About property editors

See also

The Object Inspector provides several different editors you can use to modify or set a property. If you write your own components, you can generally provide any of these ways to set your component's properties. You may want to keep in mind that, as originally designed, these property editors typically correspond to Object Pascal types.

See also

Manipulating components
Setting properties
Setting shared properties
How the Object Inspector displays properties
The Name property
Viewing nested properties
Working with code

Simple editor

Edit these properties by directly entering in the new value. Properties that appear this way generally correspond to simple types. These include numeric and string types. Delphi checks these fields to ensure that the types match. For example, if you attempt to enter a string (character) value into a numeric type field, Delphi prompts you with an "Invalid property value" message and resets the value in the field.

Drop Down List editor

To change a value using a drop-down list editor, do one of the following:

- Click the drop-down arrow and select from the list.
- Press Alt+Down Arrow and select from the list.
- Double-click the Value column until the value you want is active.

These properties correspond, in most cases, to enumerated types.

Object List editor

This editor is similar to the drop-down list editor except that the list displays only the names of objects in the form that could be assigned to that property. For example, the object list editor for the Menu property can contain only components of type TMainMenu.

Dialog box editor

The ellipse to the right of the Values column signifies that there is a dialog box that you can use to edit this property. The following is a list of all the Delphi dialog box editors:

Color editor
DDE info
Fields editor
Filter editor
Font editor
Input Mask editor
Insert Object
Mask Text editor
Menu Designer
Notebook editor
Picture editor
String List editor

To open a dialog box editor, do one of the following:

- Click the ellipsis button.
- Double-click in the Value column.
- Press Ctrl+Enter.

When you have finished editing the property, click OK.

Nested Properties editor

A plus (+) sign to the left of the property name indicates nested levels of properties that you can access by double-clicking the property name.
These properties might correspond to set types. When you change all members of a set to False, you see an empty set, or [ ], in the Value column.

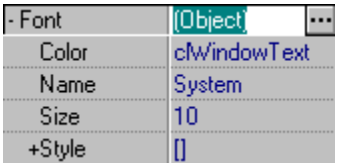
Color Property editor

The Color property is unique because it provides a combination of editors, both a drop-down list and a dialog box.

To edit the Color property, do one of the following:

- Click the drop-down arrow and select from the list.
- Double-click the Value column to display the Color editor and use it to select a new color.
- Type in the number or name of a color constant (such as clBlue, or any number that corresponds to an ASCII color value).

Font Property editor



The Font property also provides a combination of editors: a dialog box and several nested levels of properties.

Handling TApplication events

Example

Since the Application component does not appear in the Object Inspector, you must manually create the event handler.

1. In the **implementation** section of the main form, write an event handler.
2. In the **interface** section, declare the method in the type declaration of the main form.
3. Assign the name of the method to an Application event and add that statement to the Oncreate event handler of the main form.

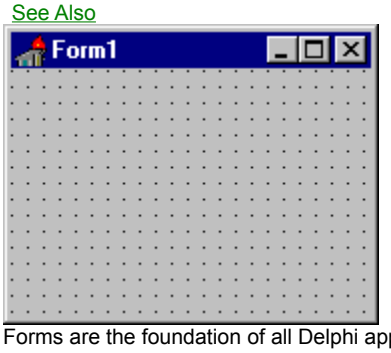
Example

The following example creates an OnActivate event handler for the Application that will display a dialog box when the application is activated.

```
unit Unit1;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  TForm1 = class(TForm)
    procedure AppOnActivate(Sender: Tobject); { You add this line }
    procedure FormCreate(Sender: Tobject); {Delphi adds this line }
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

{ You write the following method }
procedure TForm1.AppOnActivate(Sender: TObject);
begin
  MessageDlg('I am activated!', mtInformation, [mbOK], 0);
end;

{ Delphi generates the following method shell if you generate this event
  handler from the Object Inspector }
procedure TForm1.FormCreate(Sender: TObject);
begin
  { You write the following line to attach AppOnActivate to the
    OnActivate event of Application }
  Application.OnActivate := AppOnActivate;
end;

end.
```

About forms

See Also



Forms are the foundation of all Delphi applications. The form is a component. You place other components onto the form to build an application interface.
You develop your application by customizing the main form, and adding and customizing forms for other parts of the interface. You customize forms by adding components and setting properties.
The form is a window, and therefore by default includes standard window functionality such as:

- Control menu
- Minimize and Maximize buttons
- Title bar
- Resizeable borders

You can change these features, as well as any other property of the form, at design time using the Object Inspector.

See also

Adding components to the form
Creating dialog boxes
Designing a user interface
Setting properties
TForm component
Using the Form component

Client area

Enables you to view or modify any part of a form in the active window. Use the grid of dots to align objects on the form.

About the SpeedBar

See Also

The Delphi SpeedBar provides shortcuts for menu commands. The graphic below shows the default SpeedBar. However, you can customize the SpeedBar by choosing Configure from the SpeedBar SpeedMenu.

To find out more about a default SpeedBar button, click a button in the graphic above.

You can use the separator line that lies between the SpeedBar and Component palette to horizontally resize the SpeedBar.

The SpeedBar has Help Hints. To enable Help Hints, select Show Hints from the SpeedBar SpeedMenu.

See also

Component palette
Configuring the SpeedBar
Keyboard shortcuts

Open Project button

Displays the Open Project dialog box so you can open a new project.
The menu equivalent is File|New.

Open File button

Displays the Open File dialog box so you can open a new text file.
The menu equivalent is File|Open.

Save button

Stores changes made to all files in the open project the current name of each file.
If you have not previously saved the project, Delphi opens the Save As dialog box, where you can enter a file name.

Save button

Stores changes made to active file in the Code Editor.
If you have not previously saved the file Delphi opens the Save As dialog box where you can enter a filename.
The menu equivalent is File|Save.

Add File To Project button

Adds the active file in the Code Editor to the currently open project.

The menu equivalents are

- File|Add to Project
- Add File on the Project Manager SpeedMenu

Remove File From Project button

Opens the Remove From Project dialog box, where you can select a file that you want to remove from the open project.

The menu equivalents are

- File|Remove from Project
- Remove File on the Project Manager SpeedMenu

Select Unit From List button

Displays the View Unit dialog box, where you can view any unit in the current project.
When you choose a unit, that unit becomes the active page in the Code Editor.
The menu equivalent is View|Units.

Select Form From List button

Displays the View Form dialog box, where you to view any form in the current project.
When you choose a form, it becomes the active form.
The menu equivalent is View|Forms.

Toggle Form/Unit button

Makes the Code Editor the active window when the form is selected and makes the form the active window when the Code Editor is selected.

The menu equivalent is View|Toggle Form/Unit.

New Form button

Creates a blank form and a new unit and adds them to the project.
The menu equivalent is File|New.

Run button

Compiles and executes your application.

The menu equivalent is Run|Run.

Pause button

Pauses program execution and positions the execution point on the next line of code to execute.

Trace Into button

Executes the program statement highlighted by the execution point. Trace Into lets you execute routines, written by the programmer, one statement at a time. When the routine returns from the call the debugger positions the execution point on the statement following the routine call.

If the execution point highlights a routine whose code was generated by Delphi, choosing Trace Into causes the debugger to position the execution point at the statement following the routine call.

Note: The Trace Into button is disabled if symbolic debug information is off.

The menu equivalent is Run|Trace Into.

Step Over button

Executes your program one statement at a time, without branching into subroutines. Stepping through your program is helpful if your program is about to execute a routine whose code you do not need to debug at this time.

If the execution point is located on a call to a routine, then issuing Step Over runs that routine at full speed and places the execution point on the statement following the routine call.

Note: The Step Over button is disabled if symbolic debug information is off.

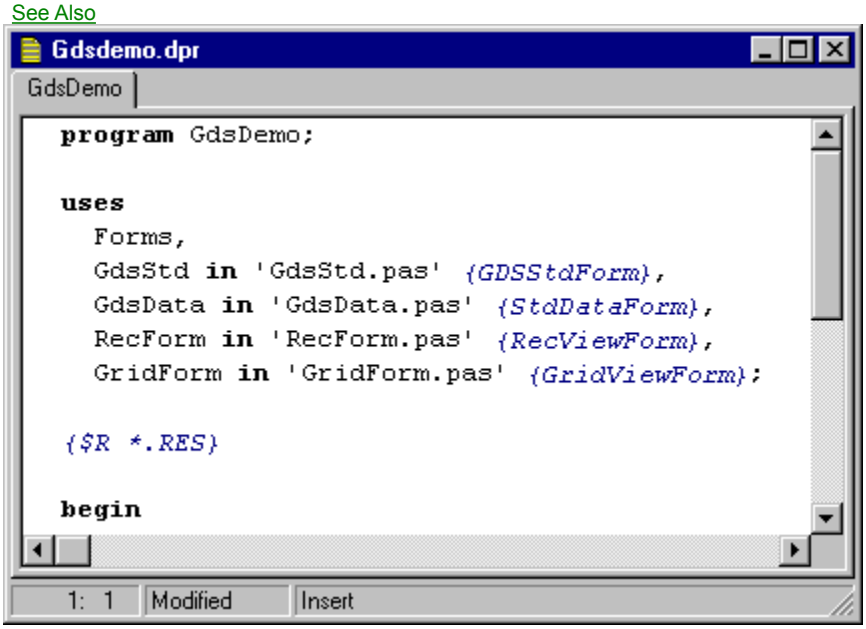
The menu equivalent is Run|Step Over.

Opening a SpeedMenu

- Right-click in the window.
- Press Alt+F10 when the cursor is in the window.

About the Code Editor

See Also



The Code Editor is a full-featured editor. It gives you access to the code that runs your application and offers many powerful features such as:

- Brief-style editing
- Color syntax highlighting
- Multiple and group Undo
- A full range of editing commands

Many commands are available on the Code Editor SpeedMenu.

You can customize the Code Editor by using the Options|Environment dialog box.

You can get Help on a token in the Code Editor by placing your cursor on that token and pressing F1.

When you open a new project, Delphi adds a page in the Code Editor for the main Form unit.

At compile time, if you receive an error, Delphi does the following things:

- Displays the error in the Code Editor message box
- Highlights the offending line

Viewing different files in the Code Editor

- Click its associated tab.

See also

Behind the scenes in the Code Editor
Code basics
Error messages
Object Inspector
Project Manager
Using the Code Editor
Viewing different files in the Code Editor
Working with code

Maximize button
Grows your window to encompass your entire screen.

Code Editor

Enables you to view or modify any part of the source code contained in the active page.

Page tabs

Provides a way to move between the open files in the Code Editor.

Title bar

Displays the name of the active file in the Code Editor.

Line and column indicator

Displays the line and column position of the cursor in the Code Editor. The first and second numbers show the line number and column number, respectively.

Modified indicator

Indicates whether the text in the active page of the Code Editor has been modified since the last time the file was saved. (Blank if the file has not been modified.)

Mode indicator

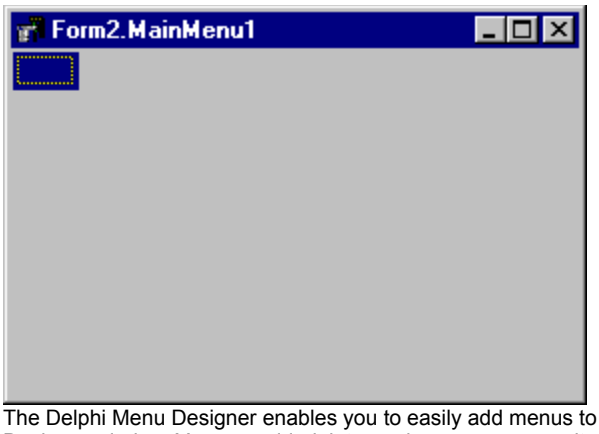
Indicates whether the editor is in Insert or Overwrite mode.

- In Insert mode (the default mode), text you type is inserted at the cursor.
- In Overwrite mode, text you type overwrites previously entered text.

Use the Insert key on your keyboard to toggle between these two modes.

About the Menu Designer

See Also



The Delphi Menu Designer enables you to easily add menus to your form. You can simply add menu items directly into the Menu Designer window. You can add, delete, and rearrange menu items at design time and you do not have to run the program to see the results. Your applications menus are always visible on the Form, as they will appear during run time.
You can build each menu structure entirely from scratch, or you can start from one of the Delphi Menu templates (predesigned menus).

You can also dynamically change menus, to provide more information or options to the user.

For more information about the Menu Designer, choose from the following topics:

Opening the Menu Designer

Editing Menu Items Without Opening the Menu Designer

Menu Designer SpeedMenu

Using Menu Templates

Importing Menus from Resource Files

Accessing and Editing Menus at Run Time

Opening the Menu Designer

1. Place a MainMenu or a PopupMenu component on the form.
2. Leaving the component selected, choose from one of the following methods:

- Double-click the MainMenu or PopupMenu component.
- Click the ellipses button in the Values column for the Items property.
- Select Menu Designer from the component's SpeedMenu.

See also

Accessing and editing menus at run time
Designing menus
TMainMenu component
TPopupMenu component

Menu title area

Menu titles display in this area. Click the highlighted block to add new items to the menu.

Menu command area

Menu commands display in this area. Click the highlighted block to add new menu commands

About the Object Inspector

See also

The Delphi Object Inspector is the gateway between your application's visual appearance and the code that makes your application run.

The Object Inspector enables you to

- Set design-time properties for components you have placed on a form (or for the form itself), and
- Create and help you navigate through event handlers.

The Object selector at the top of the Object Inspector is a drop-down list containing all the components on the active form and it also displays their object type. This lets quickly select different components on the current form..

You can resize the columns of the Object Inspector by dragging the separator line to a new position.

The Object Inspector has two pages:

- Properties page
- Events page

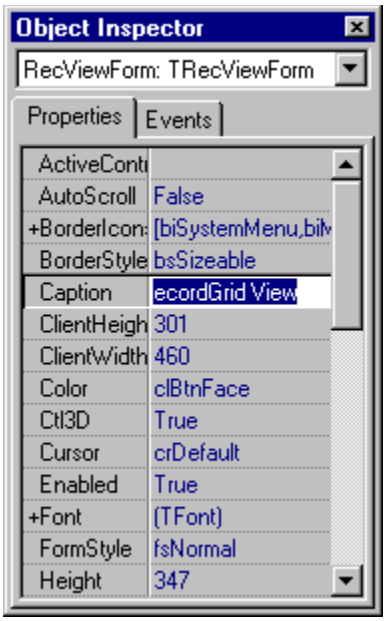
See also

Setting properties

Code basics

About the Object Inspector: Properties page

See Also



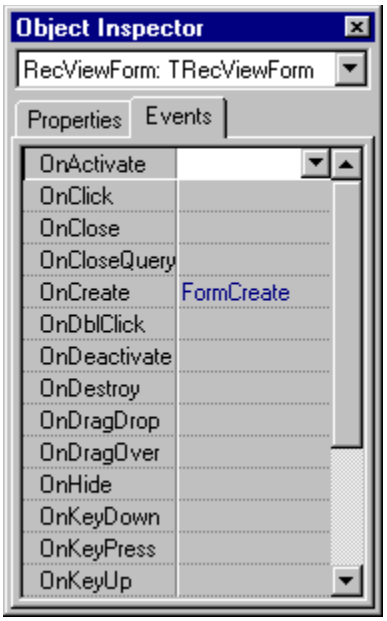
The Properties page of the Object Inspector enables you to set design-time properties for components on your form, and for the form itself. You can set run-time properties by writing source code inside event handlers.
The Properties page displays only the properties of the component that is selected on the form.
By setting properties at design time you are defining the initial state of a component.

See also

Events page
Using property editors
Setting properties
Viewing nested properties

About the Object Inspector: Events page

See Also



The Events page of the Object Inspector enables you to connect forms and components to program events. When you double-click an event from the Events page, Delphi creates an event-handler and switches focus to the Code Editor. In the Code Editor, you write the code inside event-handlers that specifies how a component or form responds to the a particular event.
The Events page displays only the events of the component that is selected in the form.

See also

Displaying and coding shared events
Locating existing event handlers
Properties page
Using property editors

Object selector

Displays the active component whose properties and events you are currently editing. You can use the drop-down list to select a component.

Property column
Lists the design-time properties for the component you have selected on your form.

Value column
Displays the current value for a property. Each value uses one of the Delphi property editors to set values.

Object Inspector tabs

Provide you with a means to switch between the Property page and the Event page of the Object Inspector. To change pages, click a tab.

Minimize button
Shrinks the window down to an icon.

Scroll bars

Provides you with a means to scroll the current window to view objects that cannot fit into the window. To scroll a window, click the up scroll arrow or the down scroll arrow, or you can drag the scroll box.

Control menu

Click the Control-menu box to open the Control menu for the current window.

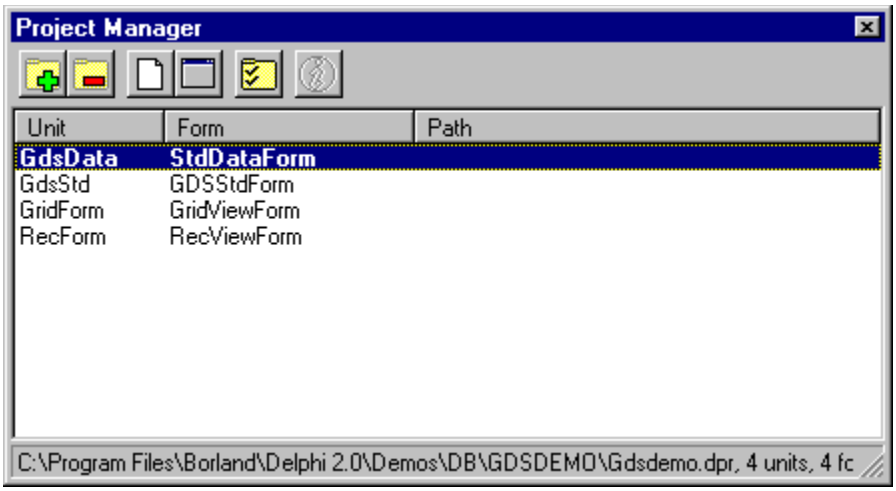
Handler column

Displays the event-handler link for an event. To generate a default event-handler link for an event, double-click the Handler column.

Event column
Lists the possible events for the component you have selected on your form.

About the Project Manager

See Also



The Delphi Project Manager displays the form and unit files associated with your project. These files are listed in the **uses** clause of your .DPR file. The Project Manager also enables you to easily navigate between files while you are developing your application project.

The Project Manager also has a SpeedMenu, which you can display by right-clicking anywhere inside the window area.

When you start Delphi, you can open the Project Manager by choosing View|Project Manager. If you save your desktop settings, you can have the Project Manager window opened by default when you open any project.

If you share files among different projects, using the Project Manager is recommended because you can quickly and easily see the location of each file in the project. This is especially helpful to know when creating backups that include all files the project uses.

Opening the Project Manager

- Choose View|Project Manager.

Note: You must have a project open in order to view the Project Manager.

Closing the Project Manager

- Double-click the Control-menu box.
- Choose Close from the Control menu.

See also

Project Manager SpeedMenu
File handling in the Project Manager
Compiling, building, and running projects
Managing projects and directories
Setting project options
Basic project tasks

Add File button

Opens the Add To Project dialog box, where you can choose a unit (.PAS) or form file (.DFM) to include in the project.

Note: You can open other files from this dialog box, but they will not be added to the project.

Remove File button

Deletes any files currently selected in the Project Manager from the project.

Note: Does not delete the file from the project directory.

View Unit button

Displays the selected file in Code Editor. If the file is not currently open, Delphi will open it.

View Form button

Displays the form for the selected module. This button is dimmed if no form is associated with the selected unit.

Options button

Opens the Project|Options dialog box, where you can specify compiler directives, linker options, output directories, conditional defines, and files to be included with the project.

Update button

Synchronizes the files listed in the Project Manager with the contents of the project file. This button is disabled unless you manually modified the .DPR file.

Unit Name column

Double-click a filename in this column to view the source for the selected unit. If the unit you selected is not currently open, Delphi will open it.

Form Name column

Double-click a filename in this column to view the selected form. If the form you selected is not currently open, Delphi will open it. If this column is empty, there is no form associated with the unit listed in the Unit Name column.

File Status line

Displays the DOS file name of the project file and indicates the number of forms and units included.

Path Name column

Display the path name of the form and unit file. If the unit has not been saved this column is empty.

Designing Menus

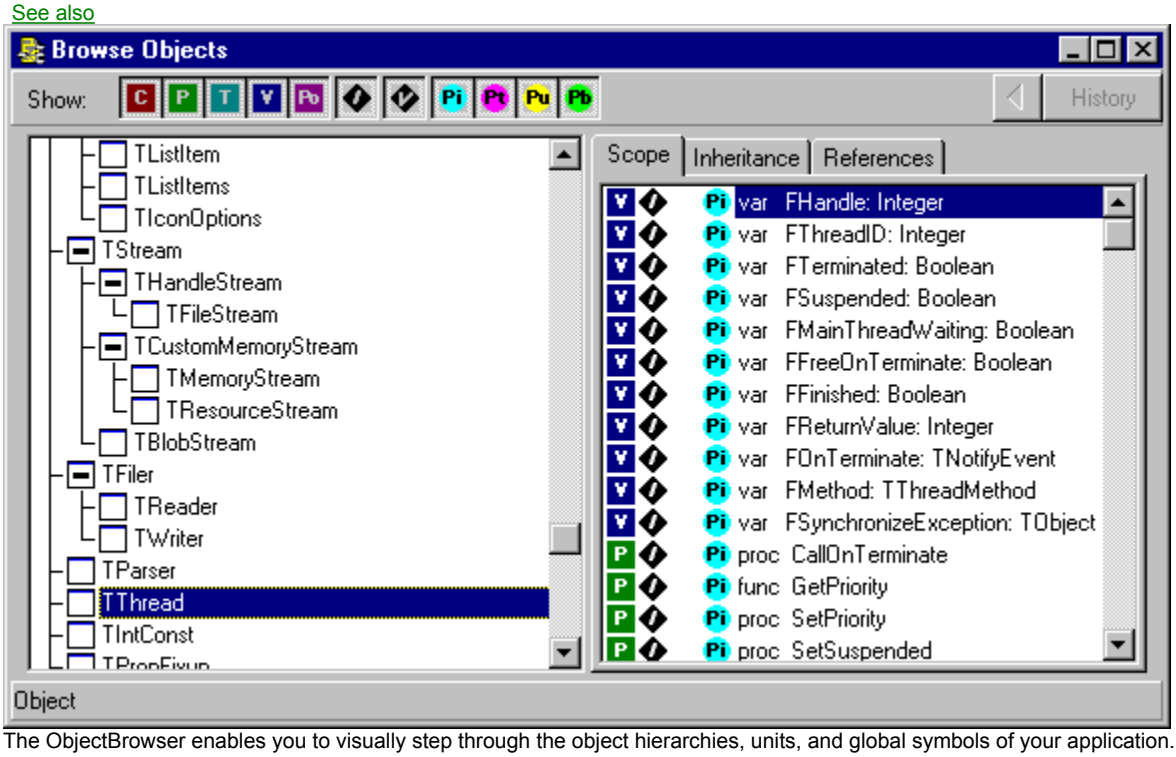
The Delphi Menu Designer enables you to easily add a menu, either predesigned or custom tailored, to your form. You simply add a menu component to the form, open the Menu Designer, and type menu items directly into the Menu Designer window. You can add or delete menu items, or drag and drop them to rearrange them during design time.

