

TWMMDISetMenu type

See also

Unit

Messages

Declaration

```
type
    TWMMDISetMenu = record
      Msg: Cardinal;
      MenuFrame: HMENU;
      MenuWindow: HMENU;
      Result: Longint;
    end;
```

Description

The TWMMDISetMenu type is the message record for the WM_MDISETMENU message.

TWMMDITile type

See also

Unit

Messages

Declaration

```
type
    TWMMDITile = record
      Msg: Cardinal;
      Tile: Word;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMMDITile type is the message record for the WM_MDITILE message.

TWMMeasureItem type

See also

Unit

Messages

Declaration

```
type
    TWMMeasureItem = record
      Msg: Cardinal;
      IDCtl: Word;
      MeasureItemStruct: PMeasureItemStruct;
      Result: Longint;
    end;
```

Description

The TWMMeasureItem type is the message record for the WM_MEASUREITEM message.

TWMMenuChar type

See also

Unit

Messages

Declaration

```
type
    TWMMenuChar = record
      Msg: Cardinal;
      User: Char;
      Unused: Byte;
      MenuFlag: Word;
      Menu: HMENU;
      Result: Longint;
    end;
```

Description

The TWMMenuChar type is the message record for the WM_MENUCHAR message.

TWMMenuSelect type

See also

Unit

Messages

Declaration

```
type
    TWMMenuSelect = record
      Msg: Cardinal;
      IDItem: Word;
      MenuFlag: Word;
      Menu: HMENU;
      Result: Longint;
    end;
```

Description

The TWMMenuSelect type is the message record for the WM_MENUSELECT message.

TWMMouse type

See also

Unit

Messages

Declaration

```
type
    TWMMouse = record
      Msg: Cardinal;
      Keys: Word;
      case Integer of
        0: (
          XPos: Integer;
          YPos: Integer);
        1: (
          Pos: TPoint;
          Result: Longint);
    end;
```

Description

The TWMMouse type is the message record for all mouse messages.

TWMMouseActivate type

See also

Unit

Messages

Declaration

```
type
    TWMMouseActivate = record
      Msg: Cardinal;
      TopLevel: HWND;
      HitTestCode: Word;
      MouseMsg: Word;
      Result: Longint;
    end;
```

Description

The TWMMouseActivate type is the message record for the WM_MOUSEACTIVATE message.

TWMMouseMove type

See also

Unit

Messages

Declaration

```
type
    TWMMouseMove = TWMMouse;
```

Description

The TWMMouseMove type is the message record for the WM_MOUSEMOVE message.

TWMMove type

See also

Unit

Messages

Declaration

```
type
    TWMMove = record
      Msg: Cardinal;
      Unused: Integer;
      case Integer of
        0: (
          XPos: SmallInt;
          YPos: SmallInt);
        1: (
          Pos: TPoint;
          Result: Longint);
    end;
```

Description

The TWMMove type is the message record for the WM_MOVE message.

TWMNCActivate type

See also

Unit

Messages

Declaration

```
type
    TWMNCActivate = record
      Msg: Cardinal;
      Active: BOOL;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMNCActivate type is the message record for the WM_NCACTIVATE message.

TWMNCCalcSize type

See also

Unit

Messages

Declaration

```
type
    TWMNCCalcSize = record
      Msg: Cardinal;
      CalcValidRects: BOOL;
      CalcSize_Params: PNCCalcSize_Params;
      Result: Longint;
    end;
```

Description

The TWMNCCalcSize type is the message record for the WM_NCCALCSIZE message.

TWMNCCreate type

See also

Unit

Messages

Declaration

```
type
    TWMNCCreate = record
      Msg: Cardinal;
      Unused: Integer;
      CreateStruct: PCreateStruct;
      Result: Longint;
    end;
```

Description

The TWMNCCreate type is the message record for the WM_NCCREATE message.

TWMNCDestroy type

See also

Unit

Messages

Declaration

```
type
    TWMNCDestroy = TWMNoParams;
```

Description

The TWMNCDestroy type is the message record for the WM_NCDESTROY message.

TWMNCHitTest type

See also

Unit

Messages

Declaration

```
type
    TWMNCHitTest = record
      Msg: Cardinal;
      Unused: Cardinal;
      case Integer of
        0: (
          XPos: SmallInt;
          YPos: SmallInt);
        1: (
          Pos: TSmallPoint;
          Result: Longint);
    end;
```

Description

The TWMNCHitTest type is the message record for the WM_NCHITTEST message.

TWMNCHitMessage type

See also

Unit

Messages

Declaration

```
type
    TWMNCHitMessage = record
      Msg: Cardinal;
      HitTest: Integer;
      XCursor: SmallInt;
      YCursor: SmallInt;
      Result: Longint;
    end;
```

Description

The TWMNCHitMessage type is the message record for all non-client area mouse messages.

TWMNCLButtonDblClk type

See also

Unit

Messages

Declaration

```
type
    TWMNCLButtonDblClk  = TWMNCHitMessage;
```

Description

The TWMNCLButtonDblClk type is the message record for the WM_NCLBUTTONDBLCLK message.

TWMNCLButtonDown type

See also

Unit

Messages

Declaration

```
type
    TWMNCLButtonDown = TWMNCHitMessage;
```

Description

The TWMNCLButtonDown type is the message record for the WM_NCLBUTTONDOWN message.

TWMNCLButtonUp type

See also

Unit

Messages

Declaration

```
type
    TWMNCLButtonUp = TWMNCHitMessage;
```

Description

The TWMNCLButtonUp type is the message record for the WM_NCLBUTTONUP message.

TWMNCMButtonDblClk type

See also

Unit

Messages

Declaration

```
type
    TWMNCMButtonDblClk = TWMNCHitMessage;
```

Description

The TWMNCMButtonDblClk type is the message record for the WM_NCMBUTTONDBLCLK message.

TWMNCMButtonDown type

See also

Unit

Messages

Declaration

```
type
    TWMNCMButtonDown = TWMNCHitMessage;
```

Description

The TWMNCMButtonDown type is the message record for the WM_NCMBUTTONDOWN message.

TWMNCMButtonUp type

See also

Unit

Messages

Declaration

```
type
    TWMNCMButtonUp = TWMNCHitMessage;
```

Description

The TWMNCMButtonUp type is the message record for the WM_NCMBUTTONUP message.

TWMNCMouseMove type

See also

Unit

Messages

Declaration

```
type
    TWMNCMouseMove = TWMNCHitMessage;
```

Description

The TWMNCMouseMove type is the message record for the WM_NCMOUSEMOVE message.

TWMNCPaint type

See also

Unit

Messages

Declaration

```
type
    TWMNCPaint = TWMNoParams;
```

Description

The TWMNCPaint type is the message record for the WM_NCPaint message.

TWMNCRButtonDblClk type

See also

Unit

Messages

Declaration

```
type
    TWMNCRButtonDblClk = TWMNCHitMessage;
```

Description

The TWMNCRButtonDblClk type is the message record for the WM_NCRBUTTONDBLCLK message.

TWMNCRButtonDown type

See also

Unit

Messages

Declaration

```
type
    TWMNCRButtonDown = TWMNCHitMessage;
```

Description

The TWMNCRButtonDown type is the message record for the WM_NCRBUTTONDOWN message.

TWMNCRButtonUp type

See also

Unit

Messages

Declaration

```
type
    TWMNCRButtonUp = TWMNCHitMessage;
```

Description

The TWMNCRButtonUp type is the message record for the WM_NCRBUTTONUP message.

TWMNextDlgCtl type

See also

Unit

Messages

Declaration

```
type
    TWMNextDlgCtl = record
      Msg: Cardinal;
      CtlFocus: Word;
      Handle: BOOL;
      Unused: Word;
      Result: Longint;
    end;
```

Description

The TWMNextDlgCtl type is the message record for the WM_NEXTDLGCTL message.

TWMNoParams type

See also

Unit

Messages

Declaration

```
type
    TWMNoParams = record
      Msg: Cardinal;
      Unused: array[0..2] of Word;
      Result: Longint;
    end;
```

Description

The TWMNoParams type is the message record for all messages that do not use any of their parameters.

TWMPaint type

See also

Unit

Messages

Declaration

```
type
    TWMPaint = record
      Msg: Cardinal;
      DC: HDC;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMPaint type is the message record for the WM_PAINT message.

TWMPaintClipboard type

See also

Unit

Messages

Declaration

```
type
    TWMPaintClipboard = record
      Msg: Cardinal;
      Viewer: HWND;
      PaintStruct: THANDLE;
      Unused: Word;
      Result: Longint;
    end;
```

Description

The TWMPaintClipboard type is the message record for the WM_PAINTCLIPBOARD message.

TWMPaintIcon type

See also

Unit

Messages

Declaration

```
type
    TWMPaintIcon = TWMNoParams;
```

Description

The TWMPaintIcon type is the message record for the WM_PAINTICON message.

TWMPaletteChanged type

See also

Unit

Messages

Declaration

```
type
    TWMPaletteChanged = record
      Msg: Cardinal;
      PalChg: HWND;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMPaletteChanged type is the message record for the WM_PALETTECHANGED message.

TWMPaletteIsChanging type

See also

Unit

Messages

Declaration

```
type
    TWMPaletteIsChanging = record
      Msg: Cardinal;
      Realize: HWND;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMPaletteIsChanging type is the message record for the WM_PALETTEISCHANGING message.

TWMParentNotify type

See also

Unit

Messages

Declaration

```
type
    TWMParentNotify = record
      Msg: Cardinal;
      case Event: Word of
        WM_CREATE, WM_DESTROY: (
          ChildWnd: HWND;
          ChildID: Word);
        WM_LBUTTONDOWN, WM_MBUTTONDOWN, WM_RBUTTONDOWN: (
          XPos: Integer;
          YPos: Integer);
        0 { Name is MS DOC }: (
          Value1: Word;
          Value2: Word;
          Result: Longint);
    end;
```

Description

The TWMParentNotify type is the message record for the WM_PARENTNOTIFY message.

TWMPaste type

See also

Unit

Messages

Declaration

```
type
    TWMPaste = TWMNoParams;
```

Description

The TWMPaste type is the message record for the WM_PASTE message.

TWMPower type

See also

Unit

Messages

Declaration

```
type
    TWMPower = record
      Msg: Cardinal;
      PowerEvt: Word;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMPower type is the message record for the WM_POWER message.

TWMQueryDragIcon type

See also

Unit

Messages

Declaration

```
type
    TWMQueryDragIcon = TWMNoParams;
```

Description

The TWMQueryDragIcon type is the message record for the WM_QUERYDRAGICON message.

TWMQueryEndSession type

See also

Unit

Messages

Declaration

```
type
    TWMQueryEndSession = TWMNoParams;
```

Description

The TWMQueryEndSession type is the message record for the WM_QUERYENDSESSION message.

TWMQueryNewPalette type

See also

Unit

Messages

Declaration

```
type
    TWMQueryNewPalette = TWMNoParams;
```

Description

The TWMQueryNewPalette type is the message record for the WM_QUERYNEWPALETTE message.

TWMQueryOpen type

See also

Unit

Messages

Declaration

```
type
    TWMQueryOpen = TWMNoParams;
```

Description

The TWMQueryOpen type is the message record for the WM_QUERYOPEN message.

TWMQueueSync type

See also

Unit

Messages

Declaration

```
type
    TWMQueueSync = TWMNoParams;
```

Description

The TWMQueueSync type is the message record for the WM_QUEUESYNC message.

TWMQuit type

See also

Unit

Messages

Declaration

```
type
    TWMQuit = record
      Msg: Cardinal;
      ExitCode: Word;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMQuit type is the message record for the WM_QUIT message.

TWMRButtonDblClk type

See also

Unit

Messages

Declaration

```
type
    TWMRButtonDblClk = TWMMouse;
```

Description

The TWMRButtonDblClk type is the message record for the WM_RBUTTONDBLCLK message.

TWMRButtonDown type

See also

Unit

Messages

Declaration

```
type
    TWMRButtonDown = TWMMouse;
```

Description

The TWMRButtonDown type is the message record for the WM_RBUTTONDOWN message.

TWMRButtonUp type

See also

Unit

Messages

Declaration

```
type
    TWMMRButtonUp = TWMMouse;
```

Description

The TWMRButtonUp type is the message record for the WM_RBUTTONUP message.

TWMRenderAllFormats type

See also

Unit

Messages

Declaration

```
type
    TWMRenderAllFormats = TWMNoParams;
```

Description

The TWMRenderAllFormats type is the message record for the WM_RENDERALLFORMATS message.

TWMRenderFormat type

See also

Unit

Messages

Declaration

```
type
    TWMRenderFormat = record
      Msg: Cardinal;
      Format: Word;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMRenderFormat type is the message record for the WM_RENDERFORMAT message.

TWMScroll type

See also

Unit

Messages

Declaration

```
type
    TWMScroll = record
      Msg: Cardinal;
      ScrollCode: SmallInt;
      Pos: SmallInt;
      ScrollBar: HWND;
      Result: Longint;
    end;
```

Description

The TWMScroll type is the message record for all scrolling messages.

TWMSetCursor type

See also

Unit

Messages

Declaration

```
type
    TWMSetCursor = record
      Msg: Cardinal;
      CursorWnd: HWND;
      HitTest: Word;
      MouseMsg: Word;
      Result: Longint;
    end;
```

Description

The TWMSetCursor type is the message record for the WM_SETCURSOR message.

TWMSetFocus type

See also

Unit

Messages

Declaration

```
type
    TWMSetFocus = record
      Msg: Cardinal;
      FocusedWnd: HWND;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMSetFocus type is the message record for the WM_SETFOCUS message.

TWMSetFont type

See also

Unit

Messages

Declaration

```
type
    TWMSetFont = record
      Msg: Cardinal;
      Font: HFONT;
      Redraw: BOOL;
      Unused: Word;
      Result: Longint;
    end;
```

Description

The TWMSetFont type is the message record for the WM_SETFONT message.

TWMSetRedraw type

See also

Unit

Messages

Declaration

```
type
    TWMSetRedraw = record
      Msg: Cardinal;
      Redraw: Word;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMSetRedraw type is the message record for the WM_SETREDRAW message.

TWMSetText type

See also

Unit

Messages

Declaration

```
type
    TWMSetText = record
      Msg: Cardinal;
      Unused: Integer;
      Text: PChar;
      Result: Longint;
    end;
```

Description

The TWMSetText type is the message record for the WM_SETTEXT message.

TWMShowWindow type

See also

Unit

Messages

Declaration

```
type
    TWMShowWindow = record
      Msg: Cardinal;
      Show: BOOL;
      Status: Word;
      Unused: Word;
      Result: Longint;
    end;
```

Description

The TWMShowWindow type is the message record for the WM_SHOWWINDOW message.

TWMSize type

See also

Unit

Messages

Declaration

```
type
    TWMSize = record
      Msg: Cardinal;
      SizeType: Word;
      Width: Word;
      Height: Word;
      Result: Longint;
    end;
```

Description

The TWMSize type is the message record for the WM_SIZE message.

TWMSizeClipboard type

See also

Unit

Messages

Declaration

```
type
    TWMSizeClipboard = record
      Msg: Cardinal;
      Viewer: HWND;
      RC: THandle;
      Unused: Word;
      Result: Longint;
    end;
```

Description

The TWMSizeClipboard type is the message record for the WM_SIZECLIPBOARD message.

TWMSpoolerStatus type

See also

Unit

Messages

Declaration

```
type
    TWMSpoolerStatus = record
      Msg: Cardinal;
      JobStatus: Word;
      JobsLeft: Word;
      Unused: Word;
      Result: Longint;
    end;
```

Description

The TWMSpoolerStatus type is the message record for the WM_SPOOLERSTATUS message.

TWMSysChar type

See also

Unit

Messages

Declaration

```
type
    TWMSysChar = TWMKey;
```

Description

The TWMKey type is the message record for the WM_SYSCHAR message.

TWMSysColorChange type

See also

Unit

Messages

Declaration

```
type
    TWMSysColorChange = TWMNoParams;
```

Description

The TWMSysColorChange type is the message record for the WM_SYSCOLORCHANGE message.

TWMSysCommand type

See also

Unit

Messages

Declaration

```
type
    TWMSysCommand = record
      Msg: Cardinal;
      case CmdType: Word of
        SC_HOTKEY: (
          ActivateWnd: HWND;
          Unused1: Word);
        SC_KEYMENU: (
          Key: Word;
          Unused2: Word);
        SC_CLOSE, SC_HSCROLL, SC_MAXIMIZE, SC_MINIMIZE, SC_MOUSEMENU,
  SC_MOVE,
        SC_NEXTWINDOW, SC_PREVWINDOW, SC_RESTORE, SC_SCREENSAVE, SC_SIZE,
        SC_TASKLIST, SC_VSCROLL: (
          XPos: SmallInt;
          YPos: SmallInt;
          Result: Longint);
    end;
```

Description

The TWMSysCommand type is the message record for the WM_SYSCOMMAND message.

TWMSysDeadChar type

See also

Unit

Messages

Declaration

```
type
    TWMSysDeadChar = record
      Msg: Cardinal;
      DeadKey: Char;
      Unused: Byte;
      RepeatCount: Integer;
      AutoRepeat: Word;
      Result: Longint;
    end;
```

Description

The TWMSysDeadChar type is the message record for the WM_SYSDEADCHAR message.