You don't even need to run your program to see the results; your design is immediately visible in the form, appearing just as it will during run time.

Your code can also change menus at run time, to provide more information or options to the user. For information, see Accessing and Editing Menus at Run Time..

The ObjectBrowser window

See also



The ObjectBrowser enables you to visually step through the object hierarchies, units, and global symbols of your application. You can use the ObjectBrowser to:

- Graphically view the object hierarchies in your application
- List the units your program uses, and view the symbols declared in the interface part or implementation part based on your compiler settings
- List all global symbols in your application, view their declarations, and go to references in your source code

The ObjectBrowser has a SpeedMenu that enables you to change symbols you are browsing and to choose display options. To access the SpeedMenu, right-click anywhere in the ObjectBrowser window or press Alt+F10.

The ObjectBrowser has two resizable panes, the Inspector pane and the Details pane.

To find out more about any part of the ObjectBrowser window, click an area in the graphic above.

To resize the Inspector pane or the Details pane,

1. Place the cursor between the two panes until the cursor changes to a double-headed arrow.
2. Drag the separator to the left or to the right.

For more information about using the ObjectBrowser, choose one of the following topics:

Browsing symbols

Searching for symbols in the ObjectBrowser

Setting filters

Enabling the ObjectBrowser

See also

Inheritance tree

ObjectBrowser keyboard shortcuts

Options|Environment|Browser

Inspector pane

Displays the symbols for the area that you are browsing. You can select symbols in this pane to inspect the declarations, inheritance information, or references for that symbol.

When you first open the ObjectBrowser, you are browsing objects, and the Inspector pane displays the object inheritance tree. You can change this default on the Browser page of the Environment Options dialog box.

To change what you are browsing in the Inspector,

- Choose Objects, Units, Globals, or Symbol from the ObjectBrowser SpeedMenu.

Details pane

Provides detailed information about the object, unit, global, or symbol selected in the Inspector pane. There are three pages in the Details pane:

Scope page

Lists all the symbols declared in the object, unit, global, or symbol selected in the Inspector pane. Change the display by selecting either Sort Always or Qualified Symbols from the ObjectBrowser SpeedMenu.

Inheritance page

Displays a collapsible local tree of ancestor and descendant objects for the object selected in the Inspector pane.

Reference page

Lists all the file names and line numbers where the symbol you have selected in the Inspector pane appears in your code. If there is no reference information for a symbol, that symbol either was not used in your code or was contained in a unit that was compiled with Symbol Info off. You can double-click any reference to jump to that line in your code.

Show Buttons

Toggles the display of the ObjectBrowser filters, which enable you to limit the symbols you are browsing. To add or remove a filter, click it.

If you selected Show Hints from the ObjectBrowser SpeedMenu, Delphi displays the name of the button if you leave the mouse over the button for longer than one second.

Info Line

Displays the symbol type that you are currently browsing.

You can toggle the Info Line display by choosing Info Line from the ObjectBrowser SpeedMenu.

History list

Click History to open the ObjectBrowser History dialog box, which displays a list of symbols you previously browsed. The most recently browsed symbol is listed first.

To return to a previous view, double-click its name.

Back button

Click Back to rewind to the previous ObjectBrowser view. If you step back, the ObjectBrowser loses all the information for what you were just browsing.

Enabling the ObjectBrowser

See also

The ObjectBrowser provides a means for you to search your code for different symbols.

To open the ObjectBrowser,

- Choose View|Browser.

Before you can start the ObjectBrowser, you need to compile your program. If you do not know how to compile your program, see Compiling, building, and running projects

Whenever you compile a project, you can always use the ObjectBrowser to inspect symbols that were declared in the **interface** part of the module. However, you can add more information by enabling the Debugging options on the Compiler page of the Project Options dialog box. The following table lists the Debugging options and their effect on the ObjectBrowser.

Option	When checked
Debug Information	Adds those symbols declared in the **implementation** part of the module.
Local Symbols	Adds line number references that you can view on the References page of the Details pane.
Symbol Info	Adds all the identifiers that are declared within a routine. That is, symbols are local in scope to a particular routine.

Note: Debug Information must be checked in order for the Local Symbols and Symbol Info options to have any effect.

See also

The ObjectBrowser window
Browsing symbols
ObjectBrowser keyboard shortcuts
Searching for symbols in the browser
Setting filters

Browsing symbols

See also

When you are viewing symbols using the ObjectBrowser, you might want to further inspect them by viewing its declaration, its object hierarchy, or implementation in the source code.

To inspect a symbol from the Inspector pane,

- Select a symbol in the Inspector pane. The contents of the Details pane are updated to reflect the contents of the selected symbol.

Objects, units, globals, and symbols that have their own declarations and can be further inspected, based on the Debugging options on the Compiler page of the Project Options dialog box.

To further inspect a symbol from the Inspector pane,

- Double-click the symbol in the Inspector pane.

  After you select the symbol, the previous contents of the Details pane move to the Inspector pane, and the declarations from the selected symbol are now listed in the Details pane. You can then inspect those symbols.

To view the symbol you are inspecting in your code, do one of the following:

- Select a symbol to browse and press Ctrl+Enter. The Code Editor becomes the active window, and Delphi places the cursor at the point of reference.
- Select a symbol to browse and press the Spacebar to activate source code highlighting. After pressing the Spacebar, you can arrow through the symbols in the active pane to view the symbol reference in the source code. When you encounter a symbol that is not used in the active unit, this command becomes disabled.

  Pressing the Spacebar does not transfer focus from the ObjectBrowser to the Code Editor.

When the selected symbol is a function or procedure, inspecting the code displays the implementation declaration of the routine. When the selected symbol is a constant, property, variable, or type, inspecting the code displays the symbol's declaration.

See also

Enabling the ObjectBrowser
The ObjectBrowser window
ObjectBrowser keyboard shortcuts
Searching for symbols in the ObjectBrowser
Setting filters

Searching for symbols in the ObjectBrowser

See also

ObjectBrowser uses an incremental search in both the Inspector pane and the Details pane. That is, if you want to search for a certain routine, you can type in its name, and the focus of the ObjectBrowser shifts to the symbol that most closely matches what you just typed.

As you type, the ObjectBrowser displays the characters between the filter buttons and the History button. To cancel the current search, press Esc.

Note: The Object Browser does not accept incremental search characters that do not match a symbol in the active pane.

To clear the characters in the incremental search,

- Press Esc.

To find the next match,

- Press Ctrl+N.

See also

Browsing symbols
Enabling the ObjectBrowser
The ObjectBrowser window
ObjectBrowser keyboard shortcuts
Setting filters

Inheritance tree

See also

When you are browsing objects using the ObjectBrowser, the Inspector pane displays an inheritance tree. The inheritance tree is an outline detailing the ancestor/descendant relationship for all the objects used in your application. Objects connected to each other by branches on the tree are inherited from a common base object.

The inheritance tree is also used on the Inheritance page of the Details pane.

Each object node on the inheritance tree has an outline indicator. The outline indicator displays whether or not that node can be expanded.

Indicator	Description
	Node of the tree is collapsed and inherited objects are not listed.
	Node of the tree is expanded and inherited objects are listed.
	No objects are inherited from this object.

To expand or collapse a node of the inheritance tree,

- Click the outline indicator to the left of the node.

See also

The ObjectBrowser window

ObjectBrowser keyboard shortcuts

You can use the following keys to move through the ObjectBrowser:

Key	Action
Up Arrow	Move up one symbol
Down Arrow	Move down one symbol
Ctrl+Up Arrow	Scroll up through the current pane without changing your selected symbol
Ctrl+Down Arrow	Scroll down through the current pane without changing your selected symbol
Ctrl+PgUp	Move to the top of the current pane without scrolling
Ctrl+PgDown	Move to the bottom of the current pane without scrolling
Home	Move to the top of the current pane
End	Move to the bottom of the current pane
Ctrl+Enter	Set focus on the implementation or declaration of the selected symbol
Spacebar	Highlight the source code for the selected symbol, without closing the ObjectBrowser or transferring focus
Enter	Browse the currently selected symbol
+	Expand the current node
-	Collapse the current node
*	Expand the entire tree
PgUp	Move up one screen in the current pane
PgDn	Move down one screen in the current pane
Esc	Cancel incremental search
Ctrl+N	Find next match for the incremental search
Tab	Move from the Inspector pane to the History list button to the Details pane

The following keyboard commands are shortcuts for setting filters and changing views in the Object Browser.

Key	Action
Ctrl+C	Toggles the Constants filter
Ctrl+F	Toggles the Functions and Procedures filter
Ctrl+T	Toggles the Types filter
Ctrl+V	Toggles the Variables filter
Ctrl+P	Toggles the Properties filter
Ctrl+D	Toggles the Inherited filter
Ctrl+A	Toggles the Virtual filter
Ctrl+1	Toggles the Private filter
Ctrl+2	Toggles the Protected filter
Ctrl+3	Toggles the Public filter
Ctrl+4	Toggles the Published filter
Ctrl+B	Steps back
Ctrl+L	Opens the Object Browser History list
Ctrl+O	Switches to the Objects view
Ctrl+U	Switches to the Units view
Ctrl+G	Switches to the Global view
Ctrl+Y	Switches to the Symbols view
Ctrl+S	Switches to the Scope page of the Details pane
Ctrl+I	Switches to the Inheritance page of the Details page
Ctrl+R	Switches to the Reference page of the Details page

Setting filters

See also

You can use the ObjectBrowser to filter for certain symbols. For example, you can choose to view just the variables in the currently selected object.

The following is a list of filters you can set.


Constants


Functions or procedures


Types


Variables


Properties


Inherited


Virtual


Private


Protected


Public


Published

To set a filter, do one of the following:

- Click the filter's associated button.
- Select the symbol filter from the Browser page of the Environment Options dialog box.
- Press the filter's keyboard shortcut.

Note: The glyph next to the symbol in either the Inspector pane or Details pane identifies the symbol type. In some cases more than one glyph can appear next to a symbol. The second letter further describes the symbol, and it appears to the right of the letter identifying the type of symbol.

See also

The ObjectBrowser window
Browsing symbols
ObjectBrowser keyboard shortcuts
Searching for symbols in the ObjectBrowser

ObjectBrowser SpeedMenu

See also

The ObjectBrowser SpeedMenu lets you quickly browse different symbols or change the display of the symbols you are browsing.

To access the ObjectBrowser SpeedMenu,

- Right-click anywhere in the ObjectBrowser window.
- Press Alt+F10.

The items on the ObjectBrowser SpeedMenu are:

Objects	Displays an inheritance tree for all the objects in your application
Units	Displays the units in the current project
Globals	Displays the global symbols in the entire project
Symbol	Opens the Browse Symbols dialog box
Qualified Symbols	Displays the qualified identifier for a symbol
Sort Always	Sorts the symbols alphabetically by name
Show Hints	Toggles the display of Help Hints on the filter buttons
Info Line	Toggles display of the Info Line

See also

The ObjectBrowser window

Objects (ObjectBrowser SpeedMenu)

Choose Objects from the ObjectBrowser SpeedMenu to display, in the Inspector pane, all the objects in your application, arranged as a vertical inheritance tree to show parent-child relationships.

Units (ObjectBrowser SpeedMenu)

Choose Units from the ObjectBrowser SpeedMenu to list, in the Inspector pane, all the units used by your application.

Globals (ObjectBrowser SpeedMenu)

Choose Globals from the ObjectBrowser SpeedMenu to display, in the Inspector pane, all the global information declared in the units used in your application.

Symbol (ObjectBrowser SpeedMenu)

Choose Symbol from the ObjectBrowser SpeedMenu to opens the Browse Symbol dialog box, where you can specify a symbol to browse.

Browse Symbol dialog box

Use this dialog box to browse a specific symbol.

To browse a specific symbol, do one of the following:

- Enter the symbol name in the edit box and click OK.
- Click the down arrow to choose from a list of previously entered symbols and click OK.

You can also use the arrow keys to move through the list box.

When you click OK, the ObjectBrowser places the symbol in the Inspector pane and displays its local information in the Details pane.

You can then go to the source code that defines the symbol or inspect the symbol declaration. For more information on browsing symbols in your code, see Browsing symbols.

Qualified Symbols (ObjectBrowser SpeedMenu)

Choose Qualified Symbols from the ObjectBrowser SpeedMenu to display the qualified identifier for a symbol. When this option is off, only the symbol name is displayed.

You can set this option using the Browser page of the Environment Options dialog box.

Sort Always (ObjectBrowser SpeedMenu)

Choose Sort Always from the ObjectBrowser SpeedMenu to display the symbols in alphabetical order by symbol name. When this option is off, the symbols sort by declaration order.

You can set this option using the Browser page of the Environment Options dialog box.

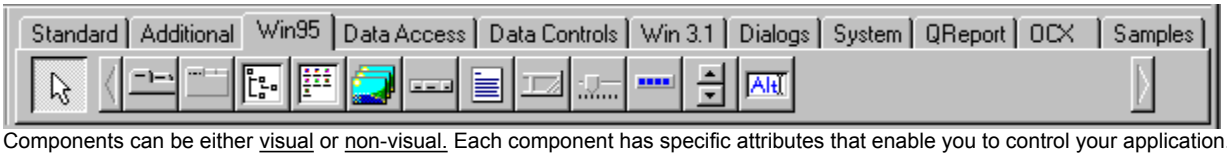
Info Line (ObjectBrowser SpeedMenu)

Choose Info Line from the ObjectBrowser SpeedMenu to toggle the display of the Info Line.

About the Component palette

See also

Components are the building blocks of every Delphi application, and the basis of the Delphi visual component library. Each page tab in the Component palette displays a group of icons representing the components used to design your application interface.



Components can be either visual or non-visual. Each component has specific attributes that enable you to control your application. These attributes are Properties, Events, and Methods.

You can horizontally resize the Component palette by dragging the seperator line which lies between the Component palette and the SpeedBar.

The Component palette provides Help Hints. Help Hints display a small pop-up window containing the name or brief description of the button when your cursor is over the button for longer than one second. To enable Help Hints, select Show Hints from the Component palette SpeedMenu.

Choose one of the following topics for infomation about its components and their functions.

Standard components	Standard Windows components
Additional components	Customized components
Windows 95 components	Windows 95 common components
Data Access components	Database access components
Data Controls components	Data-aware controls
Windows 3.1 components	Windows 3.1 components
System components	System components

Note: The components on the VBX and Samples page are provided as samples only. Source code for the components on the Samples page can be found in the DELPHI 2.0\SOURCE\SAMPLES directory of a default installation.

See also

Component palette SpeedMenu
Customizing the Component Palette
Handling events
Manipulating components
Responding to user events
Setting properties

Standard page components

See also

The components on the Standard page of the Component palette make the standard Windows control elements available to your Delphi applications.

The following illustrations show the Standard components. Click on any component for more information about each component.

See also

Component palette overview
Additional components
Windows 95 components
Data access components
Data controls components
Dialogs components
Windows 3.1 components
System components

MainMenu component

Creates menus for your form.

See the reference page for more information:

TMainMenu reference

PopupMenu component
Creates popup menus for your form.
For more information, see
TPopupMenu reference

Label component

Displays text that the user cannot select or manipulate, such as title text.

For more information, see

TLabel reference

Edit component

Displays an editing area where the user can enter or modify a single line of data.

For more information, see

TEdit reference

Memo component

Displays an editing area where the user can enter or modify multiple lines of data.

For more information, see

TMemo reference

Button component

Creates a pushbutton control that users choose to initiate actions.

For more information, see

TButton reference

CheckBox component

Presents an option that a user can toggle between Yes/No or True/False. You can use check boxes to display a group of choices that are not mutually exclusive. Users can select more than one check box in a group.

For more information, see

TCheckBox reference

RadioButton component

Presents an option that a user can toggle between Yes/No or True/False. You can use radio buttons to display a group of choices that are mutually exclusive. Users can select only one radio button in a group.

For more information, see

TRadioButton reference

ListBox component

Displays a scrolling list of choices.

For more information, see

TListBox reference

ComboBox component

Displays a list of choices in a combined list box and edit box. Users can enter data in the edit box area or select an item in the list box area.

For more information, see

TComboBox reference

ScrollBar component

Provides a way to change the viewing area of a list or form. You can also use a scroll bar to move through a range of values by increments.

For more information, see

TScrollBar reference

GroupBox component

Provides a container to group related options on a form.

For more information, see
TGroupBox reference

Radio group component

Creates a group box that contains radio buttons on a form.

For more information, see

TRadioGroup reference

Panel component

Creates panels that can contain other components on a form. You can use panels to create toolbars and status-lines.

For more information, see

TPanel reference

ScrollBox component

Creates a resizeable container that automatically displays scrollbars when necessary.

For more information, see

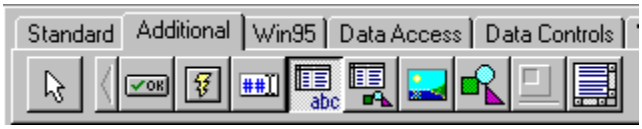
TScrollBox reference

Additional page components

See also

The components on the Additional page of the Component palette make specialized Windows control elements available to your Delphi applications.

The following illustrations show the Additional components. Click on any component for more information about each component.



See also

Component palette overview
Standard components
Additional components
Windows 95 components
Data access components
Data controls components
Dialogs components
System components

BitBtn component

Creates a button component that can display a bitmap.

For more information, see
TBitBtn reference

SpeedButton component

Provides a button that can display a glyph but not a caption. Speed buttons are often grouped within a panel to create a tool bar.

For more information, see

TSpeedButton reference

MaskEdit component

Similar to an edit component, but provides a means to specify particular formats for data entry and display.

For more information, see

TMaskEdit reference

StringGrid component

Creates a grid that you can use to display string data in columns and rows.

For more information, see
TStringGrid reference

DrawGrid component

Creates a grid that you can use to display data in columns and rows.

For more information, see
TDrawGrid reference

Image component

Displays a bitmap, icon, or metafile.

For more information, see

TImage reference

PaintBox component

Specifies a rectangular area on a form that provides boundaries for application painting.

For more information, see
TPaintBox reference

Shape component

Draws geometric shapes including an ellipse or circle, a rectangle or square, or a rounded rectangle or rounded square.

For more information, see

TShape reference

Bevel component

Creates lines or boxes with a three-dimensional, chiseled appearance.

For more information, see

TBevel reference

Windows 95 page components

See also

The components on the Win95 page of the Component palette provide access to Win95 user interface common controls available to your Delphi applications.

The following illustrations show the Window 95 components. Click on any component for more information about each component.

See also

Component palette overview
Additional components
Data access components
Data controls components
Windows 3.1 components
Dialogs components
System components

TabControl component

A tab control is analogous to a divider in a file cabinet or notebook. This component functions similar to TTabSet. To create a multiple page dialog box, use a TPageControl. To use simply a tab set, use the TTabControl.

For more information, see

TTabControl reference

PageControl component

A page set which is used to make a multiple page dialog box. A tab control is analogous to a divider in a file cabinet or notebook. You can use this control to define multiple logical pages or sections of information within the same window.

For more information, see

TPageControl reference

TreeView component

A tree view component allows you to control and display a set of objects as an indented outline based on their logical hierarchical relationship. The control includes buttons that allow the outline to be expanded and collapsed. You can use a tree view component to display the relationship between a set of containers or other hierarchical elements.

For more information, see

TTreeView reference

ListView component

The ListView component provides a way to display a list in columns. List views display data in a variety of views.

For more information, see

TListView reference

ImageList component

An image list is a collection of same-sized images, each of which can be referred to by its index. Image lists are used to efficiently manage large sets of icons or bitmaps. All images in an image list are contained in a single, wide bitmap in screen device format. An image list may also include a monochrome bitmap that contains masks used to draw images transparently (icon style).

For more information, see

TImageList reference

HeaderControl component

Using a header control, you can display a heading above columns of text or numbers. You can divide the control into two or more parts to provide headings for multiple columns. You can align the title elements left, right, or centered. You can configure each part to behave like a command button to support a specific function when the user clicks on it.

For more information, see

THeaderControl reference

RichEdit component

Rich Text Format memo control. By default, the rich text editor supports font properties, such as typeface, size, color, bold, and italic format, format properties, such as alignment, tabs, indents, and numbering Automatic drag and drop of selected text.

For more information, see

TRichEdit reference

StatusBar component

Area to post the status of actions at the bottom of the screen.

For more information, see

TStatusBar reference

TrackBar component

The Trackbar component is a control that consists of a bar that defines the extent or range of the adjustment, and an indicator that both shows the current value for the control and provides the means for changing the value.

Trackbars support a number of options. You can set the trackbar orientation as vertical or horizontal, define the length and height of the slide indicator and the slide bar component, define the increments of the trackbar, and whether to display tick marks for the control.

For more information, see

TTrackBar reference

ProgressBar component

A Progress Bar component consists of a rectangular bar that “fills” from left to right. An example of this component can see in Windows 95 when you copy files in the Windows Explorer. Use this control as feedback for long operations or background processes. The control provides visual feedback to the user about the progress of a process.

For more information, see

TProgressBar reference

UpDown component

Up and down arrow buttons to increment and decrement values.

For more information, see

TUpDown reference

HotKey component

Attaches a hot key to any component.

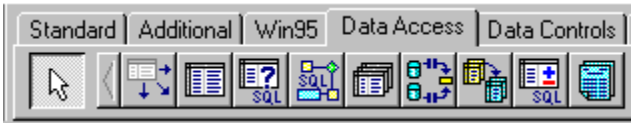
For more information, see
THotKey reference

Data Access page components

See also

The components on the Data Access page of the Component palette make specialized database access elements available to your Delphi applications.

The following illustration shows the Data Access components. Click on any component ny component for more information about each component.



See also

Component palette overview
Standard components
Additional components
Windows 95 components
Data controls components
Windows 3.1 components
Dialogs components
System components

DataSource component

Acts as a conduit between a TTable, TQuery, TStoredProc component and data-aware components, such as TDBGrid.

For more information, see

TDataSource reference

Table component

Retrieves data from a physical database table via the BDE and supplies it to one or more data-aware components through a TDataSource component. Sends data received from a component to a physical database via the BDE.

For more information, see

TTable reference

Query component

Uses SQL statements to retrieve data from a physical database table via the BDE and supplies it to one or more data-aware components through a TDataSource component, or uses SQL statements to send data from a component to a physical database via the BDE.

For more information, see

TQuery reference

StoredProc component

Enables an application to access server stored procedures. Sends data received from a component to a physical database via the BDE.

For more information, see

TStoredProc reference

Database component

Sets up a persistent connection to a database, especially a remote database requiring a user login and password.

For more information, see

TDatabase reference

Session component

A TSession component provides global control over a group of TDatabase components you associate with it. A default TSession component is automatically created for each Delphi database application. You only need to use the TSession component if you are creating a multi-threaded database application. Each database thread requires its own session component.

For more information, see

TSession reference

BatchMove component

Copies a table structure or its data. Can be used to move entire tables from one database format to another.