TWMSysKeyDown type

See also

Unit

Messages

Declaration

```
type
    TWMSysKeyDown = TWMKey;
```

Description

The TWMSysKeyDown type is the message record for the WM_SYSKEYDOWN message.

TWMSysKeyUp type

See also

Unit

Messages

Declaration

```
type
    TWMSysKeyUp = TWMKey;
```

Description

The TWMSysKeyUp type is the message record for the WM_SYSKEYUP message.

TWMSystemError type

See also

Unit

Messages

Declaration

```
type
    TWMSystemError = record
      Msg: Cardinal;
      ErrSpec: Word;
      Unused: Longint;
      Result: Longint;
    end;
```

Description

The TWMSystemError type is the message record for the WM_SYSTEMERROR message.

TWMTimeChange type

See also

Unit

Messages

Declaration

```
type
    TWMTimeChange = TWMNoParams;
```

Description

The TWMTimeChange type is the message record for the WM_TIMECHANGE message.

TWMTimer type

See also

Unit

Messages

Declaration

```
type
    TWMTimer = record
      Msg: Cardinal;
      TimerID: Word;
      TimerProc: TFarProc;
      Result: Longint;
    end;
```

Description

The TWMTimer type is the message record for the WM_TIMER message.

TWMUndo type

See also

Unit

Messages

Declaration

```
type
    TWMUndo = TWMNoParams;
```

Description

The TWMUndo type is the message record for the WM_UNDO message.

TWMVKeyToItem type

See also

Unit

Messages

Declaration

```
type
    TWMVKeyToItem = TWMCharToItem;
```

Description

The TWMVKeyToItem type is the message record for the WM_VKEYTOITEM message.

TWMVScroll type

See also

Unit

Messages

Declaration

```
type
    TWMVScroll = TWMScroll;
```

Description

The TWMVScroll type is the message record for the WM_VSCROLL message.

TWMVScrollClipboard type

See also

Unit

Messages

Declaration

```
type
    TWMVScrollClipboard = record
      Msg: Cardinal;
      Viewer: HWND;
      ScollCode: Word;
      ThumbPos: Word;
      Result: Longint;
    end;
```

Description

The TWMVScrollClipboard type is the message record for the WM_CSCROLLCLIPBOARD message.

TWMWindowPosMsg type

See also

Unit

Messages

Declaration

```
type
    TWMWindowPosMsg = record
      Msg: Cardinal;
      Unused: Integer;
      WindowPos: PWindowPos;
      Result: Longint;
    end;
```

Description

The TWMWindowPosMsg type is the message record for window-position messages. Both TWMWindowPosChanged and TWMWindowPosChanging are declared as being of type TWMWindowPosMsg.

TWMWindowPosChanged type

See also

Unit

Messages

Declaration

```
type
    TWMWindowPosChanged = TWMWindowPosMsg;
```

Description

The TWMWindowPosChanged type is the message record for the WM_WINDOWPOSCHANGED message.

TWMWindowPosChanging type

See also

Unit

Messages

Declaration

```
type
    TWMWindowPosChanging = TWMWindowPosMsg;
```

Description

The TWMWindowPosChanging type is the message record for the WM_WINDOWPOSCHANGING message.

TWMWinIniChange type

See also

Unit

Messages

Declaration

```
type
    TWMWinIniChange = record
      Msg: Cardinal;
      Unused: Integer;
      Section: PChar;
      Result: Longint;
    end;
```

Description

The TWMWinIniChange type is the message record for the WM_WININICHANGE message.

BoxRect method

Applies to
TCustomGrid

Declaration
```
function BoxRect(ALeft, ATop, ARight, ABottom: Longint): TRect;
```

Description
The BoxRect method returns the screen pixel rectangle of a box of grid cells. The ALeft and ATop parameters specify the column and row numbers of the left, top corner of the box. The ARight and ABottom parameters specify the column and row numbers of the right, bottom corner of the box.

If the specified box is only partially visible in the grid, the returned rectangle contains the pixel dimensions of only the visible portion of the box. If the specified box is not visible at all, the returned rectangle is empty.

CanEditAcceptKey method

See Also

Applies to

TCustomGrid

Declaration

```
function CanEditAcceptKey(Key: Char): Boolean; dynamic;
```

Description

The CanEditAcceptKey method determines if the grid cell being edited can accept a key pressed by the user. CanEditAcceptKey is called when the user presses a key when a cell in a grid for which goEditing is True has focus. The Key parameter specifies the key pressed.

For example, if the grid cell has an EditMask that allows only numbers to be entered, CanEditAcceptKey returns True if '7' is pressed or False if 'W' is pressed.

See also
CanEditModify method

CanEditModify method

See Also

Applies to

TCustomGrid

Declaration

```
function CanEditModify: Boolean; dynamic;
```

Description

The CanEditModify method determines if a grid cell can be modified. CanEditModify is called when the user presses a key when a cell in a grid has focus. CanEditModify returns True if the contents of the cell can be modified, or False if the contents of the cell are read-only.

For example, if the user tries to edit a cell in a grid for which goEditing is False, CanEditModify returns False.

See also
CanEditAcceptKey method

CanEditShow method

See Also

Applies to

TCustomGrid

Declaration

```
function CanEditShow: Boolean; virtual;
```

Description

The CanEditShow method determines if the TInPlaceEdit component for a cell being edited is shown. If the in-place editor should be shown, CanEditShow returns True. If the in-place editor shouldn't be shown, CanEditShow returns False.

The CanEditShow method allows you to prevent a grid cell from being shown when it normally would be shown by default. To modify CanEditShow for this functionality, call the inherited CanEditShow method first, then add code to prevent the in-place editor from being shown depending on the circumstances of your application.

See also

HideEditor method

ShowEditor method

ColumnMoved method

See Also

Applies to

TCustomGrid

Declaration

```
procedure ColumnMoved(FromIndex, ToIndex: Longint); dynamic;
```

Description

The ColumnMoved method is the protected implementation for a custom grid's OnColumnMoved event handler. The ColumnMoved method does nothing except call any event handler attached to the OnColumnMoved event. You can override ColumnMoved to provide other responses in addition to the inherited event-handler call.

The ColumnMoved method is called when the user moves a column in a grid that has goColMoving in the Options property set. The FromIndex parameter specifies the original column index (corresponding to the Col property) of the column being moved. The ToIndex parameter specifies the destination column index. Use the ColumnMoved method to move any data that underlies a column when a column is moved.

See also
RowMoved method

ColWidthsChanged method

See Also

Applies to

TCustomGrid

Declaration

```
procedure ColWidthsChanged; dynamic;
```

Description

The ColWidthsChanged method is called when the user resizes the width of a column in a grid that has goColSizing in the Options property set. Modify ColWidthsChanged if the column cells need to be redrawn depending on the width of the column. For example, if the cells contain bitmaps that are stretched to fit the width of the column, you should redraw the cells of the column in the ColWidthsChanged method.

See also

ColWidths property

RowHeightsChanged method

CreateEditor method

Applies to
TCustomGrid

Declaration
```
function CreateEditor: TInplaceEdit; virtual;
```

Description
The CreateEditor method creates the in-place editor for a grid.

DrawCell method

Applies to

TCustomGrid

Declaration

```
procedure DrawCell(ACol, ARow: Longint; ARect: TRect;
    AState: TGridDrawState); virtual; abstract;
```

Description

The DrawCell method is the protected implementation for a custom grid's OnDrawCell event handler. The DrawCell method does nothing except call any event handler attached to the OnDrawCell event. You can override DrawCell to provide other responses in addition to the inherited event-handler call.

The ACol and ARow parameters specify the column and row coordinates of the cell. The ARect parameter is the rectangular area (in pixel coordinates) to be drawn. The AState parameter specifies state information (selected, focused, or fixed) about the cell.

FGridState field

Applies to
TCustomGrid

Declaration
`FGridState: TGridState;`

Description
The FGridState field reflects the current state of the grid. Here are the possible values for FGridState:

Value	Meaning
gsNormal	The grid is in the normal state; not being moved, sized, or selected.
gsColMoving	A column of the grid is being moved.
gsColSizing	A column of the grid is being resized.
gsRowMoving	A row of the grid is being moved.
gsRowSizing	A row of the grid is being resized.
gsSelecting	A cell or group of cells is being selected in the grid.

FSaveCellExtents field

Applies to
TCustomGrid

Declaration
```
FSaveCellExtents: Boolean;
```

Description
The FSaveCellExtents field determines if the cell extent information for a grid is stored in the .DFM file. Set FSaveCellExtents to True to save the cell extent information to the .DFM file, or False to save the cell extent information elsewhere. For example, the TDBGrid cell extent information is saved with the TField object information, instead of with the grid information in the .DFM file.

GetEditLimit method

See Also

Applies to

TCustomGrid

Declaration

```
function GetEditLimit: Integer; dynamic;
```

Description

The GetEditLimit method returns the maximum number of characters that can appear in the in-place editor for the cells in a grid. The GetEditLimit method is called when a cell is about to be shown, to determine the limit of the number of character of the text.

See also

GetEditMask method

GetEditText method

GetEditMask method

See Also

Applies to

TCustomGrid

Declaration

```
function GetEditMask(ACol, ARow: Longint): string; dynamic;
```

Description

The GetEditMask method returns the edit mask for a cell in a grid. The ACol and ARow parameters specify the column and row of the cell. The returned string is an edit mask for the cell which performs the function of an EditMask property. The GetEditMask method is called when a cell is about to be shown, to determine how to display the text of a cell.

See also

GetEditLimit method

GetEditText method

GetEditText method

See Also

Applies to

TCustomGrid

Declaration

```
function GetEditText(ACol, ARow: Longint): string; dynamic;
```

Description

The GetEditText method returns the edit mask for a cell in a grid. The ACol and ARow parameters specify the column and row of the cell. The returned string is the text to be displayed in the cell. The GetEditText method is called when a cell is about to be shown, to determine the text of a cell.

See also

GetEditLimit method

GetEditMask method

SetEditText method

HideEditor method

See Also

Applies to

TCustomGrid

Declaration

```
procedure HideEditor;
```

Description

The HideEditor method forces the in-place editor for a cell to be hidden.

See also

ShowEditor method

ShowEditorChar method

InvalidateCell method

See Also

Applies to

TCustomGrid

Declaration

```
procedure InvalidateCell(ACol, ARow: Longint);
```

Description

The InvalidateCell method causes the visible portion of a grid cell to repaint. Nothing is repainted if the cell is not visible on screen. The ACol and ARow parameters specify the column and row of the cell.

See also
InvalidateRow method

InvalidateRow method

See Also

Applies to

TCustomGrid

Declaration

```
procedure InvalidateRow(ARow: Longint);
```

Description

The InvalidateRow method causes the visible portion of a grid row to repaint. Nothing is repainted if the row is not visible on screen. The ARow parameter specifies the row.

See also
InvalidateCell method

MaxCustomExtents constant

See Also

Declaration

```
const
    MaxCustomExtents = 65520 div SizeOf(Integer);
```

Description

The MaxCustomExtents constant represents the maximum number of customized column widths or row heights a single grid can contain. The constant is the number of integers that can fit in a single 64K memory segment. Grids store the sizes of each row and column that varies from the default size.

See also

ColWidths property

RowHeights property

MouseCoord method

Applies to
TCustomGrid

Declaration
```
function MouseCoord(X, Y: Integer): TGridCoord;
```

Description
The MouseCoord method returns the grid coordinates of the cell that is at a mouse coordinate location. The X and Y parameters specify horizontal and vertical pixel coordinates, respectively.

MoveColRow method

Applies to
TCustomGrid

Declaration
```
procedure MoveColRow(ACol, ARow: Longint; MoveAnchor, Show: Boolean);
```

Description
The MoveColRow method changes the current column and row of a grid, as specified by the Col and Row properties. The ACol and ARow parameters specify the new row and column to be made current, respectively.

The MoveAnchor parameter specifies whether to move the anchor to the new location , or leave the anchor at its current location. The anchor is simply a cell that extends the cell selection. When multiple cells are selected in a grid, the selection extends from one focused cell to the anchor. If only one cell is selected, the focused cell and the anchor are the same cell. Specify True for MoveAnchor to move the anchor, or False to leave the anchor at its current location (and extend the selection).

The Show parameter specifies whether to scroll the grid to display the new current cell if it exists in the non-visible portion of the grid. Specify True for Show to scroll the grid, or False to not scroll the grid.

RowHeightsChanged method

See Also

Applies to

TCustomGrid

Declaration

```
procedure RowHeightsChanged; dynamic;
```

Description

The RowHeightsChanged method is called when the user resizes the height of a row in a grid that has goRowSizing in the Options property set. Modify RowHeightsChanged if the row cells need to be redrawn depending on the height of the row. For example, if the cells contain bitmaps that are stretched to fit the height of the row, you should redraw the cells of the row in the RowHeightsChanged method.

See also

RowHeights property

ColWidthsChanged method

RowMoved method

See Also

Applies to

TCustomGrid

Declaration

```
procedure RowMoved(FromIndex, ToIndex: Longint); dynamic;
```

Description

The RowMoved method is the protected implementation for a custom grid's OnRowMoved event handler. The RowMoved method does nothing except call any event handler attached to the OnRowMoved event. You can override RowMoved to provide other responses in addition to the inherited event-handler call.

The RowMoved method is called when the user moves a row in a grid that has goRowMoving in the Options property set. The FromIndex parameter specifies the original row index (corresponding to the Row property) of the column being moved. The ToIndex parameter specifies the destination row index. Use the RowMoved method to move any data that underlies a row when a row is moved.

See also
ColumnMoved method

ScrollData method

Applies to
TCustomGrid

Declaration
```
procedure ScrollData(DX, DY: Integer);
```

Description
The ScrollData method scrolls the data portion of the grid. The DX parameter specifies the number of columns to scroll horizontally. Specify a positive number to display columns to the right, or a negative number to display columns to the left. The DY parameter specifies the number of rows to scroll vertically. Specify a positive number to display columns below, or a negative number to display columns above.

SelectCell method

Applies to
TCustomGrid

Declaration
```
function SelectCell(ACol, ARow: Longint): Boolean; virtual;
```

Description
The SelectCell method returns True if a cell in the grid can be selected by the user. The ACol and ARow parameters specify the column and row of the cell, respectively. The SelectCell method is called when the user attempts to select a cell.

Use SelectCell to reduce the selectable area of the grid. For example, if your application displays a 3 by 3 grid (9 cells), but only 8 cells contain data, you can modify SelectCell to prevent the cell that contains no data from being selected.

SetEditText method

See Also

Applies to

TCustomGrid

Declaration

```
procedure SetEditText(ACol, ARow: Longint; const Value: string); dynamic;
```

Description

The SetEditText method specifies the text value of a cell in a grid. The ACol and ARow parameters specify the column and row of the cell, respectively. The Value parameter specifies the text value to assign to the cell.

See also
GetEditText method

ShowEditor method

See Also

Applies to

TCustomGrid

Declaration

```
procedure ShowEditor;
```

Description

The ShowEditor method forces the in-place editor for a cell to be shown.

See also
HideEditor method

ShowEditorChar method

See Also

Applies to

TCustomGrid

Declaration

```
procedure ShowEditorChar(Ch: Char);
```

Description

The ShowEditorChar method forces the in-place editor for a cell to be shown, and activates the editor as if the character specified by the Ch parameter has been entered into it by the user.

An example of when ShowEditorChar is called is when a the user types a character when a cell in a grid for which goEditing is True is selected, but does not have focus. The in-place editor should be shown, and the user-typed character should be entered into the editor. ShowEditorChar is not called when the user types a character when a cell in a grid for which goEditing is True has focus, because the editor is already shown and will receive typed characters automatically.

See also

HideEditor method

ShowEditor method

SizeChanged method

Applies to

TCustomGrid

Declaration

```
procedure SizeChanged(OldColCount, OldRowCount: Longint); dynamic;
```

Description

The SizeChanged method is called when the number of columns or rows in a grid is changed. The OldColCount and OldRowCount parameters specify the previous number of columns and rows of the grid, respectively. The new numbers of columns and rows are specified by the ColCount and RowCount properties, respectively.

Sizing method

Applies to
TCustomGrid

Declaration
```
function Sizing(X, Y: Integer): Boolean;
```

Description
The Sizing method returns True if a row or column sizing operation can be performed at given pixel coordinates. The X and Y parameters specify the horizontal pixel coordinates respectively.

If X and Y specify a mouse pointer location over the border between row or column titles in a grid for which goRowSizing or goColSizing is True, then Sizing returns True. The mouse pointer should change to the crHSplit or crVSplit Cursor when Sizing returns True.

TCustomGrid component

See also Fields Methods Components

Unit

Grids

Description

The TCustomGrid component is the abstract base type for all grids, including TDrawGrid, TStringGrid, TDBGrid, and any user-defined grid components.

In addition to everything it inherits from TCustomControl, TCustomGrid introduces several new properties that are explained in the VCL reference for the grid components such as string grids. TCustomGrid also overrides some of its inherited protected methods and introduces several new methods.