For more information, see

TBatchMove reference

Update SQL

The TUpdateSQL component provides a way to use Delphi's cached updates support with read-only datasets. For example, you could use a TUpdateSQL component with a "canned" query to provide a way of updating the underlying datasets, essentially giving you the ability to post updates to a read-only dataset. You associate a TUpdateSQL component with a dataset by setting the dataset's UpdateObject property. The dataset automatically uses the TUpdateSQL component when cached updates are applied.

TUpdateSQL reference

Report component

Enables printing and viewing of database reports through ReportSmith.

For more information, see

TReport reference

Data Controls page components

See also

The components on the Data Controls page of the Component palette make specialized database control elements available to your Delphi applications.

The following illustration shows the Data Controls components. Click on any component for more information about each component.

See also

Standard components
Additional components
Windows 95 components
Windows 3.1 components
Dialogs components
System components

DBLookupListBox component

DBLookupListBox is a data-aware list box that derives its list of display items from one of two sources:

- Lookup field defined for a dataset.
- Secondary data source, data field, and key.

In either case, a user is presented with a restricted list of choices from which to set a valid field value. When a user selects a list item, the corresponding field value is changed in the underlying dataset.

Specifying a list with a lookup field

To specify list box items using a lookup field, the dataset to which you link the control must already define a lookup field. For more information about defining a lookup field for a dataset, see "Defining a lookup field" in Chapter 7, "Working with Fields" in the *Database Application Developer's Guide*.

For more information, see

TDBLookupListBox reference

DBLookupComboBox component

DBLookupComboBox is a data-aware combo box that derives its drop-down list of display items from one of two sources, a:

- Lookup field defined for a dataset.
- Secondary data source, data field, and key.

In either case, a user is presented with a restricted list of choices from which to set a valid field value. When a user selects a list item, the corresponding field value is changed in the underlying dataset. To specify combo box list items using a lookup field, the dataset to which you link the control must already define a lookup field. For more information about defining a lookup field for a dataset, see "Defining a lookup field," in Chapter 7, "Working with fields," in the Database Application Developer's Guide.

For more information, see

TDBLookupComboBox reference

DBCtrlGrid component

A DBCtrlGrid control displays multiple fields in multiple records in a tabular grid format. Each cell in the grid displays multiple fields from a single record.

For more information, see:

TDBCtrlGrid reference

DBGrid component

Data-aware custom grid that enables viewing and editing data in a tabular form similar to a spreadsheet. Makes extensive use of TField properties (set in the Fields editor) to determine a column's visibility, display format, ordering, etc.

For more information, see

TDBGrid reference

DBNavigator component

Data-aware navigation buttons that move a table's current record pointer forward or backward. The navigator can also place a table in Insert, Edit, or Browse state, post new or modified records, and retrieve updated data to refresh the display.

For more information, see

TDBNavigator reference

DBText component

Data-aware label that displays a field value in the current record.

For more information, see

TDBText reference

DBEdit component

Data-aware edit box that displays or edits a field in the current record.

For more information, see

TDBEdit reference

DBMemo component

Data-aware memo box that displays or edits BLOB text in the current record.

For more information, see

TDBMemo reference

DBImage component

Data-aware image box that displays, cuts, or pastes bitmapped BLOB images to and from the current record.

For more information, see

TDBImage reference

DBListBox component

Data-aware list box that displays a scrolling list of values from a column in a table.

For more information, see

TDBListBox reference

DBComboBox component

Data-aware combo box that displays or edits a scrolling list of values from a column in a table.

For more information, see

TDBComboBox reference

DBCheckBox component

Data-aware check box that displays or edits a Boolean data field from the current record.

For more information, see

TDBCheckBox reference

DBRadioGroup component

Data-aware group of radio buttons that display or set column values.

For more information, see

TDBRadioGroup reference

Windows 3.1 page components

See also

The components on the Windows 3.1 page of the Component palette provide Windows 3.1 control elements available to your Delphi applications.

The following illustration shows the Windows 3.1 components. Click on any component for more information about each component.

See Also

Component palette overview
Additional components
Windows 95 components
Data access components
Data controls components
Dialogs components
System components

DBLookupList component

Data-aware list box that displays values looked up from columns in another table at run time.

For more information, see

TDBLookupList reference

DBLookupCombo component

Data-aware combo box that displays values looked up from columns in another table at run time.

For more information, see

TDBLookupCombo reference

TabSet component

Creates notebook-like tabs. You can use the TabSet component with the Notebook component to enable users to change pages.

For more information, see

TTabSet reference

Outline component

Displays information in a variety of outline formats.

For more information, see

TOutline reference

Header component

Creates a sectioned region for displaying data. Users can resize each section of the region to display different amounts of data.

For more information, see

THeader reference

TabbedNotebook component

Creates a component that contains multiple pages, each with its own set of controls. Users select a page by clicking the tab at the top of the page.

For more information, see

TTabbedNotebook reference

Notebook component

Creates a component that can contain multiple pages. You can use the TabSet component with the Notebook component to enable users to change pages.

For more information, see

TNotebook reference

Dialogs page components

See also

The components on the Dialogs page of the Component Palette make the Windows common dialog boxes available to your Delphi applications. The common dialog boxes provide a consistent interface for file operations such as opening, saving, and printing files.

The following illustration shows the Dialogs components. Click on any component for more information about each component.

A common dialog box opens when its Execute method is called. Execute returns one of the following Boolean values:

- If the user chooses OK to accept the dialog box, Execute returns True.
- If the user chooses Cancel or escapes from the dialog box without saving any changes, Execute returns False.

Each common dialog box component (except the PrinterSetup component) has a set of properties grouped under Options in the Object Inspector. The Options properties affect the appearance and behavior of the common dialog boxes. To display the Options properties, double-click "Options" in the Object Inspector.

- To close a dialog box programmatically, use the CloseDialog method.
- To manipulate the position of a dialog box at run time, use the Handle,Left, Top, and Position properties.

See also

Component palette overview
Creating dialog boxes
Standard components
Additional components
Windows 95 components
Data controls components
Windows 3.1 components
System components

OpenDialog component

Makes a Windows common Open dialog box available to your application. Users can specify the name of a file to open in this dialog box.

For more information, see

TOpenDialog reference

SaveDialog component

Makes a Windows common Save dialog box available to your application. Users can specify the name of a file to save in this dialog box.

For more information, see

TSaveDialog reference

FontDialog component

Makes a Windows common Font dialog box available to your application. Users can specify font, size, and style information in this dialog box.

For more information, see

TFontDialog reference

ColorDialog component

Makes a Windows common Color dialog box available to your application. Users can specify color information in this dialog box.

For more information, see

TColorDialog reference

PrintDialog component

Makes a Windows common Print dialog box available to your application. Users can specify printing information in this dialog box, such as a range of pages and the number of copies.

For more information, see

TPrintDialog reference

PrinterSetupDialog component

Makes a Windows common Printer Setup dialog box available to your application. Users can change and set up printers in this dialog box.

For more information, see

TPrinterSetupDialog reference

FindDialog component

Makes a Windows common Find dialog box available to your application. Users can specify a string to search for in this dialog box.

For more information, see

TFindDialog reference

ReplaceDialog component

Makes a Windows common Replace dialog box available to your application. Users can specify a search string and a replacement string in this dialog box.

For more information, see

TReplaceDialog reference

System page components

See also

The components on the System page of the Component palette make specialized system control elements available to your Delphi applications.

The following illustrations show the System components. Click on any component for more information about each component.

See also

Component palette overview
Additional components
Windows 95 components
Data access components
Data controls components
Dialogs components
Windows 3.1 components
System components

Timer component

Provides way to tie events to timed intervals. This is a non-visual component.

For more information, see

TTimer reference

FileListBox component

Displays a scrolling list of files in the current directory.

For more information, see
TFileListBox reference

DirectoryListBox component

Displays the directory structure of the current drive. Users can change directories in a directory list box.

For more information, see

TDirectoryListBox reference

DriveComboBox component

Displays a scrolling list of available drives.

For more information, see
TDriveComboBox reference

FilterComboBox component

Specifies a filter or mask to display a restricted set of files.

For more information, see
TFilterComboBox reference

MediaPlayer component

Displays a VCR-style control panel for playing and recording multimedia video and sound files.

For more information, see

TMediaPlayer reference

OleContainer component

Creates an Object Linking and Embedding (OLE) client area in a form.

For more information, see
TOleContainer reference

DdeClientConv component

Establishes a client connection to a Dynamic Data Exchange (DDE) server application. This is a non-visual component.

For more information, see

TDdeClientConv reference

DdeClientItem component

Specifies the Dynamic Data Exchange (DDE) client data to transfer during a DDE conversation. This is a non-visual component.

For more information, see

TDdeClientItem reference

DdeServerConv component

Establishes a server connection to a Dynamic Data Exchange (DDE) client application. This is a non-visual component.

For more information, see

TDdeServerConv reference

DdeServerItem component

Specifies the Dynamic Data Exchange (DDE) server data to transfer during a DDE conversation. This is a non-visual component.

For more information, see

TDdeServerItem reference

Customizing the Component palette

See also

To customize the layout of the Component palette, choose the Palette page from the Environment Options dialog box.

Saving a customized Component palette

1. Open the Preferences page of the Environment Options dialog box.
2. Check Desktop from the Autosave options.
3. Click OK

Rearranging Component palette pages

1. Open the Palette page of the Environment Options dialog box.
2. Select a page from the Pages list box.
3. Click the up arrow or down arrow, or drag and drop the page to its new location.
4. Click OK for your changes to take affect.

Rearranging components on the Component palette

1. Open the Palette page of the Environment Options dialog box.
2. Select a component from the Components list box.
3. Click the up arrow or down arrow, or drag and drop the component in its new location.
4. Click OK for your changes to take affect.

Moving components to a different Component palette page

1. Open the Palette page of the Environment Options dialog box.
2. Drag and drop the component from the Components list box onto a page in the Pages list box.
3. Click OK for your changes to take affect.

Note: When you move a component to a new page, the component is added as the last item on the page.

Renaming Component palette pages

1. Open the Palette page of the Environment Options dialog box.
2. Select the page from the Pages list box.
3. Click Rename to open the Rename Page dialog box.
4. Enter a new name.
5. Click OK to close the Rename Page dialog box.
6. Click OK for your changes to take affect.

Renaming a component

1. Open the Palette page of the Environment Options dialog box.
2. Select the component from the Components list box.
3. Click the Rename button.
3. Click Rename to open the Rename Page dialog box.
4. Enter a new name.
5. Click OK to close the Rename Page dialog box.
6. Click OK for your changes to take affect.

Adding pages to the Component palette

1. Open the Palette page of the Environment Options dialog box.
2. Click the Add button to open the Add Page dialog box.
3. Enter a new page name.
4. Click OK to close the Add Page dialog box.
5. Click OK for your changes to take affect.

Removing pages from the Component palette

1. Open the Palette page of the Environment Options dialog box.
2. Select the page from the Pages list box
3. Press Delete.
4. Click OK for your changes to take affect.

Note: Before you can remove a page, it must be empty of components.

Removing components from the Component palette

1. Open the Palette page of the Environment Options dialog box.
2. Select the component you want to remove.
3. Press Delete.
4. Click OK for your changes to take affect.

See also

Component palette
About the component library
Adding components to the component library
Removing components from the component library

About user events

See also

In event-driven programming, user events are a key part of your application logic. User events correspond to the elements in your Graphical User Interface (GUI). For example, most components contain an OnClick event that can be programmed to respond when the user clicks the component. Other events, such as the form's OnActivate, are not triggered by the user, but by your program code. The code you write to respond to events is called an Event Handler.

For information about handling specific user events, choose from the following topics.

Button click events

Keyboard events

Mouse events

Drag and drop events

See also

Designing a user interface

Handling events

About keyboard events

See also

Delphi provides three events that enable you to capture user keystrokes:

- OnKeyDown
- OnKeyPress
- OnKeyUp

You can write event handlers for these events to respond to any key or key combination the user might press at run time.

Note: Responding when the user presses short-cut or accelerator keys, such as those provided with menu commands, does not require writing event handlers.

There are several important considerations when handling keyboard events. Choose a topic for more information.

Keyboard event processing order

Processing keystrokes

Redirecting keyboard events to the form

See also

Responding to user events
Specifying accelerator keys
Specifying keyboard shortcuts
TShiftState

Keyboard event processing order

See also Example

Keyboard events are received at several levels:

- The application level, with an Application.OnMessage event.

  You will rarely need to intercept keystrokes at the application level, but it is important to know that this first level is available.

- The "shortcut-key" level

  When you specify a short-cut key, such as those provided as a property of menu items, the keystroke is intercepted before the form sees it.

- The form level

  The form contains a KeyPreview property that enables you to code "global" keystroke events.

- The component level

  When you program key-press event handlers at the component level, the component with focus intercepts the keystroke.

See also

Responding to keyboard events
Processing keystrokes
Redirecting keyboard events to the form

Example

The following code uses two buttons in a form to demonstrate the order in which keyboard events are processed by your application.

When you first run the program and press Alt+Ctrl, the form turns purple, because Button1 had focus and therefore receives the keystrokes. If you click Button1 to disable it, or click Button2 to set the form's KeyPreview on, and then press Ctrl+Alt again, the form turns aqua because the form receives the keystrokes.

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
 if (shift = ([ssAlt, ssCtrl])) then form1.color := clAqua;
end;

procedure TForm1.Button1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
   if (shift = ([ssAlt, ssCtrl])) then form1.color := clPurple;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
 button1.enabled := false;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
 Form1.Keypreview := True;
end;
```

Processing keystrokes

See also Example

Every keystroke generates an OnKeyDown and OnKeyUp event. In addition, keys that have an ASCII equivalent generate an OnKeyPress event. Naturally, your code does not need to capture each keystroke, but it is important to know the order in which keystroke events are processed. See the attached example for an explicit demonstration.

Your application can capture each event at several levels (see KeyBoard event processing order). OnKeyPress returns a single ASCII character, while OnKeyDown and OnKeyUp contain parameters that reflect information about whether control keys such as Alt, Ctrl and Shift keys were pressed at the time the last keystroke occurred.

For instance, here is the OnKeyDown, OnKeyPress and OnKeyUp keystroke sequence generated when the user presses Shift+D:

```
KeyDown (Shift)
KeyDown (Shift+D)
KeyPress (D)
KeyUp (Shift+D)
KeyUp
```

When keys are pressed in combination, the OnKeyDown event passes the key for each previous OnKeyDown to the next OnKeyDown. The OnKeyPress event, by contrast, merely returns the last key pressed. However, OnKeyPress returns a different ASCII character for 'd' and 'D,' but OnKeyDown and OnKeyUp do not make a distinction between uppercase and lowercase alpha keys.

See also

Responding to keyboard events
Keyboard event processing order
Redirecting keyboard events to the form

Example

The following code uses a list box to display the keystroke processing order of the OnKeyUp, KeyDown, and OnKeyPress events of the form and the Edit component for any key you press.

Note: By adding a Default and Cancel button to the form (buttons whose Default and Cancel properties are set to True, respectively) you can view how the Esc and Enter keystrokes are processed when such buttons exist.

```
procedure TForm1.Edit1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
 Listbox1.Items.Add('Edit1.KeyDown'+ShortCutToText(ShortCut(Key, Shift)));
end;

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
   Listbox1.Items.Add('Edit1.KeyPress'+ Key);
end;

procedure TForm1.Edit1KeyUp(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
 Listbox1.Items.Add('Edit1.KeyUp'+ShortCutToText(ShortCut(Key, Shift)));
end;

procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  Listbox1.Items.Add('Form1.KeyDown'+ShortCutToText(ShortCut(Key, Shift)));
end;

procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
   Listbox1.Items.Add('Form1.KeyPress'+ Key);
end;

procedure TForm1.FormKeyUp(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  Listbox1.Items.Add('Form1.KeyUp'+ShortCutToText(ShortCut(Key, Shift)));
end;
```

Responding to the OnKeyPress event

See also Example

The OnKeyPress event is the simplest of the three events, in that it returns only a single character the user presses. The character must fall within the ASCII character set.

OnKeyPress interprets key combinations only insofar as they evaluate to a single ASCII-character. So, for example, OnKeyPress recognizes the result of Shift+A (a capital 'A'), but not the individual keys pressed. OnKeyPress does not otherwise evaluate keys combinations, and does not recognize function keys or non-ASCII keys such as Ctrl, Alt, Insert, Page Down, and so on.

- Default or Cancel buttons in the form will intercept the Enter and Escape key press for both the OnKeyPress and OnKeyDown events, (unless used in combination with Alt for the OnKeyDown.)

See also

Responding to keyboard events
Keyboard event processing order
Processing keystrokes
Redirecting keyboard events to the form

Example

The following example demonstrates how OnKeyPress and OnKeyDown can be coded to handle keyboard events.

The OnKeyPress event contains a key parameter of type Char. Therefore if you want to test for which key the user pressed, you simply enter the character.

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
 case key of
   'I': Panel1.Caption := 'Shift+I was pressed';
   'c': Panel1.caption := 'c was pressed';
   ' ': Panel1.caption := 'the space bar was pressed';
 end;
end;
```

With OnKeyDown, the key parameter is of type Word. Therefore if you want to test for which key the user pressed, you must refer to the equivalent Virtual Key Codes (see the Win32 Programmer's Reference).

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  case key of
   vk_Insert: Panel1.Caption := 'INS';
   vk_Capital: Panel1.caption := 'CAP';
   vk_Numlock: Panel1.caption := 'NumLock';
  end;
end;
```

Responding to the OnKeyUp and OnKeyDown events

See also Example

Use the OnKeyDown and OnKeyUp events when you want to interpret key combinations such as whether the SHIFT, CTRL, or ALT key is pressed at the time the active control receives the key event; and to handle keys that have no ASCII equivalent, such as function keys. The F1 key, for example, does not get captured by the OnKeyPress event because it has no alphanumeric value.

While both key events return the value of the keys pressed, the OnKeyDown event is much more commonly used. You might use OnKeyUp when you want to initiate a background process inbetween the key-down and key-up. When the user presses and holds down a key, the key returns repeated OnKeyDown events until the user releases it, at which time a single OnKeyUp is returned. In programs such as games, these specific keyboard interactions become more useful. 'Vk_Insert' is the virtual key code for the Insert key.

Note: To use OnKeyDown or OnKeyUp to test for keys the user presses, you must use Virtual key codes (see the Win32 Programmer's Reference) to specify the key. This is because the parameter key is of type word. For example, the following event handler specifies that when the user presses the Insert key, the panel in the form displays 'INS.' '

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if key = vk_Insert then Panel1.Caption := 'INS';
end;
```

See also

Responding to keyboard events
Keyboard event processing order
Processing keystrokes
Redirecting keyboard events to the form

Example

The following two event handlers respond to the OnKeyDown and OnKeyUp events by zooming and shrinking a graphical image with the user presses and releases the 'Z' key.

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if chr(Key) = 'Z' then Image1.Stretch := True;
end;

procedure TForm1.FormKeyUp(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if chr(Key) = 'Z' then image1.Stretch := False;
end;
```

Redirecting keyboard events to the form

Example

- Set the KeyPreview property of a form to True.

You can now handle keyboard events at the form level, rather than having to write separate event handlers for every component in the form that might have focus when the keyboard event occurs. The form can receive any keystrokes that the focused component can receive. Also, by using KeyPreview, you can then code unique keyboard event handlers for specific components.

KeyPreview is like having an automatic call to the form-level keyboard event handler at the start of every component-level keyboard handler. The component still sees the event, but the form has an opportunity to handle it first. For example, you could write an event handler for the FormKeyDown that performs key mappings so that a ButtonKeyDown receives the mapped key instead of the key originally pressed.

See also

Responding to keyboard events
Keyboard event processing order
Processing keystrokes

Example

The following example demonstrates how the form KeyPreview property intercepts keystrokes.

- Place a button on a form, and write the following handler for the form OnKeyDown event:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
 if (shift = ([ssAlt, ssCtrl])) then Form1.color := clAqua;
end;
```

- Run the application and press Alt+Ctrl.

Nothing happens -- this is because the control with focus receives the keyboard events, and in this case no keyboard event handler was written for the button.

You could solve this by disabling the button, or by putting the code in the button event handler. But by setting KeyPreview for the form to True, you can intercept keystrokes at the form level instead of writing additional code.

- Set the form's KeyPreview to True, and run the program again. This time the form receives the keystrokes -- when you press Ctrl+Alt, the form turns aqua.

Common component tasks

See also

When designing your application interface, there are procedures you may want to perform that are not specific to a particular component. The list below represents a sampling of such procedures. Choose a topic for more information.

Tasks

Providing Help Hints

Aligning components

Handling keyboard events

Handling mouse events

Handling drag and drop events

Setting the focus

Setting the tab order

Enabling and disabling components

Changing the Z-order of components

See also
Customizing the Component palette

Using the Form component

See also

An application usually contains multiple forms: A main form, which is the primary user interface, and other forms such as dialog boxes, secondary windows (for instance, those that display OLE 2.0 data), and so on. You can begin your form design from one of the many Form templates provided with Delphi. And, you can save any form you design as a template that you can reuse in other projects.

Working with forms

Adding a field to a form object

Creating dialog boxes

Reusing forms as DLLs

Making your form a component

Tasks

- To have the form stay on top of other open windows (for instance, the Project Manager or Alignment Palette) at run time , set the FormStyle property to fsStayOnTop.
- To remove the form's default scrollbars, change the value of the HorzScrollBar, and VertScrollBar properties.
- To make the form an MDI frame or MDI child, use the FormStyle property.
- To change the form's border style, use the BorderIcons and BorderStyle properties. (The results are visible at run time.)
- To change the icon for the minimized form, use the Icon property.
- To specify the initial position of a form in the application window, use the Position property.
- To specify the initial state of the form, (e.g., minimized, maximized or normal) use the WindowState property.
- To define the working area of the form at run time, use the ClientHeight and ClientWidth properties. (Note that ClientHeight and ClientWidth represent the area within the form's border; Height and Width represent the entire area of the form.)
- To specify which control has initial focus in the form at run-time, use the ActiveControl property.
- To pass all keyboard events to form, regardless of the selected control, use the KeyPreview property.
- To specify a particular menu, if your form contains more than one menu, use the Menu property.

See also

Customizing the Component palette
Form basics
TForm reference

About the TabSet component

See also TTabSet reference

Purpose

Use the TabSet component to create clickable tabs. You can use the TabSet component in conjunction with the Notebook component to create a multi-page dialog box or editor.

The Tabs property of the TabSet component, which contains the list of tabs, is a string list. You can perform many operations on string lists by using the methods and properties of the list. See Working with String Lists.

Tasks

Creating a tablist

Attaching tabs to a notebook component

Displaying a graphic on a tab

Responding to tab set changes

Tasks for selecting and locating tabs

- To select a specific tab, use the TabIndex property.
- To select the tab to either the right or left of the selected tab, use the SelectNext method.
- To scroll to a specific tab, use the FirstIndex property. Note that this does not change the selected tab, so the selected tab might not remain visible. This property is used more commonly to read a value than to write one.
- To retrieve the position of a mouse click on the tab set, use the ItemAtPos method. You can then pass the integer returned by ItemAtPos to the ItemRect method to return the rectangular area bounding the tab. This is useful, for example, for drawing into the tab.
- To determine the number of tabs currently visible in the set, use the VisibleTabs property (read-only at run time).