Fields

FGridState	FSaveCellExtents

Methods

BoxRect	GetEditMask	Paint
CanEditAcceptKey	GetEditText	RowHeightsChanged
CanEditModify	HideEditor	RowMoved
CanEditShow	InvalidateCell	ScrollData
ColumnMoved	InvalidateRow	SelectCell
ColWidthsChanged	KeyDown	SetEditText
CreateEditor	KeyPress	ShowEditor
CreateParams	MouseCoord	ShowEditorChar
DblClick	MouseDown	SizeChanged
DefineProperties	MouseMove	Sizing
DrawCell	MouseUp	TimedScroll
GetEditLimit	MoveColRow	TopLeftChanged

See also
TCustomControl component

TGridCoord type

See Also

Unit

Grids

Declaration

```
type
    TGridCoord = record
      X: Longint;
      Y: Longint;
    end;
```

Description

The TGridCoord type specifies a cell in a grid. The X field specifies a grid column, and the Y field specifies a grid row.

See also
TPoint type

TGridScrollDirection type

Unit
Grids

Declaration
```
type
     TGridScrollDirection = set of (sdLeft, sdRight, sdUp, sdDown);
```

Description
The TGridScrollDirection type defines the directions in which a grid can scroll when the TimedScroll method is called.

TGridState type

Unit
Grids

Declaration
```
type
    TGridState = (gsNormal, gsSelecting, gsRowSizing, gsColSizing,
  gsRowMoving, gsColMoving);
```

Description
The TGridState type defines the values for the FGridState field.

TimedScroll method

Applies to
TCustomGrid

Declaration
```
procedure TimedScroll(Direction: TGridScrollDirection); dynamic;
```

Description
The TimedScroll method is called at periodic intervals when the user is extending the selection in a grid and the mouse pointer is outside the grid. The Direction parameter specifies the direction to scroll the grid. By default, the grid is scrolled by changing the TopRow or LeftCol properties.

The grid is scrolled in the direction of the mouse pointer. For example, if the mouse pointer is above the grid, the Direction set contains sdUp. If the mouse pointer is below and to the right of the grid, the Direction set contains sdDown and sdRight.

TInplaceEdit component

See Also   Components

Unit

Grids

Description

The TInPlaceEdit component is the class used internally by grid components that allow in-place editing. When the Options property set of a grid contains goEditing, then the cells of a grid can be edited in-place. When a cell is edited in-place, a TInPlaceEdit object is used.

See also
CreateEditor method

TopLeftChanged method

Applies to
TCustomGrid

Declaration
```
procedure TopLeftChanged; dynamic;
```

Description
The TopLeftChanged method is called when the data in a grid has scrolled. TopLeftChanged is called when the TopRow or LeftCol properties change.

Changed method

Applies to
TCanvas, TGraphic, TPicture

Description
There are two Changed methods. One applies to TCanvas objects and the other applies to TGraphic and TPicture objects.

Changed method for canvases

Changed method for graphics and pictures

Changed method

See Also

Applies to

TCanvas

Declaration

```
procedure Changed; virtual;
```

Description

The Changed method is the protected implementation for a canvas's OnChange event handler. The Changed method does nothing except call any event handler attached to the OnChange event. You can override Changed to provide other responses in addition to the inherited event-handler call.

See also
Changing method

Changed method

Applies to
TGraphic, TPicture

Declaration
`procedure Changed(Sender: TObject);`

Description
The Changed method is the protected implementation for a canvas's OnChange event handler. The Changed method does nothing except call any event handler attached to the OnChange event. You can override Changed to provide other responses in addition to the inherited event-handler call.

The Changed method should be called when the graphic or picture has changed. For example, if the Changed method is of a TBitmap object specified by the Picture property of a TImage component, Changed repaints the TImage component when the bitmap has changed.

Changing method

See Also

Applies to

TCanvas

Declaration

```
procedure Changing; virtual;
```

Description

The Changing method is the protected implementation for a canvas's OnChanging event handler. The Changing method does nothing except call any event handler attached to the OnChanging event. You can override Changing to provide other responses in addition to the inherited event-handler call.

See also
Changed method

ColorToIdent function

See Also

Unit

Graphics

Declaration

```
function ColorToIdent(Color: Longint; var Ident: string): Boolean;
```

Description

The ColorToIdent function converts a numeric color value to a color identifier string, if one has been registered. If an identifer corresponds to the numeric value passed in the Color parameter, then the Ident parameter string will be modified to specify the corresponding identifier and ColorToIdent returns True. If no identifier corresponds to the numeric value, the Ident parameter string will not be modified and ColorToIdent returns False.

Note: ColorToIdent is intended for internal use. The ColorToString function should be used instead to perform the same function.

See also

ColorToString function

GetColorValues procedure

IdentToColor function

ColorToString function

See Also

Unit

Graphics

Declaration

```
function ColorToString(Color: TColor): string;
```

Description

The ColorToString function returns a string that specifies the identifier corresponding to the color passed in the Color parameter. You can pass a numeric value or an identifier (such as clBlue) in the Color parameter. If no identifier corresponds to a numeric value, ColorToString returns the hexadecimal value of the Color parameter in a string.

See also

ColorToIdent function

GetColorValues procedure

IdentToColor function

Dormant method

See Also

Applies to

TBitmap

Declaration

```
procedure Dormant;
```

Description

The Dormant method frees the HBITMAP assigned to the bitmap (accessed by the Handle property), but preserves the bitmap by creating an image of the bitmap in memory. Use Dormant when you want to reduce the amount of GDI resources consumed by your application.

See also
FreeImage method

Draw method

Applies to
TGraphic

Declaration
```
procedure Draw(ACanvas: TCanvas; const Rect: TRect); virtual; abstract;
```

Description
The Draw method draws the graphic on a canvas. The destination canvas is specified by the ACanvas parameter, and the rectangular area of the canvas in which to draw the graphic is specified by the Rect parameter. If the size of the Rect rectangle differs from the size of the graphic, the graphic is stretched to fit inside the rectangle.

FreeImage method

See Also

Applies to

TBitmap

Declaration

```
procedure FreeImage;
```

Description

The FreeImage method frees the cached file image stored in memory by the bitmap. When a bitmap is loaded into a bitmap object, Delphi creates an image of the loaded bitmap in memory. If the bitmap isn't changed, the memory image is used when you save the bitmap to verify that the bitmap has not lost color depth. FreeImage releases the memory allocated for the bitmap image. If you call FreeImage, the memory requirements of your application are reduced but you risk loosing color depth of the bitmap.

See also
Dormant method

GetColorValues procedure

See Also

Unit

Graphics

Declaration

```
procedure GetColorValues(Proc: TGetStrProc);
```

Description

The GetColorValues procedure iterates through the registered color identifier list, passing each identifier to the Proc procedure as a string parameter. Use GetColorValues to obtain strings corresponding to the color identifers such as clBlue and clBtnFace.

See also

ColorToIdent function

ColorToString function

IdentToColor function

GetDIB function

See Also

Unit

Graphics

Declaration

```
function GetDIB(Bitmap: HBITMAP; Palette: HPALETTE; var BitmapInfo; var
  Bits): Boolean;
```

Description

The GetDIB function retrieves the bits of the bitmap specified by the Bitmap parameter and copies them, in device-independent format, into the Bits structure. The Palette parameter specifies the palette of the bitmap, and the BitmapInfo parameter specifies the info header for the bitmap.

Note: Before calling GetDIB, you should retreive the sizes of the info header and the bitmap with the GetDIBSizes procedure. Then you should allocate enough memory with the MemAlloc function.

See also

GetDIBSizes procedure

MemAlloc function

GetDIBSizes procedure

See Also

Unit

Graphics

Declaration

```
procedure GetDIBSizes(Bitmap: HBITMAP; var InfoHeaderSize: Integer; var
  ImageSize: Longint);
```

Description

The GetDIBSizes function returns the sizes in bytes of the info header and image of a bitmap. The bitmap to measure is specified by the Bitmap parameter. The InfoHeaderSize parameter returns the number of bytes of the info header. The ImageSize parameter returns the number of bytes of the image.

Retrieve the sizes of the info header and the bitmap so you can allocate enough memory with the MemAlloc function before retrieving the bitmap in device-independent format with the GetDIB function.

See also

GetDIB function

MemAlloc function

GetEmpty method

Applies to

TGraphic

Declaration

```
function GetEmpty: Boolean; virtual; abstract;
```

Description

The GetEmpty method returns True if the graphic contains no image, or False if the graphic contains an image. The GetEmpty method is called to specify the Empty property.

GetHeight method

See Also

Applies to

TGraphic

Declaration

```
function GetHeight: Integer; virtual; abstract;
```

Description

The GetHeight method returns the height in pixels of the graphic. The GetHeight method is called to specify the Height property.