Tasks to specify the tab's appearance

- To specify the color of the area behind the tabs, use the BackgroundColor and DitherBackground properties.
- To specify the color of the tabs, use the SelectedColor and UnselectedColor properties.
- To specify the amount of visual space between the tab set and its outer borders, use the EndMargin and StartMargin properties.
- To change the height of the tabs, use the TabHeight property. (Note that Delphi implements the setting for this property only when the TabSet's Style property is set to OwnerDraw.)

See also

Common component tasks
TNotebook component
OnDrawTab event
OnMeasureTab event

About the Notebook component

See also TNotebook reference

Purpose

Use the Notebook component to create a single form that contains multiple pages, each of which can contain its own components. To enable the user to navigate between the notebook pages at run time, you can use a TabSet component to add a tab to each page.

The Pages property, which contains the list of notebook pages, is a string list. You can perform many operations on string lists by using the methods and properties of the list. See Working with string lists.

At design time, you can access pages of the Notebook component by changing the ActivePage property.

Tasks

Manipulating Notebook pages

Attaching tabs to a Notebook Component

- To display the current page using its position (an integer), use the PageIndex property.
- To display the current page using its name (a string), use the ActivePage property.
- To access the different notebook pages at design time, use the Object Inspector to change the ActivePage property.

See also

Common component tasks
TComboBox component
TListBox component
TMainMenu
TTabSet component

About the TabbedNotebook component

See also TTabbedNotebook reference

Purpose

Use the TabbedNotebook component to create a notebook component that has tabbed pages already built in. Contrast with the Notebook and TabSet components, which you can use together to create a component similar to the TabbedNotebook, but with more customization options -- for instance, the position of the tabs.

The Pages property, which contains the list of notebook pages, is a string list. You can perform many operations on string lists by using the methods and properties of the list. See Working with string lists.

Tasks

Manipulating Notebook pages

- To display the current page by specifying its position (an integer), use the PageIndex property.
- To display the current page by specifying its name (a string), use the ActivePage property.
- To access the different notebook pages at design time, use the Object Inspector to change the ActivePage property.
- To specify the number of tabs that appear in a row, use the TabsPerRow property.
- To change the default font that displays on the tabs, use the TabFont property.

See also

Common component tasks

Attaching tabs to a Notebook component

About the Outline component

See also TOutline reference

Purpose

Use the Outline component when you want to display a hierarchical relationship between related data. Each item of the hierarchical grouping is represented by a node, and each node may have subordinate items.

The Lines property of the outline, which holds the text for the outline node, is a string list. You can perform many operations on string lists by using the methods and properties of the list. See Working with string lists.

Tasks

Creating an outline structure

Responding to outline changes

- To access the outline nodes programmatically, use the Items property.
- To display a custom glyph on a node, use the PictureClosed, PictureLeaf, PictureMinus, PictureOpen, or PicturePlus. properties.
- To choose different types of outline display, use the OutlineStyle and Options properties.

Creating an outline structure

1. Place the following components on a form:
   - An Outline component
   - Two Edit components labeled 'Add Item' and 'Index'.
   - A SpeedButton component captioned 'Add'.
   - Two label components, Label3 captioned 'Current Index', and Label4 with no caption.
2. In the Lines property of Outline1, enter: TObject.
3. Write these event handlers for the appropriate components:

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
var
 AddStr: string;
 AddIndex:LongInt;
begin
AddStr := Edit1.Text;
AddIndex := StrToInt(Edit2.Text);
Outline1.AddChild(AddIndex, AddStr);
Label4.Caption := IntToStr(Outline1.SelectedItem);
end;

procedure TForm1.Outline1Expand(Sender: TObject; Index: Longint);
begin
Label4.Caption := IntToStr(Outline1.SelectedItem);
end;

procedure TForm1.Outline1Click(Sender: TObject);
begin
Label4.Caption := IntToStr(Outline1.SelectedItem);
end;
```

4. Set the form OnActivate event to point to the Outline1 OnClick event.

Running the example project

Now you can compile and run this application. The Outline component should display the string TObject, and this string should appear in the selected state.

With the program running...

1. In the Add Item edit control, enter: TComponent
2. In the Index edit control, enter: 1
3. Choose the Add button. Notice the change in the Outline component.
4. Double-click the TObject item in Outline1 to expand the outline. TComponent now displays one level below TObject.
5. Select TComponent in the Outline, repeat steps 1 & 2 entering 'TControl' and '2' in the edit controls, and choose the Add button again.
6. Double-click the TComponent item to expand the outline.
7. Select TControl and observe the change in the Current Index label.

See also

Common component tasks

About the Header component

See also THeader reference

Purpose

Use the Header component to create a component with resizable sections that display the text you specify. For instance, this component could be used in combination with either of the grid components to display column headings.

The Sections property of the header, which holds the text for the columnar headings, is a string list. You can perform many operations on string lists by using the methods and properties of the list. See Working with string lists.

Tasks

- To resize the header sections at design time, drag them by clicking the right mouse button.
- To prevent the use from resizing the sections at run time, set the AllowResize property to False.
- To resize the sections at run time, set the value of the SectionWidth property.

See also

Common component tasks
TDrawGrid component
TStringGrid component
TDBGrid component

Creating a tab list

See also

To create a tab list,

1. Place a TabControl or TabSet on the form.
2. Select its Tabs property, and open the String List editor.
3. Enter the text that you want the tab to display.
4. To create a new tab, press Enter.

Every line in the editor represents a tab in the TabControl.

Note: You also can add graphical objects to the string list. These graphics will appear on the tabs instead of text.

Importing a tab list from a file

1. Place a TabControl or TabSet on the form.
2. Select the Tabs property, and open the String List editor.
3. In the String List editor, choose Load to open the Load String List dialog box.
4. In the Load String List dialog box, choose the text file you wish to import.

Every line in the text file represents a tab.

See also

Manipulating Notebook pages.
Attaching tabs to a Notebook component
TNotebook component
TTabSet component
Working with string lists

Displaying a graphic on a tab at run time

See also

1. Place a TabControl or TabSet on the form.
2. Set its Style property to tsOwnerDraw.
3. Write an event handler for the OnDrawTab event to draw the graphic on the tab.

See also
Creating an owner-draw control

Attaching tabs to a Notebook component

See also

1. Add a Notebook and TabSet component to your form.
2. Create the pages for your notebook.

   This string list overrides any associated TabSet component's string list (accessed via the Tabs property).
3. Generate the following event handler for the form's OnCreate event to associate the pages to the tabs and the tabs to pages.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
TabSet1.Tabs := Notebook1.Pages;
Notebook1.PageIndex := Tabset1.TabIndex;
end;
```

Note: The tabbing behavior that occurs when tabs are clicked, automatically changes focus and displays the selected tab.

4. To change the current notebook page in response to the user clicking a tab, generate the following event handler for the TabSet's OnClick event:

```
Notebook1.PageIndex := Tabset1.TabIndex;
```

See also

About the Notebook component
TNotebook component
About the TabSet component
TTabSet component
Working with string lists

Specifying different glyphs for the different button states

See also

1. Create a bitmap that contains multiple images, all of the same size and in a horizontal row.

   The bitmap can contain up to four images, one for each possible button state. Each image in the bitmap should be the size that you want the glyph in the button to be.

2. Delphi determines how many glyphs your graphic contains, and sets the NumGlyphs property accordingly. It determines this by dividing the width of the graphic by its height. If the result is greater than four (4), or the height is greater than the width, Delphi treats the graphic as a single glyph.

Note: If the width of the graphic is not evenly divisible by its height, Delphi treats the graphic as a single glyph.

Once you load the bitmap onto your button, Delphi uses each glyph to represent a corresponding button state, based on the order in which they appear in the graphic. The following table specifies the glyph position in a bitmap and the button state Delphi assigns it.

Position	Button state	Meaning
First	Up	Button as it appears in regular up position
Second	Disabled	Button as you want it to appear when disabled
Third	Down	Button as it appears in the down position when it will return to an up position after being clicked
Fourth	Stay down	Button as it appears when it remains down after being clicked

See also

Loading an image
TImage component
TBitBtn component
AutoSizeProperty

Manipulating notebook pages

See also

By default a Notebook or TabbedNotebook component contains only one page. You can use the Pages property to add new pages. Once you add pages, you can use the ActivePage property to view the additional pages.

Adding pages to a notebook or tabbed notebook,

1. Select the Pages property.
2. Open the Notebook editor.
3. Click the Add button to open the Add Page dialog box.
4. Enter a name for this page, and an optional Help context.
5. Click OK.

To rename a page,

1. Select select the page in the Notebook editor.
2. Choose Edit to open the Edit Page dialog box.
3. Specify a new name or Help context for the page.
4. Choose OK.

To delete a page,

- Select select the page in the Notebook editor and choose the Delete button.

To rearrange the order of pages in a notebook,

- Select select the page in the Notebook editor and choose the Move Up or Move Down buttons.

See also

Working with string lists

About the BitBtn component

See also TBitBtn reference

Purpose

The BitMapButton component provides a means to customize a command button. Unlike the Button component, you can display a glyph on a bitmap button. You can also use the Kind property of the BitmapButton component to quickly create any of several predesigned command buttons. When you do so, Delphi sets properties for you and adds caption text and accompanying graphics to the button.

Tasks

Loading an image at design time

Specifying different glyphs for the different button states

- To change the button's border to match the operating system (Windows 3.x or later), use the Style property.
- To specify the amount of space between the bitmap button's glyph and its caption (if one exists) use the Spacing property.
- To specify where the glyph appears in relation to the caption, use the Layout property.
- To specify the amount of spacing between the glyph or caption and the button's border, use the Margin property.

Note: You can quickly create many standard command buttons by adding a BitBtn component to the form and setting its Kind property. The following tasks apply to standard buttons, and therefore to bitmap buttons whose Kind property is set to Custom.

Tasks for controlling focus in a form

Setting the tab order

Setting the focus in a dialog box

Enabling and disabling components

Tasks for creating command buttons in dialog boxes

Most dialog boxes contain buttons to provide the user with the following options:

Executing button code on Esc

Executing button code on Enter

Tasks for handling keyboard events

Responding to the OnKeyPress event

Responding to the OnKeyUp Event

Responding to the OnKeyDown Events

Redirecting keyboard events to the form

See also

Common component tasks

About the SpeedButton component

See also TSpeedButton reference

Purpose

The SpeedButton component provides a means to group several buttons so that their shift state at run time is interrelated. For this reason, speed buttons are often used to create tool bars. Compare to Radio buttons and Check boxes. Speed buttons can display a glyph, similar to the BitmapButton component, but unlike the bitmap button they cannot display a caption. Speed button glyphs must be Windows bitmap (.BMP) files.

Programming tasks

Adding a SpeedButton to a tool bar

Assigning a glyph to a speed button

Creating a group of speed buttons

Grouping speed buttons within container components

Setting the initial condition of a speed button

Specifying different glyphs depending on the button's state

Tasks for grouping speed buttons

- To enable several speed buttons to act as a group at run time, set their GroupIndex property to the same value (other than 0).
- To specify that a speed button in a group appears pressed when the form first appears at run time, set the button's Down property to True.
- To enable all speed buttons in a group to be in their up position, without requiring that one speed button remain pressed, set the AllowAllUp property to True for any speed button in the group. (When you set the AllowAllUp property for a speed button in a group, Delphi automatically sets the equivalent value for the remaining speed buttons in the group.)

Tasks pertaining to the speed button's appearance

- To specify the alignment of the glyph in relation to the button's border, use the Layout property.
- To specify the amount of spacing between the glyph and the button's border, use the Margin property.
- To specify the amount of space between the bitmap button's glyph and its caption (if one exists) use the Spacing property.

Note: The Layout, Margin, and Spacing properties might be affected by the type of glyph you have loaded.

See also

Common component tasks

Adding a tool bar to a form

About the MaskEdit component

See also TMaskEdit reference

Purpose

Use the MaskEdit component to provide an edit comonent that can be customized to require the user to enter only valid characters.

To read and write multiple lines of text, use the Memo component. To display text that the user cannot modify, use the Label component (or set the ReadOnly property of the MaskEdit component to True).

Tasks

Masking password characters

- To specify the edit mask, use the EditMask property. (See Input Mask editor.)
- To change the font of the text, use the Font property.
- To work with selected text at run time, use the SelLength, SelStart, and SelText properties, and the SelectAll method.
- To limit the line length, set the MaxLength property.
- To clear text from the edit mask, use the Clear method; or use the ClearSelection method to clear only selected text.
- To automatically select the text when the mask edit receives focus, set AutoSelect to True (default).
- To use the Clipboard object with text, use the CopyToClipboard, CutToClipboard, and PasteFromClipboard methods.
- To determine whether the contents of the mask edit have changed, use the Modified property.
- To prevent the height of the mask edit from changing dynamically to accommodate font changes, set the AutoSize property to False. (True is the default.)

  Note: Only the height, not the length (Width), of the mask edit is affected by the AutoSize property.
- To specify a different case for the text in the mask edit (all upper, all lower, or mixed case), use the CharCase property.

See also

Common component tasks

Edit tasks

About the StringGrid component

See also TStringGrid reference

Many properties, methods and events of the StringGrid are shared with the DrawGrid component. For more information, see About the DrawGrid component.

Purpose

Use the StringGrid component to display strings in column and row format.

Tasks

- To access the strings within the grid, use the Cells property.
- To access the objects within the grid, use the Objects property.
- To access the strings and objects within a particular row or column, use the Rows or Columns property.

See also

Common component tasks
TDrawGrid component

About the DrawGrid component

See also TDrawGrid reference

Purpose

Use the DrawGrid component to display data in column and row format.

Tasks for drawing in the grid

- Use the OnDrawCell event to specify the data you want drawn into the cell. The DefaultDrawing property (default = True) provides drawing behavior in the cell background, such as a focus rectangle and selection highlighting. To disable such behavior and start with a completely blank cell, set DefaultDrawing to False.
- To obtain the drawing area of a cell, use the CellRect method.
- To determine the column and row coordinates of the mouse cursor, use the MouseToCell method.
- To determine the column and row coordinates of the selected cell or cells, use the Selection property. You might use this in conjunction with the OnSelectCell event.
- Enable users to move columns and rows by handling the OnRowMoved and OnColumnMoved events.
- To enable the user to edit text in the grid, and to display the result, handle the OnGetEditText and OnSetEditText events.

Tasks for controlling the appearance of the grid

- Use the set members of the Options property to specify various features for the way the grid appears and responds to user interaction. The options include row and column sizing and delineation, cell selection, and tabbing and scrolling behavior. You can also set the width of grid lines with the GridLineWidth property.

Several properties affect the column width and row height:

- Use the DefaultColWidth and DefaultRowHeight, properties to set default a size for all columns or rows in the grid.
- Use the ColWidths and RowHeights properties to set the width or height at run time for a specific column or row.
- To specify nonscrolling columns or rows, use the FixedCols and FixedRows properties.
- Use the FixedColor property to specify a distinct color for nonscrolling rows and columns.
- Use the TopRow to specify which row at run time is the topmost row in the grid. (The row specified with TopRow appears immediately beneath any row specified with FixedRows.)
- Use the LeftCol property to specify which column at run time is the leftmost column in the grid. (The column specified with LeftCol appears immediately adjacent to any column specified with FixedCols.)
- Use the OnTopLeftChanged event to handle changes in either the TopRow or LeftCol properties.
- To determine how many rows and columns are visible in the grid at run time, use the VisibleRowCount and VisibleColCount properties. (Read-only.)

See also

TStringGrid component

About the StringGrid component

About the Image component

See also TImage reference

Purpose

Use the Image component to import a graphical image into your form. Delphi supports the following picture formats:

- Bitmaps (.BMP)
- Metafiles (.WMF)
- Icons (.ICO)

Tasks

Adding an Image component to the form
Changing the Z-Order of components
Loading an image
Providing an area for drawing at run time
Drawing on a bitmap
Printing graphics
Loading a picture from a file
Saving a picture to a file
Replacing a picture
About the Clipboard with graphics

- To automatically resize the graphic so that it fits the component, set the Image component's Stretch property to True. (Stretch has no effect on the size of icon (.ICO) files.)
- To automatically resize the component so that it fits the graphic, set the Image component's AutoSize property to True before you load the graphic.
- To center the graphic in the component, set the Image component's Center property to True. To align the graphic with the top-left corner of the component, set Center to False.

See also

Common component tasks
About the shape component
About the label component

About the Shape component

See also TShape reference

Use the shape component to display graphical shapes on your forms.

Tasks

Changing the Z-Order of components

Drawing Graphics At Run Time

About Brushes

About Pens

- To change the fill color of a shape component, use the Color option of the Brush property.
- To change the fill style (for example, solid or crosshatch) of a shape component, use the Style option of the Brush property.
- To change the color of the lines drawn in the component -- both the shape's border and fill lines -- use the Color and Mode options of the Pen property.
- To change the line style -- for example, solid or dotted -- and width of the component border, use the Style and Width options of the Pen property.
- To change the shape of the component, use the Shape property.

See also

Common component tasks
About the Image component
About the Label Component
Drawing vs. painting

About the Bevel component

See also TBevel reference

Purpose

Use the Bevel component to place a raised or lowered line on your form. You can use the Bevel component to place a bevelled line around, next to, above, or below related components.

Since the Bevel component does not have a Name property, you might want to use a label component as a caption.

Note: The Bevel component is not a container. If you want to group components, use the GroupBox component.

Tasks

- To choose a bevel shape, use the Shape property.
- To specify whether the bevel is raised or lowered, use the Style property.

See also

Common component tasks

TShape component

About the ScrollBox component

See also TScrollBox reference

Purpose

The ScrollBox component combines a container area, similar to a Panel component, with vertical and horizontal scroll bars. You can use a scroll box to define a scrollable subarea of the form.

Note: When you place components within container components, you create a new parent-child relationship between the container and the components it contains. The form remains the owner for all components, regardless of whether they are parented within another component.

Tasks

Adding a scrollable region

- To enable the vertical or horizontal scroll bars of the scroll box, specify values for the following two members of either the VertScrollBar or HorzScrollBar nested property set:
- The Range property. This value indicates the measurement below which you want the scroll bars to appear. For example, with a Range value of 100, the scroll bars will not appear unless the container area is less than 100 pixels in either height or width (depending on whether you are specifying the horizontal or vertical scroll bar).
- Visible must be True (default). In code, this setting would look like:

```
ScrollBox2.HorzScrollBar.Visible := True;
```

- To ensure that a particular component is in view when the form first appears, use the ScrollInView method.

See also

TScrollBar component
Common component tasks

About the DataSource component

See also TDataSource reference

Purpose

Use the DataSource component to connect dataset components such as TTable and TQuery with data-aware components such as TDBGrid and TDBEdit.

For basic background information on the DataSource component, see Overview of TDataSource component and Specifying a data source.

Connecting to databases

- To connect a dataset to a DataSource component, see Using TTable.
- To connect your application to a physical database, see Using TDatabase.

See also

Displaying data in a data control

Features chart

About the Table component

See also TTable reference

Purpose

Use the Table component to retrieve data from a physical database table and send data back to a physical database table.

Users view data retrieved by the Table component in a data-aware component. Use a DataSource to connect a Table and a data-aware component.

For basic background information on the Table component, see Using TTable and Using TQuery.

Connecting to databases

- To specify the database that contains the physical table with which you want to work, see Using TDatabase.
- To connect a Table component to a physical table, see Specifying a data source.
- To connect your application to a physical database, see Using TDatabase.

Accessing data

- To ensure no other user can access or modify the table while you have it open, set the Exclusive property to True.
- To prevent modifications to the table, set ReadOnly to True.

  You cannot change the ReadOnly property while the Active property is set to True.
- To create a persistent list of TField objects, see Using the Fields editor.
- To sort the table on a secondary index, use the IndexName property.

Creating master-detail relationships

- To create a master-detail relationship, specify the data source used by the master table in the MasterSource property of the detail.
- To specify the field upon which to link master and detail tables, use the MasterFields property to display the Field Link Designer.

See also

Displaying data in a data control.

Features chart

About the Query component

See also TQuery reference

Purpose

Use the Query component to provide SQL statements that retrieve data from a physical database table and send data back to a physical database table.

Users view data retrieved by the Query component in a data-aware component. Use the DataSource to connect a Query and a data-aware component.

For basic background information on the Query component, see Using TQuery.

Connecting to databases

- To specify the database that contains the physical table with which you want to work, see Using TDatabase.
- To connect a Query component to a physical table, see Specifying a data source.
- To connect your application to a physical database, see Using TDatabase.

Other tasks

- To create an SQL statement, select the SQL property and either double-click the values column or click the ellipses button. This opens the String List editor, where you enter the query.
- To create a persistent list of TField objects, see Using the Fields editor.
- To create a query that accesses tables in more than one database, see Creating heterogeneous queries.
- To create a live result set that can post any changes that occur, set the RequestLive property.
- To execute an SQL statement, see Using TQuery.

See also

Displaying data in a data control

Features chart

About the StoredProc component

See also TStoredProc reference

Purpose

Use the StoredProc component to access stored procedures on a database server and send data back to a physical database table.

Tasks

- To specify the database in which the stored procedure is defined, set the DatabaseName property.
- To specify the name of the stored procedure, set the StoredProcName property.
- To specify parameters for a stored procedure, double-click the Params property to open the Parameters editor.

  See Using TStoredProc
- To execute a stored procedure, see Using TstoredProc.
- To create a persistent list of TField objects, see Using the Fields editor.

See also

Features chart

Displaying data in a data control

About the Database component

See also TDatabase reference

Purpose

Use the Database component to connect to a database when you require any of the following features:

- Create a persistent database connection.
- Customize database server logins.
- Create BDE aliases local to an application.
- Control transactions and specify transaction isolation level.

Connect the Database component to a dataset component such as the Table or Query to access data.
For basic background information on the Database component, see Using TDatabase.

Connecting to databases

- To connect to a database server, see Using TDatabase.
- To edit the SQL Link parameters required to connect to a server, use the Params property to open the String List editor.
- To control connections to a database, see Using TDatabase.

Transaction processing

- To understand Delphi transaction processing, see Handling transactions.
- To explicitly control transactions, see Handling transactions.
- To specify how a transaction interacts with other transactions that access the same tables simultaneously, set the TransIsolation property.

Other tasks

- To allow for user name and password access or to bypass server security, see Logging into a server.
- To create an application-specific alias, assign a value to the DatabaseName property.

  To customize the parameters for this alias, double-click the TDatabase component .

See also

Displaying data in a data control

Features chart

About the BatchMove component

See also TBatchMove reference

Purpose

Use the BatchMove component to perform batch operations on groups or records. BatchMove lets you copy a table structure or its data. You can copy entire tables from one database format to another.

Tasks

- To copy a dataset to a table, see Using TBatchMove.
- To append a dataset to a table, see Appending a dataset states.
- To update a table with data from a dataset, see one of the following topics:
- Setting dataset states.

See also

Displaying data in a data control

Features chart

About the Report component

See also TReport reference

Purpose

Use the Report component to enable printing and viewing of database reports through ReportSmith.

Connecting the report

- To connect to an existing ReportSmith report, specify the following:
- The name in the ReportName property
- The directory in the ReportDir property.
- To preconnect the report to a database, so it does not prompt for login information, call the Connect method.