See also
GetWidth method
SetHeight method
SetWidth method

GetWidth method

See Also

Applies to

TGraphic

Declaration

```
function GetWidth: Integer; virtual; abstract;
```

Description

The GetWidth method returns the width in pixels of the graphic. The GetWidth method is called to specify the Width property.

See also

GetHeight method

SetHeight method

SetWidth method

IdentToColor function

See Also

Unit

Graphics

Declaration

```
function IdentToColor(const Ident: string; var Color: Longint): Boolean;
```

Description

The IdentToColor function converts a color identifier string to a numeric color value, if one has been registered. If a numeric value corresponds to the identifier string passed in the Ident parameter, then the Color parameter be modified to specify the corresponding value and IdentToColor returns True. If no value corresponds to the identifier string, the Color value will not be modified and IdentToColor returns False.

See also

ColorToIdent function

GetColorValues procedure

InitGraphics procedure

Unit
Graphics

Declaration
```
procedure InitGraphics;
```

Description
The InitGraphics procedure initializes the Delphi graphics unit. The Controls unit, included in every project by default, calls InitGraphics automatically.

Caution: You should never call InitGraphics from your code. Doing so will generate errors.

LoadFromClipboardFormat method

See Also

Applies to

TGraphic

Declaration

```
procedure LoadFromClipboardFormat(AFormat: Word; AData: THandle;
    APalette: HPALETTE); virtual; abstract;
```

Description

The LoadFromClipboardFormat method loads a graphic from the Clipboard. LoadFromClipboardFormat replaces the current image with the data pointed to by the AData parameter. The palette for the graphic is specified by the APalette parameter. LoadFromClipboardFormat is called if you have registered the TGraphic with the TPicture object with the RegisterClipboardFormat method.

See also

SaveToClipboardFormat method

SupportsClipboardFormat method

MemAlloc function

See Also

Unit

Graphics

Declaration

```
function MemAlloc(Size: Longint): Pointer;
```

Description

The MemAlloc function allocates the number of bytes passed in the Size parameter from the heap.

See also

GetDIB function

GetDIBSizes procedure

ReadData method

See Also

Applies to

TGraphic

Declaration

```
procedure ReadData(Stream: TStream); virtual;
```

Description

The ReadData method reads the graphic image from the stream written to with the WriteData method. You need to implement ReadData only if you've implemented a new WriteData method. Otherwise, ReadData will simply call LoadFromStream (and WriteData will call SaveToStream).

You would want to implement a new ReadData method if you intend to change the way the graphic is stored in the .DFM file. Instead of simply writing to and reading from a stream, you might want to change or compress the graphic data before storing the data in the .DFM file.

See also
WriteData method

RegisterClipboardFormat method

See Also

Applies to

TPicture

Declaration

```
class procedure RegisterClipboardFormat(AFormat: Word; AGraphicClass:
  TGraphicClass);
```

Description

The RegisterClipboardFormat method registers a new TGraphic class for use in LoadFromClipboardFormat and SaveToClipboardFormat methods of TGraphic. The AFormat parameter specifies a Clipboard format that has been registered with Windows with the Windows API function call RegisterClipboardFormat. The AGraphicClass parameter specifies the graphic class to register.

You must register any new graphic classes (other than TBitmap, TIcon, or TMetafile) to be able to load or save the graphic data from the Clipboard.

See also
SupportsClipboardFormat method

RegisterFileFormat method

See Also

Applies to

TPicture

Declaration

```
class procedure RegisterFileFormat(const AExtension, ADescription: string;
    AGraphicClass: TGraphicClass);
```

Description

The RegisterFileFormat method registers a graphic file format with TPicture that can be used with a Open or Save dialog box. The AExtension parameter specifies the three-character DOS file extension to associate with the graphic class (for example, "BMP" is associated with TBitmap). The ADescription parameter specifies the description of the graphic to appear in the drop down list of the dialog box (for example, "Bitmaps" is the description of TBitmap). The AGraphicClass parameter registers the new graphic class to associate with the file format.

See also
RegisterClipboardFormat method

SaveToClipboardFormat method

See Also

Applies to

TGraphic

Declaration

```
procedure SaveToClipboardFormat(var AFormat: Word; var AData: THandle;
    var APalette: HPALETTE); virtual; abstract;
```

Description

The SaveToClipboardFormat method saves a graphic to a Clipboard format. The palette for the graphic is specified by the APalette parameter. To be able to save the TGraphic to a Clipboard format, the format needs to have been registered with the TPicture object with the RegisterClipboardFormat method.

See also

LoadFromClipboardFormat method

SupportsClipboardFormat method

SetHeight method

See Also

Applies to

TGraphic

Declaration

```
procedure SetHeight(Value: Integer); virtual; abstract;
```

Description

The SetHeight method resizes the height of the graphic. The new height in pixels is specified by the Value parameter. The SetHeight method is called when the value of the Height property changes.

By default, if the new height is smaller than the original height, the graphic image is truncated (not stretched) to fit in the new height. If the new height is larger than the original width, again the graphic is not stretched. Instead, the Brush color of the graphic is used to fill the new area created when the height was expanded, unless the graphic contains transparent colors. In that case, the transparent color is used to fill the new area.

See also

GetHeight method

GetWidth method

SetWidth method

SetWidth method

See Also

Applies to

TGraphic

Declaration

```
procedure SetWidth(Value: Integer); virtual; abstract;
```

Description

The SetWidth method resizes the width of the graphic. The new width in pixels is specified by the Value parameter. The SetWidth method is called when the value of the Width property changes.

By default, if the new width is smaller than the original width, the graphic image is truncated (not stretched) to fit in the new width. If the new width is larger than the original width, again the graphic is not stretched. Instead, the Brush color of the graphic is used to fill the new area created when the width was expanded, unless the graphic contains transparent colors. In that case, the transparent color is used to fill the new area.

See also

GetHeight method

GetWidth method

SetHeight method

StringToColor function

See Also

Unit

Graphics

Declaration

```
function StringToColor(S: string): TColor;
```

Description

The StringToColor function returns a color that corresponds to the identifier string passed in the S parameter. For example, to obtain clBlue, pass 'clBlue' in the S parameter.

See also

ColorToIdent function

ColorToString function

GetColorValues procedure

SupportsClipboardFormat method

See Also

Applies to

TPicture

Declaration

```
class function SupportsClipboardFormat(AFormat: Word): Boolean;
```

Description

The SupportsClipboardFormat method returns True if the Clipboard format specified by the AFormat parameter is supported by the LoadFromClipboardFormat method. For a format to be supported, it must have been registered with the RegisterClipboardFormat method.

See also
SaveToClipboardFormat method

TBitmap object (protected)

Unit

Graphics

Description

The TBitmap object is a graphic object derived from TGraphic, customized to handle Windows bitmaps. In addition to the public and published items explained in the VCL reference for the TBitmap object, TBitmap overrides three inherited protected methods.

TBitmap overrides the Draw method to render the bitmap image, and overrides ReadData and WriteData to read and write bitmap data, respectively.

Methods

Draw	ReadData	WriteData

See also
TGraphic object (protected)
TBitmap object VCL reference

TCanvas object (protected)

See also Methods Objects

Unit

Graphics

Description

The TCanvas object represents the drawing surface of a Delphi component. In addition to everything it inherits from its ancestor, TPersistent, TCanvas introduces numerous properties, methods, and events. Nearly all of these are public or published, and explained in the VCL reference for the TCanvas object.

TCanvas does introduce three protected methods, however. The Changed and Changing methods trigger the OnChange and OnChanging events, respectively, and the CreateHandle method creates a device-context handle for the canvas.

Methods

Changed	Changing	CreateHandle

See also

TPersistent object

TCanvas object VCL reference

TGraphic object (protected)

See also Properties Methods Events Objects

Unit

Graphics

Description

The TGraphic object is the abstract base object for all graphic images in Delphi, such as TBitmap, TIcon, and TMetafile. You can also define your own graphic image types by deriving from TGraphic.

In addition to everything inherited from its ancestor, TPersistent, TGraphic introduces properties, methods, and events common to all graphic-image objects. Most of these are public or publish, and explained in the VCL reference for the TGraphic object.

The protected methods introduced by TGraphic serve to provide the structure of a common interface to all graphics objects. In addition to overriding several inherited methods, TGraphic introduces a Changed method that triggers the OnChange event, a Draw method to render the image, ReadData and WriteData method which load and store the binary image, and load and save methods for various media, including streams, files, and the Clipboard.

Properties

Empty	Modified	Width
Height		

Methods

Assign	LoadFromClipboardFormat	SaveToClipboardFormat
Changed	LoadFromFile	SaveToFile
Create	LoadFromStream	SaveToStream
DefineProperties	ReadData	WriteData
Draw		

Events

OnChange

See also

TPersistent object

TGraphic object VCL reference

TGraphicClass type

See Also

Unit

Graphics

Declaration

```
type
    TGraphicClass = class of TGraphic;
```

Description

The TGraphicClass type is a container class for the TGraphic object and its descendents. TGraphicClass is the type of the parameter of the GraphicExtension and GraphicFilter functions.

See also
TGraphic object (protected)

TIcon object (protected)

See also Methods Objects

Unit

Graphics

Description

The TIcon object is a graphic object derived from TGraphic, customized to handle Windows icons. In addition to the public and published items explained in the VCL reference for the TIcon object, TIcon overrides two inherited protected methods.

TIcon overrides the Draw method to render the icon image, and overrides SaveToClipboardFormat to handle copying icons to the Clipboard.

Methods

Draw	SaveToClipboardFormat

See also

TGraphic object (protected)

TIcon object VCL reference

TMetafile object (protected)

See also Methods Objects

Unit

Graphics

Description

The TMetafile object is a graphic object derived from TGraphic, customized to handle Windows metafiles. In addition to the public and published items explained in the VCL reference for the TMetafile object, TMetafile overrides three inherited protected methods.

TMetafile overrides the Draw method to render the metafile image, and overrides ReadData and WriteData to read and write metafile data, respectively.

Methods

Draw	ReadData	WriteData

See also
TGraphic object (protected)
TMetafile object VCL reference

TPicture object (protected)

See also Methods Objects

Unit

Graphics

Description

The TPicture object is a container for graphic objects (defined by TGraphic) that can load images from any of the registered file types. In addition to everything it inherits from its ancestor, TPersistent, TPicture introduces a number of properties, methods, and events. Most of these are public or published, and explained in the VCL reference for the TPicture object.

TPicture does override the protected DefineProperties method, however, and introduces one other protected method, Changed, which triggers the OnChange event.

Methods

Changed	DefineProperties

See also
TPersistent object
TGraphic object
TPicture object VCL reference

WriteData method

See Also

Applies to

TGraphic

Declaration

```
procedure WriteData(Stream: TStream); virtual;
```

Description

The WriteData method writes the graphic image to the stream specified by the Stream parameter. By default, WriteData simply calls SaveToStream. You need to implement a new WriteData method only if you intend to change the way the graphic is stored in the .DFM file. If you implement a new WriteData method, you should implement a new ReadData method that corresponds to WriteData for the graphic.

For example, if you wanted to reduce the number of bytes required to store the graphic image data in the .DFM file, you could attach your compression algorithm to the WriteData method, then call SaveToStream, saving the compressed structure to the stream. Of course, you would need to decompress the graphic image data in the ReadData method to retrieve the graphic.

See also
ReadData method

Activate method

See also

Applies to

TForm

Declaration

```
procedure Activate; dynamic;
```

Description

The Activate method is the protected implementation method for the OnActivate event of form components. The form component calls Activate in response to the CM_ACTIVATE message, sent when a form gets the focus either because of a change in active windows within an application or because of the application becoming active.

The default Activate method calls any event handler attached by the user to the OnActivate event.

See also

Deactivate method

OnActivate event

AllocateHWnd function

See also

Unit

Forms

Declaration

```
function AllocateHWnd(Method: TWndMethod): HWND;
```

Description

The AllocateHWnd function constructs an invisible window in Windows with the window-procedure method passed in the Method parameter and returns the invisible window's window handle.

AllocateHWnd is useful for components that need a window handle for receiving messages, but which don't need an onscreen representation. Timer components, for example, need to receive WM_TIMER messages, but don't appear on the screen at all.

Windows allocated with AllocateHWnd should be freed with the DeallocateHWnd procedure.

See also
DeallocateHWnd procedure

AutoScrollInView method

See also

Applies to

TScrollingWinControl

Declaration

```
procedure AutoScrollInView(AControl: TControl);
```

Description

The AutoScrollInView method implements automatic scrolling to ensure that a control receiving focus (specified by AControl) and any scrolling controls that contain it all scroll so that the control is showing when it receives focus. AutoScrollInView calls the ScrollInView method to perform any needed scrolling.

See also
ScrollInView method

Deactivate method

See also

Applies to

TForm

Declaration

```
procedure Deactivate; dynamic;
```

Description

The Deactivate method is the protected implementation method for the OnDeactivate event of form components. The form component calls Deactivate in response to the CM_DEACTIVATE message, sent when a form gets the focus either because of a change in active windows within an application or because of the application becoming active.

The default Deactivate method calls any event handler attached by the user to the OnDeactivate event.

See also
Activate method
OnDeactivate event

DeallocateHWnd procedure

See also

Unit

Forms

Declaration

```
procedure DeallocateHWnd(Wnd: HWND);
```

Description

The DeallocateHWnd procedure disposes of an invisible window created by the AllocateHWnd function. The Wnd parameter is the window handle returned by AllocateHWnd.

See also
AllocateHWnd function

DialogHandle property

See also Example

Applies to

TApplication

Declaration

```
property DialogHandle: HWnd;
```

Description

The DialogHandle property provides a mechanism for using non-Delphi dialog boxes in a Delphi application. When you create a dialog box using the CreateDialog API function, that dialog box needs to be able to see messages from the application's message loop to operate as a modeless dialog box.

Assigning the handle of your modeless dialog box to DialogHandle when it receives an activation message (WM_NCACTIVATE) and setting DialogHandle to zero when the dialog box receives a deactivation message.

Example

Here is part of the window procedure for Delphi's encapsulation of the Windows search-and-replace common dialog box:

```
function SearchAndReplaceWndProc(Wnd: HWnd; Msg, WParam: Word; LParam: Longint): Longint;
begin
  try
    case Msg of
      WM_DESTROY: Application.DialogHandle := 0; { clear handle on destroy }
      WM_NCACTIVATE:
        if Bool(WParam) then      { if it's activating... }
          Application.DialogHandle := Wnd{ ...assign the handle... }
        else Application.DialogHandle := 0;      { ...otherwise clear it }
      ...
    end;
    Result := CallDefDialogProc;
  except
    Application.HandleException(nil);
  end;
end;
```

See also

HookMainWindow method

UnhookMainWindow method

DisableTaskWindows function

See also

Unit

Forms

Declaration

```
function DisableTaskWindows(ActiveWindow: HWnd): Pointer;
```

Description

The DisableTaskWindows function disables all the windows in the application except the one passed in the ActiveWindow parameter. Call DisableTaskWindows when you want to use a non-Delphi dialog box as a modal dialog box in a Delphi application. There is no need to use DisableTaskWindows when using a Delphi dialog box, as the ShowModal method handles the modality.

The return value is a pointer to a list of the disabled windows, which you pass to the EnableTaskWindows procedure after the modal dialog box closes to enable the previously disabled windows. If an exception occurs while disabling the task windows, DisableTaskWindows automatically reenables any windows already disabled.

See also
EnableTaskWindows procedure

DoHide method

See also

Applies to

TForm

Declaration

```
procedure DoHide; dynamic;
```

Description

The DoHide method is the protected implementation method for the OnHide eventof form components. The form component calls DoHide in response to the CM_SHOWINGCHANGED message, sent when the application shows or hides the form.

The default DoHide method calls any event handler attached by the user to the OnHide event.

See also

OnHide event

DoShow method

DoShow method

See also

Applies to

TForm

Declaration

```
procedure DoShow; dynamic;
```

Description

The DoShow method is the protected implementation method for the OnShow event of form components. The form component calls DoShow in response to the CM_SHOWINGCHANGED message, sent when the application shows or hides the form.

The default DoShow method calls any event handler attached by the user to the OnShow event.

See also

OnShow event

DoHide method

EnableTaskWindows procedure

See also

Unit

Forms

Declaration

```
procedure EnableTaskWindows(WindowList: Pointer);
```

Description

The EnableTaskWindows procedure enables the windows in the list pointed to by the WindowList parameter. WindowList is a pointer returned as the value of the DisableTaskWindows function.

See also
DisableTaskWindows function

FreeObjectInstance procedure

See also

Unit

Forms

Declaration

```
procedure FreeObjectInstance(ObjectInstance: Pointer);
```

Description

The FreeObjectInstance procedure disposes of an object instance created by the MakeObjectInstance function. The ObjectInstance parameter is the window-procedure pointer returned by MakeObjectInstance.

See also
MakeObjectInstance function

HintWindowClass variable

See also

Unit

Forms

Declaration

```
const
    HintWindowClass: THintWindowClass = THintWindow;
```

Description

The HintWindowClass variable determines the type of component used by the application to display hint windows. The default value of HintWindowClass is THintWindow. If you derive a new type of hint-window component from THintWindow, assign that new type to HintWindowClass on application startup, and the application will create instances of your new type when called to create hint windows.