Running the report

- To load ReportSmith Runtime and print or display the specified report, call the Run method.
- To specify whether to print or display the report, set the Preview property.
- To specify whether to automatically unload the ReportSmith Runtime executable after a report runs, set the AutoUnload property.
- If an application runs only one report at a time, AutoUnload should be True.
- If an application runs a series of reports, AutoUnload should be False.
- To send a macro command to ReportSmith, call the RunMacro method.
- To rerun a report when report variables have changed, call the ReCalcReport method

Using variables

- To specify variables to use with the report, use the InitialValues property to open the String List editor.
- To change a specific report variable, call the SetVariable method.

See also

Displaying data in a data control

Features chart

About the DBGrid component

See also TDBGrid reference

Purpose

Use the DBGrid component to display data from a dataset component in a spreadsheet-like grid. You can specify a static set of columns for the grid in the Fields editor at design time or a dynamic set of columns at run-time.

Controlling grid appearance

- To specify the appearance of fields in the grid during display and editing, set the DisplayFormat and EditFormat properties, respectively.
- To prevent a TField component from being displayed in a grid, set the Visible property.
- To permit users to rearrange the order of columns in the grid, set the DragMode property.

Controlling editing in the grid

- To ensure that a value is entered for a field, set the Required propety.

See also

Common data control component tasks

Data controls summary

About the DBNavigator component

See also TDBNavigator reference

Purpose

Use the DBNavigator component to enable users to navigate through records and to perform operations such as inserting records and posting changes.

Tasks

- To specify which buttons are visible on the navigator, set the VisibleButtons property.
- To display help hints when the mouse cursor passes over the navigator buttons, set the ShowHint property.

See also

Common data control component tasks

Data controls summary

About the DBText component

See also TDBText reference

Purpose

Use the DBText component to display a field value in the current record as a read-only label on a form, similar to the TLabel component.

Users cannot modify the data displayed in a DBText. To enable users to modify the contents of the field, use the DBEdit component.

Tasks

- To ensure that the DBText control resizes itself to display data of varying widths, set the AutoSize property of TDBText to True (the default).
- To wrap text displayed in the Label, set WordWrap to True.

  Setting WordWrap to True causes text entered after a space to wrap to the next line, unless there is still room in the current line.
- To specify whether you want the Label text to align left, right, or centered, use the Alignment property.
- To prevent the Label from obscuring other components, set its Transparent property to True.

See also

Common data control component tasks

Data controls summary

About the DBEdit component

See also TDBEdit reference

Purpose

Use the DBEdit component to read or write a single line of data from a specific column in the current record of a dataset.

- To read and write multiple lines of text, use the DBMemo component.
- To display text that the user cannot modify, use the DBText component (or set the ReadOnly property of the DBEdit component to True).

TDBEdit is a data-aware version of the TEdit component.

For basic background information on the DBEdit component, see Displaying data in a data control.

Tasks

- To specify the font of the text, use the Font property.
- To specify the line length, set the MaxLength property.
- To automatically select the text when the Edit box receives focus, set AutoSelect to True (default).
- To prevent the height of the edit box from changing dynamically to accommodate font changes, set the AutoSize property to False. (True is the default.)

  Only the height, not the length (Width), of the edit box is affected by the AutoSize property.
- To specify a case for the text in the edit box (all upper, all lower, or mixed case), use the CharCase property.

See also

Common data control component tasks

Data controls summary

About the DBMemo component

See also TDBMemo reference

Purpose

Use the DBMemo component to enable users to view and modify a multi-line field in a dataset or formatted text in BLOB format.

- Use the DBEdit component to display or edit a single line of text.
- Use the DBImage component to display images in BLOB format.

DBMemo is a data-aware version of the TMemo component.

Appearance of memos and data

- To provide scroll bars in the memo, set the ScrollBars property.
- To prevent word wrap, set the WordWrap property to False.
- To specify the alignment of text within the control, set the Alignment property.
- To suppress the display of the data in the field until a user double-clicks it, set the AutoDisplay property.

  Because TDBMemo can display large amounts of data, it can take time to paint the display. Setting AutoDisplay to False can improve performance.
- To change the font of the text, use the Font property.

Editing

- To prohibit editing, set the ReadOnly property of TDBMemo to True. By default, DBMemo permits editing.
- To permit users to enter tabs in a memo, set the WantTabs property to True.
- To cut, copy, and paste text programatically within a memo control, use the CutToClipboard, CopyToClipboard, and PasteFromClipboard methods.
- To specify the number of characters users can enter into the database memo, use the MaxLength property.

See also

Common data control component tasks

Data controls summary

About the DBImage component

See also TDBImage reference

Purpose

Use the DBImage component to display a graphic image or BLOB data in image format in a field of a dataset.

DBImage cannot display formatted text in BLOB format. Use the DBMemo component for this purpose.

Tasks

- To edit a BLOB graphic, cut or copy it to the clipboard, modify it in your editor, and paste it back into Delphi. To perform these tasks programatically, use the CutToClipboard, CopyToClipboard, and PasteFromClipboard methods.
- To suppress the display of the data in the field until a user double-clicks it, set the AutoDisplay property.

Note: Because DBImage can display large amounts of data, it can take time to paint the display. Setting AutoDisplay to False can improve performance.

See also

Common data control component tasks

Data controls summary

About the DBListBox component

See also TDBListBox reference

Purpose

Use the DBListBox component to display a list of items from which users can update a field in the current record of a dataset.

The Items property of the DBListBox component, which specifies the items contained in the list, is a string list. You can perform many operations on string lists by using the methods and properties of the list. See Working with string lists.

DBListBox is a data-aware version of the TListBox component.

Controlling the display

- To specify the height of the list box, set the IntegralHeight property.
- If IntegralHeight is True, the default, the position of the bottom of the list box changes to ensure that the bottom item is always fully visible.
- If IntegralHeight is False, the bottom of the list box is determined by the ItemHeight property, and the bottom item might not be completely displayed.
- To specify the height of each item, set the ItemHeight property. You must also set the style to csOwnerDrawFixed.

See also

Common data control component tasks

Data controls summary

About the DBComboBox component

See also TDBComboBox reference

Purpose

Use the DBComboBox component to combine the functionality of a DBList box and a DBEdit. Users can update a field in the current record of a dataset by typing a value or choosing a value from the drop-down list.

The Items property of the DBComboBox component, which specifies the items contained in the list, is a string list. You can perform many operations on string lists by using the methods and properties of the list. See Working with string lists.

DBComboBox is a data-aware version of the TComboBox component.

Controlling the display

- To specify the maximum number of items displayed in the list, set the DropDownCount property.
- To control the display style of the Items list, specify the Style property.

  For example, Style lets you specify csDropDownList for a drop-down list or csSimple for a simple scrolling list.
- To specify the height of each item, set the ItemHeight property. You must also set the style to csOwnerDrawFixed.
- To display the item list in alphabetical order, set Sorted to True.

See also

Common data control component tasks

Data controls summary

About the DBCheckBox component

See also TDBCheckBox reference

Purpose

Use the DBCheckBox component to provide a check box that can read or edit the contents of a field in a dataset.

The DBCheckbox controls its own Checked or Unchecked state by comparing the contents of the field to the contents of the ValueChecked and ValueUnchecked properties. (Both values cannot match at the same time.)

- If the value of the ValueChecked property matches the value of the field, Delphi checks the check box.
- If the value of the ValueUnchecked property matches the value of the field, Delphi unchecks the check box.

DBCheckBox is a data-aware version of the TCheckBox component.

Setting values

- To specify the value DBCheckBox posts to the database when it is unchecked, set the ValueUnchecked property.

  By default, this value is set to False, but you can specify any alphanumeric value.
- To specify the value DBCheckBox posts to the database when it is checked, set the ValueChecked property.

  By default, this value is set to True, but you can specify any alphanumeric value.

Controlling the appearance

- To group check boxes in the form, place them all inside a single Panel, GroupBox, or ScrollBox component.

  You must first place the container component on the form, then place the check box components inside it.
- To display a label for the check box, set the Caption property.
- To place the check box to the right or left of the text in its Caption, set its Alignment property.
- To make a check box unavailable to the user, set its Enabled property to False.
- To enable the user to gray the check box, set its AllowGrayed property to True.

See also

Common data control component tasks

Data controls summary

About the DBRadioGroup component

See also TDBRadioGroup reference

Purpose

Use DBRadioGroup to set the value of a field in a dataset when there is a limited number of possible values for the field.

The radio group contains one button for each value a field can accept. Delphi automatically selects a radio button whose text string matches the current field in the database.

DBRadioGroup is a data-aware version of the TRadioGroup component.

Tasks

- To specify a set of radio buttons and labels to appear next to each button, set the Items property.
- Delphi creates a radio button and a label for each string specified in the Items list.
- When the current record contains a value that matches one of the strings in the Items property, that radio button is automatically selected.
- To specify an optional list of strings to post when a user selects a radio button, set the Values property.

Strings are associated with buttons in numeric sequence; that is, the first string is associated with the first button , the second string with the second button, and so on. For example, if the strings, "Magenta", "Yellow", and "Cyan", are listed for Values, and the user selects the first button (labeled "Red"), then Delphi posts "Magenta" to the database.

See also

Common data control component tasks

Data controls summary

About the DBLookupList component

See also TDBLookupList reference

Purpose

Use the DBLookupList component to display a list of items from which users can update a field in the current record of a dataset.

Unlike DBListBox, Delphi populates the DBLookupList list values dynamically from a second dataset, known as the lookup dataset, at run time.

The DBLookupList component is similar to DBLookupCombo, but it displays a scrollable list of available choices instead of a drop-down list.

Connecting the DBLookupList

- To specify a data source for the lookup dataset, set the LookupSource property.

  The LookupSource must be a different data source than the DataSource. If you want to display values from a column in the same table as the first dataset, place a second data source and dataset component on the form and point them at the same data as the first data source and dataset.
- To specify a field in the LookupSource dataset that Delphi links to the primary dataset, set the LookupField property.

  The LookupField property column must contain the same values as the DataField property column, although the column names can differ.

Controlling the display

- To specify the columns that DBLookupList displays, set the LookupDisplay property.

  If you do not set LookupDisplay, DBLookupList displays the values found in the LookupField column. Use LookupDisplay to display a column other than the LookupField column, or to display multiple columns in the drop-down list. Use semicolons to separate multiple column names.
- To specify the appearance of multiple columns, set the Options property.
- loColLines, if True, separates the columns in the lookup list with lines.
- loRowLines, if True, separates the rows in the lookup list with lines.
- loTitles, if True, displays column names as titles above the columns in the lookup list.

See also

Common data control component tasks

Data controls summary

About the DBLookupCombo component

See also TDBLookupCombo reference

Purpose

Use the DBLookupCombo component to combine the functionality of a DBList box, DBEdit, and DBLookupList. Users can update a field in the current record of a dataset by choosing a value from the drop-down list.

Delphi populates the DBLookupCombo list values dynamically from a second dataset, known as the lookup dataset, at run time.

Connecting the DBLookupCombo

- To specify a data source for the lookup dataset, set the LookupSource property.

  The LookupSource must be a different data source than the DataSource. If you want to display values from a column in the same table as the first dataset, place a second data source and dataset component on the form and point them at the same data as the first data source and dataset.
- To specify a field in the LookupSource dataset that Delphi links to the primary dataset, set the LookupField property.

  The LookupField property column must contain the same values as the DataField property column, although the column names can differ.

Controlling the display

- To specify the columns that DBLookupCombo displays, set the LookupDisplay property.

  If you do not set LookupDisplay, TDBLookupCombo displays the values found in the LookupField column. Use LookupDisplay to display a column other than the LookupField column, or to display multiple columns in the drop-down list. Use semicolons to separate multiple column names.
- To specify the appearance of multiple columns, set the Options property.
- loColLines, if True, separates the columns in the lookup list with lines.
- loRowLines, if True, separates the rows in the lookup list with lines.
- loTitles, if True, displays column names as titles above the columns in the lookup list.

See also

Common data control component tasks

Data controls summary

Common data control component tasks

See also

Data control components are data-aware components that you connect to a dataset through a DataSource component.

Some of the tasks that you perform with data controls are related to all the data-aware components, not to a specific component. The list below represents a sampling of such procedures. Choose a topic for more information.

To learn background information about data controls, see Data controls summary.

Tasks

- To link a data control to a dataset, see Data controls summary.
- To specify the field that you want to display in a data control, use the DataField property.

  You cannot connect the DBGrid or DBNavigator components to an individual field.
- To disable, enable, and refresh data controls, see Displaying data in a data control.
- To specify update privileges in the data controls, see Displaying data in a data control.

See also

Data controls summary

About the OpenDialog component

See also TOpenDialog reference Example

Purpose

Use the OpenDialog component to create a common Windows Open dialog box that enables users to specify the name of a file to open.

When a user enters a file name in an Open dialog box, Delphi assigns the file name as the value of the OpenDialog FileName property. Use the value of the FileName property to specify the file you want to open.

Tasks

Setting file filters

Enabling Help in a common dialog box

Opening a dialog box with a menu command

- To specify a default file extension, use the DefaultExt property.
- To specify a filter or mask to display a restricted set of files, use the Filter property.
- To set the current directory, use the InitialDir property.
- To change the title of the Open dialog box, use the Title property.
- To let the user select multiple file names, use the Options property to set ofAllowMultiSelect to True.

See also

About the dialog components

Example

The following example shows you how to use the OpenDialog component to open a file that a user specifies in an Open dialog box.

In this example, OpenTableDialog is an OpenDialog component, and Table1 is a Table component. When a user specifies a file in the Open dialog box, Delphi sets the FileName property of OpenTableDialog. When a user accepts the dialog box, the Execute method returns True, and the code in the **if...then** loop is executed.

The event handler first sets the Active and Connected properties of Table1 to False, so the TableName property can be reassigned. Then the TableName property of Table1 is set to the value of the OpenTableDialog component's FileName property. When the Active and Connected properties of Table1 are set to True, the file the user specified is opened.

```
procedure TForm1.OpenTableButtonClick(Sender: TObject);
begin
     if OpenTableDialog.Execute then
     begin
          Table1.Active := False;
          Table1.Connected := False;
          Table1.TableName := OpenTableDialog.Filename;
          Table1.Active := True;
          Table1.Connected := True;
     end;
end;
```

About the SaveDialog component

See also TSaveDialog referencet

Purpose

Use the SaveDialog component to create a common Windows Save dialog box that enables users to specify the name of a file to save.

When a user enters a file name in a Save dialog box, Delphi assigns the file name as the value of the SaveDialog FileName property. Use the value of the FileName property to specify the file you want to open.

Tasks

Setting file filters

Enabling Help in a common dialog box

Opening a dialog box with a menu command

- To specify a default file extension, use the DefaultExt property.
- To specify a filter or mask to display a restricted set of files, use the Filter property.
- To set the current directory, use the InitialDir property.
- To change the title of the Save dialog box, use the Title property.
- Use the Options property to set several options such as,
- Enabling the user select multiple file names, (ofAllowMultiSelect)
- Displaying or hiding the Read-only checkbox (ofReadOnly, ofHideReadOnly)
- Prompting the user if they attempt to save using a filename that exists (ofOverwritePrompt)
- Prompting the user if they attempt to save a file that is read-only (ofNoReadOnlyReturn)

and so on.

See also

About the dialog components

OpenDialog component tasks

Using the FontDialog component

See also TFontDialog reference Example

Purpose

Use the FontDialog component to create a common Windows Font dialog box that enables you to set the font at run time for any component that contains a Font property.

Changing displayed fonts

You can change the font displayed by components that contain a Font property by assigning the font the user chooses in the Font dialog box to the component's Font. For example,

```
FontDialog1.Execute;
Memo1.Font.Assign(FontDialog1.Font);
```

Changing printed fonts

The Delphi printer object contains a Fonts property that specifies the list of fonts supported by the current printer. These fonts appear in the Font list of the Font dialog box. Unless your program specifies otherwise, the printer uses the default (System) font that is returned by the Windows device driver to print text from your application.

To change the printer's font, simply assign the Font property for the memo that contains the text you want to print, to the Font property for the printer object's Canvas. This downloads the selected font to the printer.

For example:

```
Printer.Canvas.Font := Memo1.Font;
```

Other tasks

Enabling Help in a common dialog box

Opening a dialog box with a menu command

See also

About the dialog components

Example

The following example ensures that the value of the memo component Font property is reflected in the Font dialog box, and that settings made from within the dialog box are reflected back into the memo.

```
procedure TEditForm.SetFont(Sender: TObject);
begin
  FontDialog1.Font := Memo1.Font;
  if FontDialog1.Execute then
Memo1.Font := FontDialog1.Font;
end;
```

Note: You generally don't want the dialog box to display anything other than default settings unless the user makes modifications from within the dialog box at run time. Any settings you make at design time to the Font dialog box are, as they should be, overridden by any event handlers that you write to handle font changes at run time. To change the font displayed when the user first opens the Font dialog box in an application, change the Font property for the memo component -- not for the Font dialog box.

About the ColorDialog component

See also TColorDialog reference Example

Purpose

Use the ColorDialog component to provide a Color dialog box that enables you to set the color at run time for any component that contains a Color property.

Tasks

Enabling Help in a common dialog box

Opening a dialog box with a menu command

See also

About the dialog components

Example

The following code uses a Button click to execute the Color dialog box, and change the color of the form to the color the user selects in the Color dialog box when the user chooses OK from inside the dialog box.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  If ColorDialog1.Execute then
  Form1.Color := ColorDialog1.Color;
end;
```

About the PrintDialog component

See also TPrintDialog reference

Purpose

Use the PrintDialog component to create a common Windows printer dialog box that the user can interact with to perform printing tasks. When the user chooses the Setup button from within the printer dialog box, the Windows Printer Setup dialog box displays.

Tasks

Printing the contents of a memo

About the printer object

Enabling Help in a common dialog box

Opening a dialog box with a menu command

PrintDialog component properties

The properties available in the Object Inspector for the PrintDialog component correspond to several areas of the Print dialog box in the running application. Although Delphi makes it easy to do so, you do not usually need to reset the default options for this dialog box at design time. The topics below summarize several of the more commonly used properties.

MinPage

MaxPage

Options | poPageNums

Options | poPrintToFile

Options | poSelection

PrintRange

See also

About the dialog components

About the PrinterSetupDialog component

See also TPrinterSetupDialog reference

Purpose

You do not have to have a PrinterSetupDialog component in your form in order to display the common Windows Printer Setup dialog box in your application at run time, if your application uses the PrintDialog component. When the user chooses the Setup button from within the Printer Dialog box, the Printer Setup dialog box automatically appears.

The settings in this dialog box, and in the Options and Advanced Options dialog boxes, write to and read from settings in the user's WIN.INI file. Therefore you do not write much code that interacts directly with this component, other than launching it. All the functions it provides after that are handled by Windows.

Tasks

Enabling Help in a common dialog box

Opening a dialog box with a menu Command

See also

About the dialog components

About the FindDialog component

See also TFindDialog reference

Purpose

Use the FindDialog component to create a common Windows Find dialog box.

Tasks

Finding a text string using the FindDialog

Enabling Help in a common dialog box

Opening a dialog box with a menu command

- To specify a value for the text string to search for, use the FindText property.
- To specify a value to replace the text string searched for, use the ReplaceText property.
- To close the dialog, use the CloseDialog method.

Finding a text string using the FindDialog component

Here is an exercise you can try to see how to work with the FindDialog component.

1. On a form, place an Edit component, a SpeedButton component, and a FindDialog component. Write some text in the Text property of the Edit component.
2. Write the following code for the SpeedButton OnClick event:

```
FindDialog1.Execute;
```

3. The following event handler code for the FindDialog OnFind event searches the text of the Edit control for a string provided by the user in the Find dialog box. If the string is found, the matching text is selected in the Edit1 component. Otherwise, a 'not found' message displays.

```
procedure TForm1.FindDialog1Find(Sender: TObject);
var
ToFind: string;          {String to find}
FindIn: string;          {String to search}
Found: integer;          {String found indicator}
Index: integer;          {Start pos. of found text}
FoundLen: integer;       {Length of found text}
begin
ToFind := FindDialog1.FindText;     {User specified string in dialog}
FoundLen := Length(FindDialog1.FindText); {Length of specified text}
FindIn := Form1.Edit1.Text;      {Text to search for string}
Found := Pos(ToFind, FindIn);    {Search for string}
If Found > 0 then
  begin
  Form1.BringToFront;             {Edit control must have focus in
  Form1.ActiveControl := Edit1;    order to display selected text}
  Edit1.SelStart:= Found -1;      {Select the string found by POS}
  Edit1.SelLength := FoundLen;
  end;
Else
  begin
  MessageDlg('Text not found', mtInformation, [mbOK], 0);
  end;
end;
```

Note: If you also use a ReplaceDialog box in your form, you can move the **var** declaration in the above event handler to the beginning of the Implementation section of the form unit. See Replacing a text string using the ReplaceDialog.

See also

TReplaceDialog component

About the dialog components

About the ReplaceDialog component

See also TReplaceDialog reference

Purpose

Use the ReplaceDialog component to create a common Windows Replace dialog box.

Tasks

Replacing a text string using the ReplaceDialog

Enabling Help in a common dialog box

Opening a dialog box with a menu command

- To specify a value for the text string being searched for, use the FindText property.
- To specify a value to replace the text string searched for, use the ReplaceText property.
- To close the dialog, use the CloseDialog method.

Replacing a text string using the ReplaceDialog component

Here is an exercise you can try to see how to work with the ReplaceDialog component.

1. On a form, place an Edit component, a SpeedButton component, and a ReplaceDialog component. Write some text in the Text property of the Edit component.
2. Write the following code for the SpeedButton OnClick event:

```
ReplaceDialog1.Execute;
```

3. Declare the following variables in the Implementation section of the Form1 unit:

```
var
ToFind: string;         {String to find}
FindIn: string;         {String to search}
Found: integer;         {String found indicator}
Index: integer;         {Start pos. of found text}
FoundLen: integer;      {Length of found text}
FoundText: string;      {String to be replaced}
DidFind: boolean;       {Flags if Find runs before Replace}
```

4. Use the following event handler code for the ReplaceDialog OnFind event. This code searches the text of the Edit control for a string provided by the user in the Replace dialog box. If the string is found, the matching text is selected in the Edit1 component. Otherwise, a 'not found' message displays.

```
procedure TForm1.ReplaceDialog1Find(Sender: TObject);
begin
ToFind := ReplaceDialog1.FindText;     {String from Find control}
FoundLen := Length(ReplaceDialog1.FindText); {Length of text}
FindIn := Form1.Edit1.Text;               {Text to search for String}
Found := Pos(ToFind, FindIn);             {Search for the String}
If Found > 0 then
  begin
  Form1.BringToFront;             {Edit control must have focus in
  Form1.ActiveControl := Edit1;    order to display selected text}
  Edit1.SelStart:= Found -1;      {Select the string found by POS}
  Edit1.SelLength := FoundLen;
  end;
Else
  begin
  MessageDlg('Text not found', mtInformation, [mbOK], 0);
  end;
end;
```

5. Use the following code for the ReplaceDialog OnReplace event. This code checks to see that a search was done for a string to be replaced. If not, it calls the OnFind event handler before replacing text selected in the Edit control with the string specified in the Replace control of the Replace dialog box.