See also

THintWindow component

THintWindowClass type

HookMainWindow method

See also

Applies to

TApplication

Declaration

```
procedure HookMainWindow(Hook: TWindowHook);
```

Description

The HookMainWindow method enables a non-Delphi dialog box to receive messages sent to its parent, represented by the application object's Handle property.

Hooking the main window ensures that the non-Delphi dialog box behaves correctly as a child of the application, not as a stand-alone window. For example, switching among applications with Alt+Tab treats the application as a single task after you call HookMainWindow, rather than treating the non-Delphi dialog box as being a separate task.

There is no problem with leaving a dialog box "hooked" into the main window, even for extended periods. However, should the dialog box close, you should call the UnhookMainWindow method to release the hook.

See also
UnhookMainWindow method

IsAccel function

Unit
Forms

Declaration
```
function IsAccel(VK: Word; const Str: string): Boolean;
```

Description
The IsAccel function returns True if the character in the virtual key passed in the VK parameter is the accelerator key indicated in the string passed in Str, and False if it is not. IsAccel checks to see whether there is an ampersand (&) character in Str, and if so, whether the character following it is the same as the character represented by VK.

KeyDataToShiftState function

See also

Unit

Forms

Declaration

```
function KeyDataToShiftState(KeyData: Longint): TShiftState;
```

Description

The KeyDataToShiftState function translates the KeyData field passed with keyboard messages from Windows into a set of shift-state information for use by Delphi keyboard events (OnKeyDown, OnKeyPress, OnKeyUp).

See also
KeysToShiftState function

KeysToShiftState function

See also

Unit

Forms

Declaration

```
function KeysToShiftState(Keys: Word): TShiftState;
```

Description

The KeysToShiftState function translates the Keys field passed with keyboard messages from Windows into a set of shift-state information for use by Delphi keyboard events (OnKeyDown, OnKeyPress, OnKeyUp).

See also
KeyDataToShiftState function

MakeObjectInstance function

See also

Unit

Forms

Declaration

```
function MakeObjectInstance(Method: TWndMethod): Pointer;
```

Description

The MakeObjectInstance function returns a pointer to a window-procedure function that calls the window-procedure method passed in the Method parameter.

Windows passes messages for a window to a function called that window's window procedure. Delphi components are objects, and use object methods as their window procedures for handling messages. Since Windows cannot accept a method pointer as a window procedure, Delphi creates a surrogate window procedure in the form of a function that Windows can call, which passes the messages on to the specified method.

If you create window-procedure functions with MakeObjectInstance, be sure to dispose of them with the FreeObjectInstance procedure.

See also
FreeObjectInstance procedure

Resize method

See also

Applies to

TScrollBox, TForm, TCustomPanel

Declaration

```
procedure Resize; dynamic;
```

Description

The Resize method is the protected implementation method for the OnResize event of scroll boxes, forms, and panels. These controls all call Resize in response to the WM_SIZE message, indicating a change in the size of the control or form.

By default, Resize does nothing except call any event handler attached to the OnResize event. You can override Resize to provide other responses in addition to the inherited event-handler call.

See also
OnResize event

TApplication component (protected)

See also Property Methods Components

Unit

Forms

Description

The TApplication component is the component type of the application component for all Delphi applications. In addition to the items in the VCL reference for using the application component, there are several parts of interest to component writers.

The DialogHandle property and the HookMainWindow and UnHookMainWindow methods allow you to use non-Delphi dialog boxes in Delphi applications. TApplication also overrides the WriteComponents method to avoid writing the application object out to the stream.

Property

DialogHandle

Methods

HookMainWindow	UnHookMainWindow	WriteComponents

See also

Application variable

TApplication component VCL reference

TDesigner object

See also Objects

Unit

Forms

Description

The TDesigner object is an abstract object type derived directly from TObject. It provides the basis for all form designers in the Delphi environment, most importantly, TFormDesigner. The abstract designer object provides two properties used by all designer objects and defines five abstract methods that working designer objects must override.

Properties

Property	Usage
Form	Provides a reference to the form currently being designed
IsControl	Reflects whether the form is in its form state or its control state

Methods

Method	Usage
IsDesignMsg	Called for each message sent to a component in the designer. Returns True if the message is a design message, meaning one the designer should handle for the component.
Modified	Property and component editors call this method whenever anything in a component changes, allowing the designer to reflect the change.
Notification	Called by the Notification method of the form being designed, allowing the designer to respond to all the same notifications as the form.
PaintGrid	Called whenever the designer needs to paint the alignment grid on the form's canvas.
ValidateRename	Called by the form's ValidateRename method, generally to provide tighter restrictions on renaming, such as prohibiting reserved words.

See also
TFormDesigner object

TForm component (protected)

See also Methods Components

Unit

Forms

Description

The TForm component is the immediate ancestor type for all forms designed in the Delphi environment. A form is a specialized kind of scrolling windowed control. In addition to the items in the VCL Reference for using form components, there are a number of protected methods that TForm either introduces or overrides.

Methods

- Activate
- ChangeScale
- CreateParams
- CreateWindowHandle
- CreateWnd
- Deactivate
- DefaultHandler
- DefineProperties
- DestroyWindowHandle
- DestroyWnd
- DoHide
- DoShow
- GetClientRect
- Notification
- Paint
- PaintWindow
- ReadState
- Resize
- SetName
- ValidateRename
- VisibleChanging
- WndProc

See also

TFormClass type

TScrollingWinControl component

TForm component VCL reference

TFormClass type

See also

Unit

Forms

Declaration

```
type
    TFormClass = class of TForm;
```

Description

The TFormClass type is the object-type reference type for form component types.

See also
TForm component (protected)

TScrollBox component (protected)

See also Methods Components

Unit

Forms

Description

The TScrollBox component is a customized scrolling windowed control. In addition to the items in the VCL reference for using the scroll-box component, there are several protected methods of interest to component writers.

TScrollBox introduces a Resize method that implements the OnResize event, and also overrides the Create constructor and the CreateParams method.

Methods

Create	CreateParams	Resize

See also

TScrollingWinControl component

TScrollBox component VCL reference

TScrollingWinControl component

See also Properties Methods Components

Unit

Forms

Description

The TScrollingWinControl component is an abstract component type that is the basis for two useful components: TForm and TScrollBox. In addition to the items inherited from TWinControl, a scrolling windowed control has three properties and several new methods.

The three properties introduced by TScrollingWinControl are AutoScroll, HorzScrollBar, and VertScrollBar. The scroll-bar properties are published, and therefore show up in all scrolling windowed controls and forms. AutoScroll is protected, but published by both TScrollBox and TForm.

TScrollingWinControl introduces two methods that handle scrolling: ScrollInView, which ensures that a specified control is fully visible, and AutoScrollInView, which does the same for automatically-scrolling controls. TScrollingWinControl also overrides the Create constructor and the AlignControls, CreateWnd, and Destroy methods.

Properties

AutoScroll	HorzScrollBar	VertScrollBar

Methods

AlignControls	Create	Destroy
AutoScrollInView	CreateWnd	ScrollInView

See also

TForm component (protected)

TScrollBox component (protected)

TWindowHook type

See also

Unit

Forms

Declaration

```
type
    TWindowHook = function(var Message: TMessage): Boolean of object;
```

Description

The TWindowHook type is a method-pointer type used for the dialog-procedures of non-Delphi dialog boxes. The dialog procedure is similar to a window procedure for a window, in that it processes messages for the dialog box, but its syntax is somewhat different.

The application object's HookMainWindow and UnhookMainWindow methods each take a parameter of type TWindowHook.

See also

HookMainWindow method

UnhookMainWindow method

TWndMethod type

See also

Unit

Forms

Declaration

```
type
    TWndMethod = procedure(var Message: TMessage) of object;
```

Description

The TWndMethod type is a method-pointer type used for window procedures. The AllocateHWnd and MakeObjectInstance functions both take window-procedure methods of type TWndMethod as their parameters.

See also

AllocateHWnd function

MakeObjectInstance function

UnhookMainWindow method

See also

Applies to

TApplication

Declaration

```
procedure UnhookMainWindow(Hook: TWindowHook);
```

Description

The UnhookMainWindow method releases the dialog procedure previously "hooked" by a call to the HookMainWindow method.

See also
HookMainWindow method

Topic Not Found

The topic you are looking for was not found. The problem might be with the Help file or it might be with the link to the topic. You may be able to find this topic by using the Contents, Index or Search tab.

To search for a topic,

1 Click the Contents tab to browse through topics by category.

2 Click the Index tab to see a list of index entries.

Either type the word you're looking for or scroll through the list.

3 Click the Find tab to search for words or phrases

Welcome to RoboHELP. Click Topic (Ctrl+T) to add your first Help topic.

This is associated with the Delphi 95 project file document.