```
procedure TForm1.ReplaceDialog1Replace(Sender: TObject);
begin
If DidFind = False then       {If no search for string took place}
  begin
  ReplaceDialog1Find(Sender); {Search for string, replace if found}
  Edit1.SelText := ReplaceDialog1.ReplaceText;
  end
Else                          {If search ran, replace string}
  begin
  Edit1.SelText := ReplaceDialog1.ReplaceText;
  end;
end;
```

See also

About the dialog components

TFindDialog component

Enabling Help in a common dialog box

See also

1. Set Help to True. This displays the Help button on the dialog box

   For each common dialog box component, the Help property is prefaced with initials related to the component. For example, the Help property for the Font dialog box component is called fdHelp, for Font Dialog.
2. Specify a HelpContext for the dialog box component by entering a long integer.

   This long integer must correspond to a topic in the Help file.
3. On the Application page of the Project Options dialog box, specify the filename and location of the Help file.

See also

Dialogs page components
Non-Delphi resource files
Specifying the project Help and icon files

About the component library

See Also

The components displayed in the Component palette reflect the contents of a library file that Delphi maintains. You can customize this library (and therefore what is displayed in the palette) by adding or removing components. Whenever you customize the library, Delphi rebuilds the library file.

You can develop as many such customized library files as you choose, by saving each library configuration under a different name. You can then choose to use a particular library file when building applications so that your library provides you with exactly the components you prefer.

When modifying libraries, keep the following points in mind:

- You can work on a project in Delphi only if the current component library contains all the components used in the project.

If you try to load a project that uses components not currently installed, you will get a message from Delphi indicating which type it tried to load but could not.

- Delphi always saves the last version of the component library, so if you remove components that you later want to use,

you can easily revert to the previous version. The previous version of the library file is stored with the same name, and the extension .~DC.

The following topics describe how to customize the component library:

- Adding components to the component library
- Removing components from the component library
- Saving a customized library
- Loading a component library

See also

Creating components.
Installing Visual Basic components
Handling an unsuccessful compilation

Adding components to the library

See Also

1. Choose Component|Install to open the Install Components dialog box
2. If necessary, specify a path to the component unit, or choose Add to open the Add Module dialog box. The directory in which the unit resides must be on the search path specified in this dialog box.
3. In the Add Module dialog box, enter the name of the unit you want to add, or choose browse to specify a search path.

   You cannot specify a literal path in the Add Module dialog box. The path you choose in the Add Module dialog box adds the directory path to the search path listed in the Install Components dialog box.
4. Click OK to close the Add Module dialog box.

   The name(s) of the component unit(s) you have specified appear at the bottom of the Installed Units list. If you select the unit name, the class names for units already residing in the library are displayed in the Component classes list. Class names for newly added units are not shown.
5. Choose OK to close the Install Components dialog box and rebuild the library.

   Note that even if you have made no changes to the unit list, Delphi rebuilds the library. If you have unsaved forms open, you are prompted to save them. Once the library is rebuilt, the components you have installed are reflected in the Component palette.

Note: Newly installed components initially appear on the page of the Component palette that was specified by the component writer in the component source code. You can move the components to a different page after they have been installed on the palette.

Removing components from the component library

1. Choose Component|Install to open the Install Components dialog box.
2. In the Installed Units list box, select the unit you want to remove.
3. Choose Remove to remove the items from the list box.
4. Choose OK to close the Install Components dialog box.

Delphi rebuilds the library, and the deleted components no longer appear in the palette.

See also

Creating components
Customizing the component library
Installing Visual Basic components
Saving a customized library
Loading a component library

Saving a customized library

See Also

1. Specify the full path and new file name in the Library Filename box of the Install Components dialog box.
2. Recompile the component library

You can install this customized library whenever you want, or specify it as the default component library.

Loading a component library

1. Choose Component|Open Library to open the Open Library dialog box.
2. Select the library file you want to install.
3. Click OK.

Delphi replaces the current component library with the library file you have chosen and reflects the contents of your library file in the Component palette.

See also

Adding components to the component library
Removing components from the component library
Customizing the component library
Installing Visual Basic components

Installing OCX controls

See Also

1. Open the Install Components dialog box.
2. Choose OCX to open the Install OCX File dialog box.
3. Navigate until you locate the .OCX file you want to add, then choose OK.

   The Install OCX dialog box appears, with the name of the specified .OCX file displayed in a read-only edit box.
4. Use the options in the Install OCX dialog box to specify

- The directory path where the .OCX file is located (use the Browse button if you are not sure of the full directory path for the file)
- The page on the palette where you want the OCX control to be installed
- A new name for the component class, if necessary

  Change the name, for example, if you are installing a OCX control that has the same class name as an existing component in your project.

5. Once you have specified the options you want, choose OK to return to the Install OCX File dialog box.
6. Repeat steps 3 through 5 until you have specified all the OCX controls you want to add.

   They should appear in the Installed Units list.
7. Choose OK to close the Install OCX File dialog box and rebuild the component library.

Note: In many cases, OCX controls are written so as to require placement in the system path, or in the \WINDOWS\SYSTEM directory. If you see a message stating "Can't load OCX" when you compile or run your application, verify that any OCX controls are in the proper location.

See also

Adding components to the component library
Removing components from the component library
Customizing the component library
Handling an unsuccessful compilation

Handling an unsuccessful component library compilation

See Also

If Delphi encounters a problem, it displays an error message, stops building the library, and displays the Code Editor so that you can correct the error. You may be able to correct the problem simply by adding missing directories to the search path.

In the event of an unsuccessful compile, Delphi maintains the Unit list with your most recent modifications so that you can try rebuilding again later in the same session, without having to exit Delphi.

Once you correct the error, you must reopen the Install Components dialog box to rebuild the library. If you do not want to recompile, you can choose Revert to restore the unit list to match the state of the currently installed library.

Note: Choosing Revert works only before you successfully rebuild the library. You cannot use Revert to specify a previous version of the library after it is been successfully rebuilt.

After making changes to the library, you can save the code for the library project file using the .DPR extension.

To save library source code,

1. Choose Options|Environment, then click the Library page tab.
2. Check the Save Library Source Code check box.

The **uses** clause of this library source file lists all the units (.DCU) that are used to build the Component palette.

See also

Adding components to the component library
Removing components from the component library
Installing Visual Basic components
Customizing the component library

About the MainMenu component

See also TMainMenu reference

Purpose

Use the MainMenu component to access the Menu Designer, where you create menus for your form. You can also use the PopupMenu component. Popup menus do not have a menu bar, and typically do not appear until the user right-clicks in the form.

Tasks

Designing menus

Working with menus at run time

See also

Designing a user interface

Creating dialog boxes

About the PopupMenu component

See also TPopupMenu reference

Purpose

Use the PopupMenu component to provide a form or other component with a menu that appears independently of any main menu in the form, typically when the user right-clicks. To create a menu that appears attached to the top of your form, use the MainMenu component.

Tasks

Designing menus

Working with menus at run time

See also

Designing a user interface

Creating dialog boxes

About the Label component

See also TLabel reference

Purpose

Use a label to display text that the user cannot edit, for example, to display a name for components that do not have their own Caption property.

To display text that the user can modify, use the Edit or Memo components. (You can also display read-only text in the Edit component by setting its ReadOnly property to True.)

Tasks

- To ensure that the Label dynamically resizes to accommodate the text it contains, set AutoSize to True (the default).
- To wrap text displayed in the Label, set WordWrap to True.

  Setting WordWrap to True causes text entered after a space to wrap to the next line, unless there is still room in the current line.
- To specify whether you want the Label text to align left, right or centered, use the Alignment property.
- To prevent the Label from obscuring other components, set its Transparent property to True.

  This is useful when using the label in conjunction with graphical objects such as Shapes, and pictures displayed in the Image component. (Because the Label is a non-windowed control, you cannot place it in front of windowed controls such as buttons.)
- To associate a component with the Label, set the Label's FocusControl property to the component you want associated with the Label, and provide an accelerator key as part of the Label's Caption.

  Whenever the user presses the accelerator key, focus shifts to the component specified in FocusControl.

See also

Common component tasks

About the Edit component

See also TEdit reference

Purpose

Use an edit box to read or write a single line of text (the Edit component does not recognize end-of-line characters).

To read and write multiple lines of text, use the Memo component. To display text that the user cannot modify, use the Label component (or set the ReadOnly property of the Edit component to True).

Tasks

- To change the font of the text, use the Font property.
- To work with selected text at run time, use the SelLength, SelStart, and SelText properties, and the SelectAll method.
- To limit the line length, set the MaxLength property.
- To clear text from the edit box, use the Clear method; or use the ClearSelection method to clear only selected text.
- To automatically select the text when the Edit box receives focus, set AutoSelect to True (default).
- To use the Clipboard object with text, use the CopyToClipboard, CutToClipboard, and PasteFromClipboard methods.
- To determine whether the contents of the edit box have changed, use the Modified property.
- To prevent the height of the edit box from changing dynamically to accommodate font changes, set the AutoSize property to False. (True is the default.)

  Note: Only the height, not the length (Width), of the edit box is affected by the AutoSize property.
- To specify a different case for the text in the edit box (all upper, all lower, or mixed case), use the CharCase property.

See also

Common component tasks

About the Memo component

See also TMemo reference

Purpose

The Memo component provides an area for text manipulation. Use the Memo component to read in multiple lines of text, whether entered by the user or programmatically. (Contrast with the Edit component, which handles only single lines of text.)

The Lines property of the Memo component, which contains the text for the memo, is a string list. You can perform many operations on string lists by using the methods and properties of the list. See Working with string lists.

Tasks

- To align text within the memo, use the Alignment property. To align the memo within the form, use the Align property. To ensure that the memo always occupies the client space of the form, set Align to alClient. (Note: This can make working in the form at design time difficult.)
- To change the font of the text, use the Font property.
- To work with selected text at run time, use the SelLength, SelStart, and SelText properties, and the SelectAll method.
- To limit the number of characters the memo will accept, set the MaxLength property.
- To clear text from the memo, use the Clear method; or use the ClearSelection method to clear only selected text.
- To use the Clipboard object with text, use the CopyToClipboard, CutToClipboard, and PasteFromClipboard methods.
- To determine whether the contents of the memo have changed, use the Modified property.
- To enable tabs to be inserted between the memo's text, set WantTabs to True.

Note: Setting WantTabs to True disables users from tabbing out of the memo itself. If your memo appears on a form with other components, leave WantTabs False so that users can tab among all the controls on the form. With WantTabs set to False, users can tab within the memo by pressing Ctrl+Tab.

See also

Common component tasks

Working with string lists.

About the Button component

See also TButton reference

Purpose

Use the Button component to provide buttons users can choose to carry out commands. Button components have two possible click events: OnClick and OnDblClick.

Tasks for controlling focus in a form

Setting the tab order

Setting the focus in a dialog box

Enabling and disabling components

Tasks for creating command buttons in dialog boxes

Most dialog boxes contain buttons to provide the user with the following options:

Exit without accepting changes

Exit and accept changes

Tasks for handling keyboard events

Responding to the OnKeyPress event

Responding to the OnKeyUp

Responding to the OnKeyDown events

Redirecting keyboard events to the form

See also

Common component tasks
Responding to keyboard event
Assigning a glyph to a SpeedButton
Manipulating components

About the CheckBox component

See also TCheckBox reference

Purpose

Use the CheckBox component to present Yes/No or True/False options to the user, particularly where more than one choice at a time is available from a group of choices. Contrast with the RadioButton component, which you use to present such choices when only one choice is available at a time.

Tasks for arranging check box components in the form

- To group check boxes in the form, place them all inside a single Panel, GroupBox, or ScrollBox component. (You must first place the container component on the form, and then place the check box components inside it.)

Tasks for controlling the appearance of the check box

- To check or uncheck the check box, set either its Checked property or its State property to the value you want.
- To make a check box unavailable to the user, set its Enabled property to False.
- To gray the check box at run time, set its State property to cbGrayed.
- To enable the user to gray the check box, set its AllowGrayed property to True.
- To place the check box to the right or left of the text in its Caption, set its Alignment property.

Note: The Alignment property is only supported when Ctrl3D is False (The CTL3D.DLL does not support alignment options in Radio Buttons or Check Boxes).

See also

Customizing the component palette

Grouping components

About the RadioButton component

See also TRadioButton reference

Purpose

The RadioButton component encapsulates the standard Windows option button. Use a radio button to present Yes/No or True/False options to the user. Only one radio button can be selected at a time within the form or other container component. (Contrast with the CheckBox component, which when grouped makes multiple choices available.)

Tasks for arranging RadioButton components in the form

- To group radio buttons in the form, place them all inside a single Panel, GroupBox, or ScrollBox component. (You must first place the container component on the form, and then place the radio button components inside it.)

Tasks for controlling the appearance of the Radio button

- To select or unselect the radio button, set either its Checked property or its State property to the value you want.
- To make a radio button unavailable to the user, set its Enabled property to False.
- To place the radio button to the right or left of the text in its Caption, set its Alignment property.

Note: The Alignment property is only supported when Ctl3D is False (The CTL3D.DLL does not support alignment options in Radio Buttons or Check Boxes).

See also

Customizing the Component palette

Grouping components

About the ListBox component

See also TListBox reference

Purpose

Use the ListBox component to display a scrollable list of items that users can select but cannot directly modify. You need to populate the list box programmatically, as users cannot enter or edit items in the list. Contrast with the ComboBox component, which has a Text property to enable users to enter their own items.

The Items property of the ListBox component, which contains the text of the list, is a string list. You can perform many operations on string lists by using the methods and properties of the list. See Working with string lists.

Tasks

Creating an owner-draw control

Responding to list box changes

Tasks for selecting items

- To determine whether an item is selected, use the Selected property.
- To determine which item is selected, use the ItemIndex property.
- To enable users to select more than one item in the list, set the MultiSelect property to True. With MultiSelect as True, you can use SelCount to count the number of items selected.

Tasks for displaying items

- To display the items in alphanumeric sorted order, set the Sorted property to True.
- To specify columns in the list, set the Columns property to the number of columns you want.
- To control spacing between items in the list, use the ItemHeight properties.
- To ensure that the last item in the list is not initially partially obscured, set the IntegralHeight to True. With IntegralHeight True, the list box cannot be sized so that an item is only partly visible.

See also

Common component tasks
OnMeasureItem event
OnDrawItem event
Working with string lists.

About the ComboBox component

See also TComboBox reference

Purpose

Use the ComboBox component to combine the functionality of a list box and an edit box, enabling users to select from a predefined list or to enter their own text.

The Items property of the ComboBox component, which contains the text of the list, is a string list. You can perform many operations on string lists by using the methods and properties of the list. See Working with string lists.

Tasks for selecting items or text

- To determine which item is selected, use the ItemIndex property.
- To work with selected text at run time, use the SelLength, SelStart, and SelText properties, and the SelectAll method.
- To clear text from the edit box, use the Clear method.

Tasks for displaying items

- To display the items in alphanumeric sorted order, set the Sorted property to True.
- To control spacing between items in the list, use the ItemHeight property.
- To ensure that the last item in the list is not initially partially obscured, set the IntegralHeight to True. With IntegralHeight True, the combo box cannot be sized so that an item in the list is only partly visible.

Other combo box tasks

Creating an owner-draw control

- To choose from a variety of combo boxes to display in your application, use the Style property of the combo box component.

See also

Common component tasks
OnMeasureItem event
OnDrawItem event
Working with string lists.

About the ScrollBar component

See also TScrollBar reference

Purpose

The ScrollBar component provides a way to change which portion of a list or form is visible, or to move through a range by increments. For example, you could use a scroll bar to indicate elapsed time or volume level in a sound recording. By setting the Kind property of the scroll bar, you can make both vertical and horizontal scroll bars available to your application.

Compare the ScrollBar component to the ScrollBox component, which combines scroll bars with a container area, similar to a Panel, where you can place other components.

Note: Some components, such as the Memo component, contain a ScrollBars property. which can be enabled or disabled to display scroll bars in the component.

Tasks

- To specify how the scroll bar responds when the user scrolls, use the OnScroll event.
- To specify an initial position for the scroll box on the scroll bar, use the Position property. You can also use Position to change the position of the scroll box at run time.
- To specify the increment of movement when the user clicks in the scroll bar, use the LargeChange property.
- To specify the increment of movement when the user clicks a scroll arrow, use the SmallChange property.
- To specify how many incremental units exist in the scroll bar, use the Min and Max properties.
- To specify the Position, Min, and Max properties at run time, use the SetParams method.

See also

Common component tasks

TScrollBox component

About the GroupBox component

See also TGroupBox reference

Purpose

The GroupBox component, similar to the Panel and ScrollBox components, acts as a container for other components within your form. Use it to group related components, such as radio buttons or check boxes, so that the components can be manipulated as a single unit.

If you want to place a border around components without putting them in a container, use the Bevel component.

Note: When you place components within container components, you create a new parent-child relationship between the container and the components it contains. The form remains the owner of all components, regardless of whether they are parented within another component.

Tasks

Grouping components

See also

Common component tasks

About the RadioGroup component

See also TRadioGroup reference

Purpose

The RadioGroup component enables you to easily create a component with labeled, grouped radio buttons. Compare with the GroupBox component and the RadioButton component. For instance, the RadioButton component has a Checked property that specifies whether the button is selected. To determine whether a radio button in the RadioGroup component is selected, you must inspect the component's string list index. (See the example.)

The Items property of the RadioGroup component, which contains the text of the caption that appears by the radio button, is a string list. You can perform many operations on string lists by using the methods and properties of the list. See Working with string lists.

Tasks

- To create radio buttons in a radio group
- To display more than one column of radio buttons, use the Columns property.
- To determine the selected radio button, use the ItemIndex or Items property.

At run time, when the user selects a radio button it appears selected and a focus rectangle surrounds the button's caption (string list value).

See also

Common component tasks

RadioButton tasks

GroupBox tasks

Creating radio buttons in a radio group

1. With the RadioGroup component selected in the form, choose the Items property.

   The String List editor appears.

2. In the String List editor, enter the text (or number) you want to appear by the button; or, load a text file that contains the strings you want to use.

   For each line you enter, a radio button will be created in the component.

3. Click OK to close the String List editor.

   The component displays the radio buttons.

About the Panel component

See also TPanel reference

Purpose

The Panel component provides a container area, similar to the GroupBox or ScrollBox components. The Panel includes Alignment, Bevel, and Hint properties that make it easy to create status lines and tool bars in your forms.

Note: When you place components within container components, you create a new parent-child relationship between the container and the components it contains. The form remains the owner of all components, regardless of whether they are parented within another component.

Tasks

Creating a tool bar

Creating a status line

Grouping components

Providing Help Hints

Tasks for customizing the appearance of the panel

- To specify where the panel aligns in relation to the form, set its Align property. (Note that this is distinct from the Alignment property, which is used to align the text within the Panel.)
- To change the raised or lowered appearance of the panel outline, set the BevelInner or BevelOuter properties.
- To change the width of the panel border or its raised or lowered appearance (bevel), set the BorderWidth or the BevelWidth properties.

See also

Common component tasks

About the Timer component

See also TTimer reference

Purpose

Use the Timer component to trigger an event, either one time or repeatedly, after a measured interval. You write the code that you want to occur at the specified time inside the timer component's OnTimer event.

Tasks

- To specify the amount of elapsed time before the timer event is triggered, use the Interval property.
- To discontinue a timed event, set the timer component's Enabled property to False.
- Displaying a SplashScreen

Displaying a SplashScreen

The following two event handlers display and close a form called SplashScreen before the application's main form opens. The constant Startup is declared in Form1's **interface** part. The first event handler calls the Show method of SplashScreen from Form1's OnActivate event.

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  if Startup then
  begin
    Startup := False;
    SplashScreen.Show;
  end;
end;
```

SplashScreen contains a Timer component whose Interval property is set to 3000, so the form is displayed for three seconds and then closes. The form's Close method is attached to the timer component's OnTimer event.

```
procedure TForm2.Timer1Timer(Sender: TObject);
begin
  Close;
end;
```

See also

Common component tasks

About the PaintBox component

See also TPaintBox reference

Purpose

Provides a means of limiting drawing on a form to the specific rectangular area encompassed by the PaintBox.

Tasks

- To access the drawing "surface" of the PaintBox, use the Canvas property.
- To draw on the canvas of the PaintBox, write appropriate code in the event handler for the OnPaint event.

See also

Common component tasks

About the FileListBox component

See also TFileListBox reference

Purpose

Displays files and enables the user to select them in the current directory.

Tasks for controlling display and selection

- To enable multiple selections, use the MultiSelect property.
- To determine what file is selected, use the Selected property.
- To control the type of files that display according to file attributes, use the FileType property.

Tasks for using the component

You normally use the FileListBox component in conjunction with at least two other components: the FilterComboBox and an Edit component. In most cases you would also use a DriveComboBox and a DirectoryListBox component.

- To filter the types of files that display in the FileListBox by file type, specify the name of the FileListBox in the FileList property of a FilterComboBox component.
- To enable the user to specify a file name or wildcard, provide an Edit component and specify its name in the FileEdit property of the FileListBox. If you don't want users to do this, you can set the Edit control's visible to False.
- The fully qualified file name of the user's selection is returned in the FileName property of the FileListBox. To extract the path, file name, or file extension, use the ExtractFilePath, ExtractFileName, or ExtractFileExt function.

For an example of how to use this component in conjunction with other components to select a complete drive, path, and file name, study the FILECTRL example application in the \DELPHI\DEMOS directory.

See also

Creating a select file dialog box with system page components
About the FilterComboBox component
About the DirectoryListBox component
About the DriveComboBox component
Common component tasks

About the DirectoryListBox component

See also TDirectoryListBox reference

Purpose

Enables run-time selection of a directory on the current drive, or the drive selected in an associated DriveComboBox.

Tasks

- To provide run-time selection of the drive whose directories the DirectoryListBox displays, use this component in conjunction with a DriveComboBox. Specify the name of the DirectoryListBox in the DirList property of the DriveComboBox.
- To provide run-time selection of file name(s) residing in the directory selected in a DirectoryListBox, provide a FileListBox component and specify its name in the FileList of the DirectoryListBox.
- To determine which drive is currently accessed by the DirectoryListBox, use the Drive property.
- To determine the current directory for this component or an associated FileListBox, use the Directory property.
- To determine the current user selection, use the Selectedproperty.

For an example of how to use this component in conjunction with other components to select a complete drive, path, and file name, study the FILECTRL example application in the \DELPHI\DEMOS directory.

See also

Creating a select file dialog box with system page components
About the FileListBox component
About the DriveComboBox component
About the FilterComboBox component
Common component tasks

About the DriveComboBox component

See also TDriveComboBox reference

Purpose

Enables run-time selection of a local or network drive.

Tasks

- To extract the name of the drive to which the DriveComboBox points, use the Text or Drive property.
- To provide for selection of a directory, place a DirectoryListBox component on the form and specify its name in the DirList of the DriveComboBox.

For an example of how to use this component in conjunction with other components to select a complete drive, path, and file name, study the FILECTRL example application in the \DELPHI\DEMOS directory.

See also

Creating a select file dialog box with system page components
About the FileListBox component
About the FilterComboBox component
About the DirectoryListBox component
Common component tasks

About the FilterComboBox component

See also TFilterComboBox reference

Purpose

Use the FilterComboBox to control the type of files available to the end user for selection in a FileListBox component.

Tasks

- Specify the file types available for selection by setting values in the Filter property using the Filter editor. This editor appears when you double-click the Value column of the Filter property in the Object Inspector. For example, to include document and text files in the filter, you would specify in the appropriate columns of the Filter editor:

  Document files (*.doc) | *.doc

  Text files (*.txt) | *.txt

- To specify the name of the FileListBox component whose display you want the FilterCompboBox to control, use the FileList property.

You will most likely use the FilterComboBox and its associated FileListBox in conjunction with a DriveComboBox, a DirectoryListBox, and an Edit control. By referencing each other's appropriate properties, these components used together enable the selection and return of one or more path specifications and file names.

For an example of how to use this component in conjunction with other components to select a complete drive, path, and file name, study the FILECTRL example application in the \DELPHI\DEMOS directory.

See also

Creating a select file dialog box with system page components
About the FileListBox component
About the DriveComboBox component
About the DirectoryListBox component
Common component tasks

About the MediaPlayer component

See also TMediaPlayer reference

Purpose

Use the MediaPlayer component to enable your application to control a media playing or recording device such as a CD-ROM player, video player/recorder, or MIDI sequencer.

Deciding how to use the MediaPlayer

How you use and configure the media player component depends largely on the type of device you want to control with it, and whether you want to use all the capabilities of the device or limit your application to only certain ones.

For example, if you are controlling a video cassette recorder and do not want your user to accidentally erase the tape, you could disable or hide the Record segment of the MediaPlayer component.

If the medium has tracks as does a compact disc, you might not need to use the Rewind capability of the MediaPlayer, whereas you would need this for a video tape or Digital Audio Tape (DAT) device.

Tasks for controlling component functions

- To show or hide individual button segments on the MediaPlayer component, use the VisibleButtons property.
- To enable or disable individual segments on the Media Player component, use the EnabledButtons property.
- To control whether or not the device automatically rewinds at the end of the medium before playing or recording, use the AutoRewind property.
- To determine whether or not other components or applications can access the device, use the Shareable property.

Tasks for controlling device access

- To specify or change the type of device controlled by the MediaPlayer component, use the DeviceType property.
- To specify or change the starting position within the currently loaded medium, use the Start property.
- To specify or change the current position of the medium, use the Position property.

Tasks for accessing information on the medium

- To specify or change the media file to play or record, use the FileName property
- To determine the ID of the current device, use the run-time/read-only property DeviceID.
- To determine the position within the currently loaded medium from which playing or recording will start, use the run-time/read-only property StartPos.
- To determine the number of tracks on the open multimedia device, use the run-time/read-only property Tracks.
- To determine the length of a track, use the run-time/read-only property TrackLength for the current TrackNum index.
- To determine the length of a track, use the run-time/read-only property TrackPosition for the current TrackNum index.

Tasks for controlling device operation

The MediaPlayer component provides a number of methods for controlling the operation of a media play or record device. Some of these correspond to one of the button segments of the component (indicated by an asterisk (*)).You have the option of calling any of these methods in your program code as well.

The methods (listed alphabetically) for controlling a device are:

*Back	*Pause	Rewind
Close	PauseOnly	*StartRecording
*Eject	*Play	*Step
*Next	*Previous	*Stop
Open	Resume	

See also

Common component tasks

Windows Multimedia Reference

About the OLEContainer component

See also TOLEContainer reference

Purpose

Use the OLEContainer component to provide your application with the ability to link and embed objects from an OLE server.

When you activate an object inside the OLE container, control transfers to the OLE server application, so the user can access all the functionality of the server application from within your container application.

Tasks

About OLE

Linking vs. embedding

OLE 1.0 and OLE 2.0

OLE data in files

See also
Controlling other applications using DDE

About the DDEClientConv component

See also TDDEClientConv reference

Purpose

Use the DDEClientConv component to provide your application with the ability to establish a DDE conversation with another application. When you place this component on your form, your application becomes a DDE client.

This component works in conjuction with the DDEClientItem component to make your application a complete DDE client.

Tasks

Controlling other applications using DDE

Establishing a link with a DDE client

Establishing a link with a DDE server

Poking data

Creating DDE client applications

- To define the type of connection when linking to to a DDE server application, use the ConnectMode property.
- To filter characters out of the text data transfer from a server application, use the FormatChars property.
- To specify the DDE server application, use the DDEService property.
- To specify the topic of a DDE conversation, use the DDETopic property.
- To specify the executable name for the DDE server application, use the ServiceApplication property.

See also
About DDE

About the DDEClientItem component

See also TDDEClientItem reference

Purpose

Use the DDEClientItem component to define the item of a DDE conversation.. Use it in conjunction with a DDEClientConv component to make your application a DDE client.

Tasks

Controlling other applications using DDE

Establishing a link with a DDE client

Establishing a link with a DDE server

Poking data

Creating DDE client applications

- To specify the DDE client conversation component, use the DDEConv property.
- To specify the item of the DDE conversation, use the DDEItem.

See also
About DDE

About the DDEServerConv component

See also TDDEServerConv reference

Purpose

Use the DDEServerConv component to provide your application with the ability to establish a DDE conversation with another application. When you place this component on your form, your application becomes a DDE server.

This component works in conjuction with the DDEServerItem component to make your application a complete DDE server.

Tasks

Creating DDE server applications

Establishing a link with a DDE client

Establishing a link with a DDE server

See also
About DDE

About the DDEServerItem component

See also TDDEServerItem reference

Purpose

Use the DDEServerItem component the define the topic of a DDE conversation with another application.
Use it in conjunction with a DDEServerConv component to make your application a DDE server.

Tasks

Creating DDE server applications
Establishing a link with a DDE client
Establishing a link with a DDE server

- To specify the DDE server conversation component, use the ServerConv. property.

See also
About DDE

Creating a select file dialog box using system page components

1. Place the following controls on a form:
   - DriveComboBox named DriveCombo1
   - DirectoryListBox named DirList1
   - Edit component named FileEdit1
   - FilterComboBox named Filter1
   - FileListBox name FileList1
   - Label component named PathLabel
2. Select DriveCombo1 and set the `DirList := DirList1`.
3. Select DirList1. Set `DirLabel := PathLabe`l. and `FileList := FileList1`.
4. Select Filter1. Set `FileList := FileList1` and set the Filter property as desired.
5. Select FileList1. Set the `FileEdit := FileEdit1`. If you want to select multiple file names, set `MultiSelect := True`.
6. Code the event handlers for each of the following:
   Form OnCreate eventFileSelFormOnCreateSource
   Filter1 OnChange event
   DriveCombo1 OnChange event
   FileList1 OnChange event

Event handler code for Form OnCreate event

```
begin
  FileList1.Mask := Filter1.Mask;
  Filter1Change(Filter1);
  PathLabel.Caption := DirList1.Directory;
end;
```

Event handler code for Filter1 OnChange event

```
begin
  FileEdit.Text := FileList1.Mask;
  PathLabel.Caption := DirList1.Directory;
end;
```

Event handler code for DriveCombo1 Onchange Event

```
begin
  PathLabel.Caption := DirList1.Directory;
end;
```

Event handler code for FileList1 OnChange **event**

```
begin
  PathLabel.Caption := DirList1.Directory + '\' + FileList1.Filename;
end;
```

The VBX Switch component

This VBX component is provided as a sample.

The VBX Gauge component

This VBX component is provided as a sample.

The VBX Pict component

This VBX component is provided as a sample.

Sample components

The components included on the Samples page of the Delphi Component Palette are provided as samples only. Source code to these sample components is included in the DELPHI\SOURCE\SAMPLES directory of a default installation.

Database Explorer

The Database Explorer enables you to maintain a persistent connection to a remote database server during application development and to work with BDE aliases and metadata objects. With the Database Explorer, you can create, view, and modify:

- BDE aliases.
- Metadata objects such as tables, views, triggers, and stored procedures.
- Users and server security information (not implemented in this field test).

Note: For this field test, the Database Explorer only supports viewing aliases and metadata. It does not support editing and creating aliases or metadata.

The All Database Aliases pane of the Database Explorer displays all the valid aliases defined. Select an alias to display the definition of the alias.

To connect to the database specified by an alias,

1. Select the alias in the All Database Aliases pane.
2. Do one of the following,

- Choose Explorer|Open.
- From the Database Explorer SpeedMenu, choose Open.

When you are connected to a database, the icon in the left pane will turn green.

To expand a database,

- In the All Database Aliases pane, click "+" next to the alias you want to view. The native server object types expand beneath the icon.

Once connected to a database, you can perform SQL operations on the database.

To perform SQL operations,

1. Select the Enter SQL tab.
2. Enter SQL statements in the statement area.
3. Click on the Execute button.

Your SQL statements will execute and the results will be displayed in the table grid.

Edit Breakpoint dialog box

See also

Use the Edit Breakpoint dialog box to add a breakpoint or to change an existing breakpoint. The breakpoint appears in the Code Editor and the Breakpoints List window.

You can set a breakpoint that interrupts program execution in one of the following situations:

- When a line of code you specify is executed.
- When a condition you specify occurs as a line of code is executed.
- When a pass count occurs. A pass count is a condition or line number that is executed a certain number of times.

Use this dialog box to specify where you want a breakpoint to occur:

- To create a breakpoint based on a line number, enter a value for Filename and Line Number.
- To create a breakpoint based on a condition, enter a value for Filename, Line Number, and Condition.
- To create a breakpoint based on the number of times a line is executed, enter a value for Filename, Line Number, and Pass Count.
- To create a breakpoint based on the number of times a condition occurs as a line of code executes, enter a value for Filename, Line Number, Condition, and Pass Count.

To open the Edit Breakpoint dialog box, do one of the following:

- Choose Run|Add Breakpoint.
- Choose Add Breakpoint from the Breakpoint List SpeedMenu.
- Right-click an existing breakpoint in the Breakpoint List window and choose Edit Breakpoint from the Breakpoint List SpeedMenu.

Dialog box options

Filename

Sets or changes the program file for the breakpoint. Enter the name of the program file for the breakpoint.

Line Number

Sets or changes the line number for the breakpoint. Enter or change the line number for the breakpoint.

Condition

Specifies a conditional expression that is evaluated each time the breakpoint is encoutered. Program execution stops when the expression evaluates to True. Enter a conditional expression to stop program execution.

You can enter any valid language expression. However, all symbols in the expression must be accessible from the breakpoint's location, and the expression cannot contain function calls.

Pass Count

Stops program execution at a certain line number after a specified number of passes. Enter the number of passes.

Delphi decrements the pass count number each time the line containing the breakpoint is encountered. When the pass count equals 1, Delphi pauses program execution.

Because Delphi decrements pass counts with each pass, you can use them to determine which iteration of a loop fails. Set the pass count to the maximum loop count and run your program. When the program fails, you can calculate the number of loop iterations by examining the pass count number.

When you use pass counts with conditions, program execution pauses the nth time that the conditional expression is true. Delphi decrements the pass count only when the conditional expression is true.

See also

Breakpoints
Breakpoint list window
Breakpoint list window tasks

Watch Properties dialog box

See also

Use the Watch Properties dialog box to add a watch or to change the properties of an existing watch. The watch appears in the Watch List window.

In addition to changing the properties of a watch, you can change the value of a watch expression. Use the Evaluate/Modify dialog box to change the value of a watch expression.

To open the Watch Properties dialog box, do one of the following:

- Choose Add Watch At Cursor from the Code Editor SpeedMenu.
- Choose Run|Add Watch.
- Choose Add Watch from the Watch List SpeedMenu.
- Right-click an existing watch in the Watch List window and choose Edit Watch from the Watch List SpeedMenu.

Dialog box options

Expression

Specifies the expression to watch. Enter or edit the expression you want to watch. Use the drop-down button to choose from a history of previously selected expressions.

Repeat Count

Specifies the repeat count when the watch expression represents a data element; specifies the number of elements in an array when the watch expression represents an array. Enter the repeat count number.

When you watch an array and specify the number of elements as a repeat count, the Watch List window displays the value of every element in the array.

Digits

Specifies the number of significant digits in a watch value that is a floating-point expression. Enter the number of digits.

Enabled

Enables or disables the watch. Disabling a watch hides the watch from the current program run. When you disable a watch, its settings remain defined, but Delphi does not evaluate the watch.

Disabling watches improves performance of the debugger because it does not monitor the watch as you step through or run your program. When you set a watch, it is enabled by default.

To format the display of a watch expression,

- Select a radio button to specify the format of the display.

See Watch properties format types for complete information.

See also

Watch Properties dialog box

Watch Properties format types

See also

By default, the debugger displays the result of a watch in the format that matches the data type of the expression. For example, integer values are normally displayed in decimal form. If you select the Hexadecimal radio button in the Watch Properties dialog box for an integer type expression, Delphi changes the display format from decimal to hexadecimal.

Character

Shows special display characters for ASCII 0 to 31. By default, such characters are shown using the appropriate C escape sequences (/n, /t, and so on). This format type affects characters and strings.

String

Shows ASCII 0 to 31 as C escape sequences. Use only to modify memory dumps. This format type affects characters and strings.

Decimal

Shows integer values in decimal form, including those in data structures. This format type affects integers.

Hexadecimal

Shows integer values in hexadecimal with the 0x prefix, including those in data structures. This format type affects integers.

Ordinal

Shows integer values as ordinals.

Pointer

Shows pointers in segment:offset notation with additional information about the address pointed to. It tells you the region of memory in which the segment is located and the name of the variable at the offset address. This format type affects pointers.

Record

Shows both field names and values such as (X:1;Y:10;Z:5) instead of (1,10,5).

Default

Shows the result in the display format that matches the data type of the expression. This format type affects all.

Memory dump

Shows the size in bytes starting at the address of the indicated expression. By default, each byte displays two hex digits. Use the memory dump with the character, decimal, hexadecimal, and string options to change the byte formatting.

See also

Watches
Watch list window
Watch list window tasks

Evaluate/Modify dialog box

See also

Use the Evaluate/Modify dialog box to evaluate or change the value of an existing expression or property. You can evaluate any valid language expression, except those that contain:

- Local or static variables that are not accessible from the current execution point.
- Function calls

Delphi enables you to change the values of variables and items in data structures during the course of a debugging session. You can test different error hypotheses and see how a section of code behaves under different circumstances by modifying the value of data items during a debugging session.

When you modify the value of a data item through the debugger, the modification is effective for that specific program run only. Changes you make through the Evaluate/Modify dialog box do not affect your source code or the compiled program. To make your change permanent, you must modify your source code in the Code Editor, then recompile your program.

Modifying values (especially pointer values and array indexes), can have undesirable effects because you can overwrite other variables and data structures. Use caution whenever you modify program values from the debugger.

Keep these points in mind when you modify program data values:

- You can change individual variables or elements of arrays and data structures, but you cannot change the contents of an entire array or data structure.
- The expression in the New Value box must evaluate to a result that is assignment-compatible with the variable you want to assign it to. A good rule of thumb is that if the assignment would cause a compile-time or run-time error, it is not a legal modification value.
- You cannot directly modify untyped arguments passed into a function, but you can typecast them and then assign new values.

Formatting values

When you evaluate an expression, the current value of the expression is displayed in the Result field of the dialog box. If you need to, you can format the result by adding a comma and one or more format following specifiers to the end of the expression entered in the Expression edit box. See Evaluate/modify format specifiers for more information.

To open the Evaluate/Modify dialog box, do one of the following:

- Choose Run|Evaluate/Modify.
- Choose Evaluate/Modify from the Code Editor SpeedMenu.

Dialog box options

Expression

Specifies the variable, field, array, or object to evaulate or modify. Enter the variable, field, array, or object to evaluate or modify.

By default, the word at the cursor position in the Code Editor is placed in the Expression edit box. You can accept this expression, enter another one, or choose an expression from the history list of previously evaluated expressions.

Result

Displays the value of the item specified in the Expression text box after you choose Evaluate or Modify.

New value

Assigns a new value to the item specified in the Expression edit box. Enter a new value for the item if you want to change its value.

Evaluate

Evaluates the expression in the Expression edit box and displays its value in the Result edit box.

Modify

Changes the value of the expression in the Expression edit box using the value in the New Value edit box.

See also

Evaluating and modifying expressions in the Debugger

Evaluate/Modify format specifiers

See also

By default, the debugger displays the result in the format that matches the data type of the expression. Integer values, for example, are normally displayed in decimal form. To change the display format, type a comma (,) followed by a format specifier after the expression.

Example

Suppose the Expression box contains the integer value z and you want to display the result in hexadecimal:

1. In the Expression box, type z,h.
2. Choose Evaluate.

Format specifiers

The following table describes the Evaluate/Modify format specifiers.

Specifier	Types affected	Description
,C	Char, strings	Character. Shows characters for ASCII 0 to 31 in the Pascal #nn notation.
,S	Char, strings	String. Shows ASCII 0 to 31 in Pascal #nn notation.
,D	Integers	Decimal. Shows integer values in decimal form, including those in data structures.
,H or ,X	Integers	Hexadecimal. Shows integer values in hexadecimal with the $ prefix, including those in data structures.
,Fn	Floating point	Floating point. Shows n significant digits where n can be from 2 to 18. For example, to display the first four digits of a floating point value, type ,F4. If n is not specified, the default is 11.
,P	Pointers	Pointer. Shows pointers as 32-bit addresses with additional information about the address pointed to. It tells you the region of memory in which the pointer is located and, if applicable, the name of the variable at the offset address.
,R	Records, classes, objects	Records/Classes/Objects. Shows both field names and values such as (X:1;Y:10;Z:5) instead of (1,10,5).
,nM	All	Memory dump. Shows n bytes, starting at the address of the indicated expression. For example, to display the first four bytes starting at the memory address, type ,4M. If n is not specified, it defaults to the size in bytes of the type of the variable. By default, each byte is displayed as two hex digits. Use memory dump with the C, D, H, and S format specifiers to change the byte formatting.

See also
Evaluate/modify dialog box

About the intergrated debugger

No matter how careful you are when you code, your programs are likely to contain errors, or bugs, that prevent them from running the way you intended. Debugging is the process of locating and fixing errors in your programs.

There are two different types of errors:

- Logical errors
- Run-time errors

Delphi provides debugging features, collectively referred to as the integrated debugger, that let you find and fix errors in your programs.

The integrated debugger is a full-featured debugger that enables you to:

- Control the execution of your program
- Monitor the values of variables and items in data structures
- Modify the values of data items while debugging

Before you use the debugger, you must compile your program with symbolic debug information.

Once you have compiled your program with debug information, you can begin a debugging session by running your program from Delphi. The integrated debugger takes control whenever you run, trace, or step through your program code.

When you run your program under the control of the debugger, it behaves as it normally would; your program creates windows, accepts user input, calculates values, and displays output. When your program is not running, the debugger has control, and you can use its features to examine the current state of the program. By viewing the values of variables, the functions on the call stack, and the program output, you can ensure that the area of code you are examining is performing as it was designed to.

As you run your program through the debugger, you can watch the behavior of your application in the windows it creates. For best results during your debugging sessions, arrange your screen so you can see both the Code Editor and your application window as you debug.

Debugger concepts and tasks

Choose from the following topics for information about using the integrated debugger:

Debugger tasks

Debugger tools

The integrated debugger has several windows, menus, and SpeedMenus. Choose from the following topics for more information:

Windows	About the Breakpoints, Watch, and Call Stack windows
Menus	Lists and descriptions of the debugger menu items
SpeedMenus	Lists and descriptions of the SpeedMenus in each of the debugger windows

Debugger windows

See also

You can use the following windows to help you work in the integrated debugger:

Breakpoint List window	Shows all breakpoints set in the current source code and enables you to add, modify, and remove breakpoints
Watch List window	Shows all watches set in the current source code and enables you to add, modify, and remove watches
Call Stack window	Shows the sequence of function calls and the arguments passed to each function call

See also

Debugger menus

Debugger SpeedMenus

Debugger menus

See also

You can use the following menu items to help you work in the integrated debugger:

View menu

- View|Breakpoints
- View|Call Stack
- View|Watches

Compile menu

- Project|Compile

Run menu

- Run|Run
- Run|Parameters
- Run|Step Over
- Run|Trace Into
- Run|Run To Cursor
- Run|Program Pause
- Run|Program Reset
- Run|Add Watch
- Run|Add Breakpoint
- Run|Evaluate/Modify

See also

Debugger windows

Debugger SpeedMenus

Debugger SpeedMenus

See also

You can use the following SpeedMenus to help you work in the integrated debugger:

- Breakpoint List SpeedMenu
- Watch List SpeedMenu
- Call Stack SpeedMenu

You can also use the following commands on the Code Editor SpeedMenu:

- Code Editor SpeedMenu|Toggle Breakpoint
- Code Editor SpeedMenu|Run To Cursor
- Code Editor SpeedMenu|Evaluate/Modify
- Code Editor SpeedMenu|Add Watch At Cursor

See also

Debugger windows

Debugger menus

See also

Debugger tasks

Run-time errors

See also

A run-time errors occur when your program contains valid statements, but the statements cause errors when they are executed. For example, your program might try to open a nonexistent file, or it might try to divide a number by zero. The operating system detects run-time errors and stops program execution when they occur.

Using the debugger, you can run to a specific program location. From there, you can execute your program one statement at a time, watching the behavior of your program with each step. When you execute the statement that causes your program to fail, you can fix the source code, recompile the program, and resume testing.

See also

About the Debugger

Logic errors

Logic errors

See also

Logic errors occur when your program statements are valid, but the actions they perform are not the actions you intended. For example, logic errors occur when variables contain incorrect values, when graphic images do not look right, or when the output of your program is incorrect.

Logic errors are often the difficult to find because they can show up in unexpected places. You need to thoroughly test your program to ensure that it works as designed. The debugger helps you locate logic errors by monitoring the values of variables and data objects as your program executes.

See also

About the Debugger

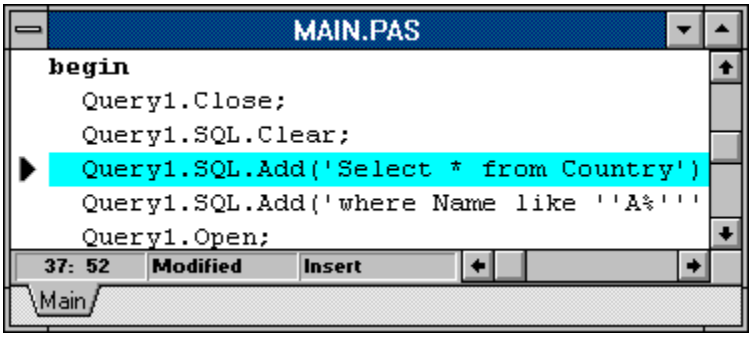
Run-time errors

The execution point

See also

The execution point indicates the next line of source code the debugger will execute. Whenever you pause program execution within the debugger, such as when you run to the cursor or step to a program location, the debugger highlights the execution point.

The following illustration shows the execution point highlighted in the Code Editor:



See also

About the Debugger

Customizing debugging colors

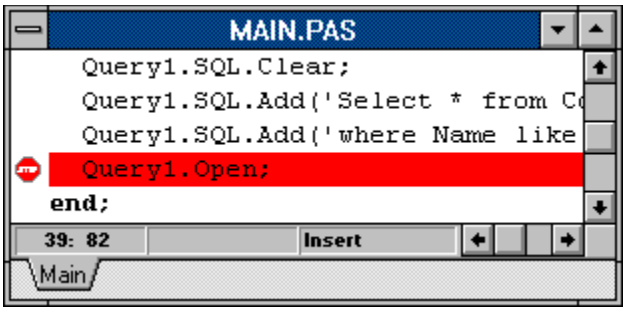
Breakpoints

See also

Breakpoints pause program execution during a debugging session at source code locations that you specify.

You can set breakpoints before potential problem areas of your source code, then run your program at full speed. Your program pauses when it encounters a breakpoint, and the Code Editor displays the line containing the breakpoint. You can then use the debugger to view the state of your program, or to step over or trace into your code one line at a time.

The following illustration shows a breakpoint set in the Code Editor:



You need to set breakpoints on an executable line of code. Breakpoints set on comment lines, blank lines, declarations, or other non-executable lines of code are displayed as invalid breakpoints in the Code Editor, and they are disabled when you run your program.

See also

About the Debugger
Breakpoint list window
Breakpoint list window tasks
Customizing debugging colors

Watches

See also

Watches monitor the changing values of variables or expressions during your program run. As your program runs, the value of the watch changes as your program updates the values of the variables contained in the watch expression.

If the execution point moves to a location where any of the variables in the watch expression are undefined, then the entire watch expression becomes undefined. If the execution point returns to a location where the watch expression can be evaluated, then the Watch List window again displays the current value of the watch expression.

After you enter a watch expression, use the Watch List window to display the current value of the expression.

See also

About the Debugger
Watch list window
Watch list window tasks

Data values

See also

You frequently need to examine the values of variables and expressions to uncover bugs in your program. For example, it is helpful to know the value of the index variable as you step though a **for** loop, or the values of the parameters passed to a function call.

The integrated debugger has the following tools to help you examine the data values in your program:

- The Watch list window
- The Evaluate/modify dialog box
- The Call stack window

Data evaluation operates at the level of expressions. An expression consists of constants, variables, and values contained in data structures, combined with language operators. In fact, almost anything you can use as the right side of an assignment operator can be used as a debugging expression, with the exception of function calls.

See also
About the Debugger

Breakpoint List window

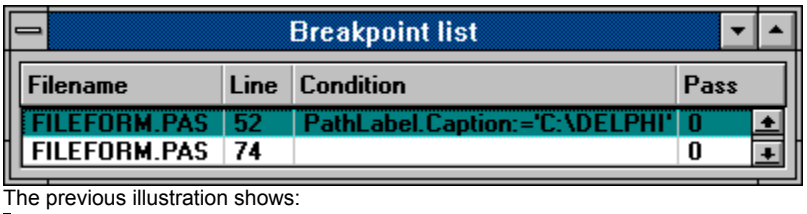
See also

The Breakpoint List window displays a list of current breakpoints and lets you add, edit, delete, enable, and disable breakpoints.

A breakpoint is a place in your program where execution temporarily stops. When the debugger reaches a breakpoint, it pauses your program. You can specify any number of breakpoints in your code.

- To display the Breakpoint List window, choose View|Breakpoints.

The following illustration shows a typical Breakpoint List window:



The previous illustration shows:

- The file containing both breakpoints is FILEFORM.PAS.
- Line numbers 52 and 74 in FILEFORM.PAS contain breakpoints.
- The breakpoint at line 52 is a conditional breakpoint.
- The pass count number for both breakpoints is zero.

The Breakpoint List window lists all breakpoints by their file name and line number. In addition, each breakpoint listing shows any contidtion and pass count associated with the breakpoint. If a breakpoint is either disabled or invalid, then the breakpoint listing is greyed in the Breakpoint List window.

Use the Breakpoint List window to view and maintain all your breakpoints, instaed of searching through your source files. You can view or edit the code at any breakpoint location by using the commands on the Breakpoint List window SpeedMenu.

See also

Debugger windows
About the Debugger
Breakpoints
Breakpoint List window tasks

Watch List window

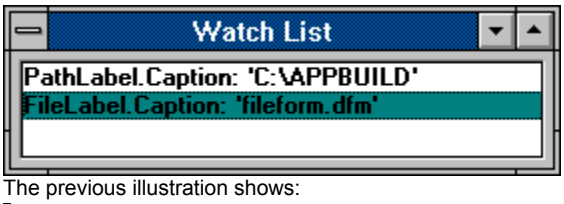
See also

The Watch List window displays the current value of your watches and lets you add, edit, delete, enable, and disable watches.

A watch monitors the changing values of variables or expressions during your program run. As your program runs, the value of the watch changes as your program updates the values of the variables contained in the watch expression.

- To display the Watch List window, choose View|Watches.

The following illustration shows a typical Watch List window:



The previous illustration shows:

- The value of PathLabel.Caption is 'C:\DELPHI'.
- The value of FileLabel.Caption is 'fileform.dfm'.

If the execution point moves to a location where any of the variables in the watch expression are undefined, then the entire watch expression becomes undefined. If the execution point returns to a location where the watch expression can be evaluated, then the Watch List window again displays the current value of the watch expression.

See also

Debugger windows
About the Debugger
Watches
Watch List window tasks

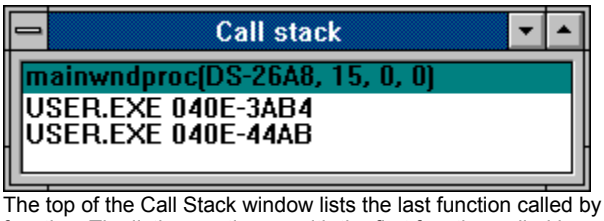
Call Stack window

See also

The Call Stack window displays the function calls that brought you to your current program location and the arguments passed to each function call.

- To display the Call Stack window, choose View|Call Stack.

The following illustration shows a typical Call Stack window:



The top of the Call Stack window lists the last function called by your program. Below this is the listing for the previously called function. The listing continues, with the first function called in your program located at the bottom of the list. Each function listing in the window is followed by the arguments that were passed when the call was made.

Only functions in the currently loaded module are listed in the Call Stack window.

See also

Debugger windows
About the Debugger
Locating function calls

Debugger tasks

See also

The following topics provide an overview of debugger tasks.

General debugging tasks

- Enabling the Debugger
- Generating debugging information
- Specifying program arguments
- Starting a debugging session
- Customizing debugging colors
- Controlling program execution
- Debugging start-up code

Working with breakpoints

- Setting breakpoints
- Locating breakpoints
- Disabling and enabling breakpoints
- Deleting breakpoints
- Modifying breakpoint properties

Working with watches

- Setting watches
- Disabling and enabling watches
- Deleting watches

Examining and modifying data values

- Evaluating and modifying expressions

Working with function calls

- Locating function calls

Handling exceptions

- Handling exceptions in the debugger

See also
About the Debugger

Enabling the Debugger

See also

1. Choose Options|Environment and click the Preferences page tab.
2. Check the Integrated Debugging check box, if necessary (this box is checked by default).
3. Check the Minimize On Run check box to minimize the Delphi environment when you run your program.

See also

Debugger tasks

Symbolic debugging information

See also

Before you debug your program, you first need to compile it with symbolic debug information.

Symbolic debug information enables the debugger to make connections between your program's source code and the machine code that's generated by the compiler. This enables you to view the actual source code of your program while running the program through the debugger.

Symbolic debug information is contained in a symbol table. When you generate symbolic debug information, the compiler stores a symbol table in each associated .DCU file.

Once you have fully debugged your program, it is important to compile it with debugging information turned off. This will reduce the size of your final .EXE file.

To generate symbolic debug information for your project

1. Choose Options|Project to open the Compiler Options dialog box.
2. Click the Compiler Options tab to access the debugging options.
3. Check the Debug Information and Local Symbols check boxes.

To include symbolic debug information in an executable file

1. Choose Options|Project to open the Compiler Options dialog box.
2. Click the Linker tab to access the linker options.
3. Check the Debug Info In EXE check box.

You must include symbolic debug information in your final .EXE file if you want to use Turbo Debugger (the stand-alone debugger).

See also

To include symbolic debug information in an executable file
To generate symbolic debug information for your project
Debugger tasks

Specifying command-line arguments

See also

1. Choose Run|Parameters.
2. In the Run Parameters dialog box, type the arguments to pass to your program when you run it under debugger control.
3. Choose OK.

See also

Debugger tasks

Starting a debugging session

See also

If you find a run-time or logic error in your program, you can begin a debugging session by running your program under the control of the debugger.

To prepare your program to run under debugger control,

1. Generate debug information when you compile your program.
2. Run your program from within Delphi.

You have complete control of your program's execution when you are debugging. You can pause the program at any point to do any of the following:

- Examine the values of variables and data structures
- View the sequence of function calls
- Modify the values of variables to see how different values affect the behavior of your program

You can also use the debugger to monitor each line of your program code as it is executed. See Controlling program execution for more information.

See also

Debugger tasks

Customizing debugging colors

See also

You can customize the colors used to indicate the execution point and the enabled, disabled, and invalid breakpoint lines.

To set execution point and breakpoint colors,

1. Choose Options|Environment. The Environment Options dialog box appears.
2. Select the Editor Colors page tab.
3. From the Element list, select Execution Point, Active Break, Disabled Break, or Invalid Break, then set the background and foreground colors you want.

See also

Debugger tasks

Controlling program execution

See also

You can use the debugger to control whether your program will execute a single line of code, an entire function, or an entire program block. By specifying when the program should run and when it should pause, you can quickly move over the sections that you know work correctly and concentrate on the sections that are causing problems.

The debugger considers multiple program statements on one line as a single line of code; you cannot individually debug multiple statements contained on a single line of text. In addition, the debugger considers a single statement that spans several lines of text as a single line of code.

You can control program execution in the debugger in the following ways:

Run	Runs the current program without pausing at each line. This is equivalent to running a program outside the debugger.
Step Over	Executes a program one line at a time, stepping over procedures while executing them as a single unit.
Trace Into	Executes a program one line at a time, tracing into procedures and following the execution of each line.
Run To Cursor	Runs the current program until the debugger reaches the line in the Code Editor where the cursor is.
Program Pause	Temporarily pauses the execution of a running program.
Program Reset	Ends the current program run and resets the program so you can restart it.
Add Breakpoint	Specifies places in your source code for program execution to stop.

The debugger provides two ways to run a program:

Conventional	Runs the loaded program until you interrupt execution with Run|Program Stop or until your program encounters a breakpoint.
To Cursor	Runs the loaded program to the location of the cursor in the Module window.

See also

Debugger tasks

Debugging start-up code

See also

By default, when you initiate a debugging session by choosing Run|Trace Into or Run|Step Over, Delphi moves the execution point to the first line of code that contains debugging information (this is normally a location that contains user-written code).

You can override this default behavior and step over or trace into the first executable statement in the program when you begin a debugging session. This feature provides the ability to debug the start-up routines of your project.

To step over or trace into the first executable statement in the program,

1. Choose Options|Environment, then click the Preferences page tab.
2. Check the Step Program Block check box.
3. Run your program by choosing Trace Into or Step Over.

See also

Debugger tasks

Run|Step Over

Run|Trace Into

Breakpoint list window tasks

See also

Use the Breakpoint List window to perform any of the following tasks:

- Setting breakpoints
- Locating breakpoints
- Disabling and enabling breakpoints
- Deleting breakpoints
- Modifying breakpoint properties

See also

Breakpoints
Breakpoint list window
Debugger tasks

Setting breakpoints

See also

You can set breakpoints before you begin debugging or while your program is running (by accessing the Code Editor). Your application will halt when it reaches a breakpoint.

To set a breakpoint that interrupts execution at a specific line in your program, do one of the following:

- Click the pointer beyond the left end of the line in the Code Editor.
- Right-click the line and choose Toggle Breakpoint from the Code Editor SpeedMenu.
- Place the insertion point anywhere in the line and press F5.

To set a breakpoint for a condition, do one of the following:

- Choose Run|Add Breakpoint and enter the required information in the Edit Breakpoint dialog box.
- Right-click the Breakpoint List window, choose Add Breakpoint, and enter the required information in the Edit Breakpoint dialog box.
- Right-click an existing breakpoint in the Breakpoint List window, choose Edit Breakpoint, and enter the required information in the Edit Breakpoint dialog box.

See also

Breakpoints
Breakpoint list window
Breakpoint list window tasks

Locating breakpoints

See also

If a breakpoint is not visible in the Code Editor, you can use the Breakpoint List window to quickly locate the breakpoint in your source code.

To scroll the Code Editor to the location of a breakpoint,

- Right-click the breakpoint in the Breakpoint List window and choose View Source.

To scroll the Code Editor to the location of a breakpoint and make the Code Editor active,

- Right-click the breakpoint in the Breakpoint List window and choose Edit Source.

If you choose View Source, the Breakpoint List window remains active so you can modify the breakpoint or go on to view another. If you choose Edit Source, the Code Editor gains focus, enabling you to modify the source code at that location.

See also

Breakpoints
Breakpoint list window
Breakpoint list window tasks

Disabling and enabling breakpoints

See also

Disabling a breakpoint hides the breakpoint from the current program run. When you disable a breakpoint, its settings remain defined, but the breakpoint does not cause your program to stop. When you set a breakpoint, it is enabled by default.

Disabling is useful when you temporarily do not need a breakpoint but want to preserve its settings.

To disable a single breakpoint,

- Right-click the breakpoint in the Breakpoint List window and choose Disable Breakpoint.

To disable all breakpoints in a source code file,

- Right-click the Breakpoint List window and choose Disable All Breakpoints.

To enable a single breakpoint,

- Right-click the breakpoint in the Breakpoint List window and choose Enable breakpoint.

To enable all breakpoints in a source code file,

- Right-click the Breakpoint List window and choose Enable All Breakpoints.

See also

Breakpoints
Breakpoint list window
Breakpoint list window tasks

Deleting breakpoints

See also

When you no longer need to examine the code at a breakpoint location, you can delete the breakpoint from the debugging session. You can delete breakpoints using either the Code Editor or the Breakpoints window:

To delete a single breakpoint, do one of the following:

- Right-click the breakpoint in the Breakpoint List window and choose Delete breakpoint.
- Right-click the breakpoint in the Code Editor and choose Toggle breakpoint.
- Place the insertion point anywhere in the Code Editor line containing the breakpoint and press F5.
- Click the stop-sign glyph at the left end of the line containing the breakpoint in the Code Editor.

To delete all breakpoints in a source code file,

- Right-click the Breakpoint List window and choose Delete all breakpoints.

See also

Breakpoints
Breakpoint list window
Breakpoint list window tasks

Modifying breakpoint properties

See also

You can specify breakpoint properties when you create a breakpoint, or you can edit the properties after creation. Delphi enables you to specify the following breakpoint properties:

- The name of the file containing the breakpoint
- The line number where the breakpoint is located
- A Boolean condition that causes a program break to occur
- A pass count condition that causes a break to occur when a certain situation occurs a specified number of times

How to specify breakpoint properties

Use the Edit Breakpoint dialog box to modify breakpoint properties. To display the Edit Breakpoint dialog box, do any of the following:

- Choose Run|Add Breakpoint.
- Right-click the Breakpoint List window and choose Add Breakpoint.
- Right-click an existing breakpoint in the Breakpoint List window and choose Edit Breakpoint.

See also

Breakpoints
Breakpoint list window
Breakpoint list window tasks

Watch list window tasks

See also

Use the Watch List window to perform any of the following tasks:

- Setting watches
- Disabling and enabling watches
- Deleting watches

See also

Watches

Watch list window

Setting watches

See also

Use the Watch properties dialog box to set a watch that displays the value of any expression.

To display the Watch Properties dialog box, do one of the following:

- Right-click the line and choose Add Watch At Cursor from the Code Editor SpeedMenu.
- Place the insertion point anywhere in the line and press Ctrl+F5.
- Choose Run|Add Watch.
- Right-click the Watch List window and choose Add Watch.
- Right-click an existing watch in the Watch List window and choose Edit Watch.

See also

Watches
Watch list window
Watch list window tasks

Disabling and enabling watches

See also

Disabling a watch hides the watch from the current program run. When you disable a watch, its settings remain defined, but Delphi does not evaluate the watch.

Disabling watches improves performance of the debugger because it does not monitor the watch as you step through or run your program. When you set a watch, it is enabled by default.

To disable a single watch,

- Right-click the watch in the Watch Properties window and choose Disable Watch.
- Double-click the watch in the Watch Properties window to open the Watch Properties dialog box, then uncheck the Enabled check box.

To disable all watches in a source code file,

- Right-click the Watch Properties window and choose Disable All Watches.

To enable a single watch,

- Right-click the watch in the Watch Properties window and choose Enable Watch.

To enable all watches in a source code file,

- Right-click the Watch Properties window and choose Enable All Watches.

See also

Watches
Watch list window
Watch list window tasks

Deleting watches

See also

When you no longer need to examine the value of an expression, you can delete the watch from the debugging session.

To delete a single watch,

- Right-click the watch in the Watch List window and choose Delete Watch.

To delete all watches in a source code file,

- Right-click the watch in the Watch List window and choose Delete All Watches.

See also

Watches
Watch list window
Watch list window tasks

Evaluating and modifying expressions in the Debugger

See also

Use the Evaluate/Modify dialog box to evaluate or change the value of an existing expression or property.

To evaluate an expression or property,

1. Display the Evaluate/Modify dialog box by choosing Run|Evaluate/Modify or by right-clicking the Code Editor and choosing Evaluate/Modify from the Code Editor SpeedMenu.
2. Specify the expression in the Expression edit box.

   To evaluate a property, explicitly specify the property name. For example, enter `Button1.Caption`.
3. Choose Evaluate.

   The value of the item is displayed in the Result edit box.

To change the value of the specified expression or property,

1. Display the Evaluate/Modify dialog box by choosing Run|Evaluate/Modify or by right-clicking the Code Editor and choosing Evaluate/Modify from the Code Editor SpeedMenu.
2. Specify the expression in the Expression edit box.

   To modify a property, explicitly specify the property name. For example, enter `Button1.Caption`.
3. Enter a value in the New Value edit box.
4. Choose Modify.

   The new value is displayed in the Result edit box.

See also
Evaluate/Modify dialog box

Locating function calls

See also

You can use the Call Stack window to locate a function call in your source code quickly.

To scroll the Code Editor to the location of a function call,

- Right-click the function call in the Call Stack window and choose View Source.

To scroll the Code Editor to the location of a function call and make the Code Editor active,

- Right-click the function call in the Call Stack window and choose Edit Source.

If you choose View Source, the Call Stack window remains active. If you choose Edit Source, the Code Editor gains focus, enabling you to modify the source code at that location.

Stepping over function calls

The Call Stack window is useful if you accidentally trace into code you wanted to step over. Using the Call Stack window, you can return to the point from which the current function was called, then resume debugging.

To use the Call Stack window to step over function calls,

1. In the Call Stack window, right-click the calling function (the second function in the Call Stack window) and choose Edit Source.
   The Code Editor becomes active with the cursor positioned at the location of the function call.
2. In the Code Editor, move the cursor to the statement following the function call.
3. Choose Run|Run to Cursor.

See also
Call stack window

Handling exceptions in the Debugger

See also

Delphi enables you to control the way exceptions are handled while you are debugging. In addition, Delphi eases the debugging session by treating most hardware exceptions as Object Pascal language exceptions. This means Delphi traps the hardware exceptions generated by your Delphi application, and you can gracefully recover before your program run ends with a system crash.

If a hardware or language exception occurs while you're debugging a Delphi application, your program halts and Delphi displays the Exception dialog box. If you choose OK, your program run continues, although your program will be in an unstable condition due to the exception.

To pause the program run when an exception occurs,

1. Choose Options|Environment, then click the Preferences page tab.
2. Check the Break On Exceptions check box.

If Break On Exception is checked, Delphi still displays the Exception dialog box when an exception is generated. However, when you choose OK to close the dialog box, Delphi opens the Code Editor with the execution point positioned on the location of the exception.

After examining the program code, it is best to choose Run|Program Reset to teminate your program run. You can then correct the error that caused the exception before resuming the debugging session.

See also

Debugger tasks

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

Designing a user interface

See also

Creating a user interface involves setting properties for forms and their components. Properties determine how a component appears and responds when the application first runs. Changing component properties after the application is running involves writing code.

Many of the components included with Delphi, such as the MainMenu, ScrollBar, Panel, and TabSet components, are specifically designed to round out the look and feel of your user interface.

The following topics describe how to use Delphi components to design the user interface you want.

Adding a picture

Adding scroll bars

Adding a status line

Adding a tool bar

Providing Help Hints for your application components

Creating dialog boxes

Creating tabbed pages

Designing menus

Drawing shapes

Providing an area for text manipulation

See also

Working with forms

Using the dialog components

Providing Help Hints for your application components

See also

The components in the Delphi Component palette and the buttons on the Delphi SpeedBar display Help Hints when the mouse cursor rests over them for more than one second.

Since the Delphi interface was created using Delphi, any component in your form can also display Help Hints.

To provide a Help hint for a component,

1. Specify a value for the component's Hint property.
2. Set the component's ShowHint property to True.

You can control other aspects of Help Hints, such as the hint color, or the length of time before the hint is displayed, by setting these values for the application itself.

To specify a different hint color,

- Set the application's HintColor property to the color you want. (The default color is Yellow.) For example:

```
Application.HintColor := clBlue.
```

To specify the length of time before the hint is displayed,

- Set the application's HintPause property.

For example, to specify that the hint is displayed three (3) seconds after the mouse has paused over the component, set Application.HintPause to 3000.

Note: For more information see Handling application events.

See also
Showing Help Hints in the Delphi IDE

Using the Code Editor

See also

The Delphi Code Editor provides a convenient and reliable way to view and modify your source code. Delphi generates a page in the Code Editor in the following situations:

- Whenever you create a new project
- Whenever you add a form or unit to a project
- Whenever you open a file, even if you do not add it to the project file

You can also use the Code Editor to open text files for viewing or modification.

To open a Code Editor page,

- Add a new form, unit or other file to a project.

To select a Code Editor page,

- At the bottom of the Code Editor, click the tab corresponding to the page you want to view or modify.

To modify text in the Code Editor,

1. Position the cursor at the position you want new text to begin.
2. Type in the new text, pressing Enter to end each line.

To close a Code Editor page do one of the following:

- Select the Code Editor page you wish to close, the choose File|Close.
- Right-click the Code Editor window, then choose Close Page.

To close all Code Editor pages and the project do one of the following:

- Choose Control|Close.
- Double-click the Control-menu box in the upper left corner of the Code Editor.

If you have modified code in a page and have not saved the changes, Delphi opens the Save File As dialog box, where you can enter a file name.

If you have modified the project and have not saved the changes, Delphi opens the Save As dialog box, where you can enter a name for the project.

See also

Behind the scenes in the Code Editor
Code Editor
Code Editor SpeedMenu
Code Editor window
Getting Help in the Code Editor
Keyboard shortcuts
Viewing pages in the Code Editor

Code Editor window

See also

The Code Editor window contains one or more Code Editor pages. The Code Editor window cannot be empty - once you close the last page in the Code Editor window, the window is closed.

You can open multiple files in one Code Editor window. Each file opens on a new page of the Code Editor, and each page is represented by a tab at the bottom of the window. For example, when you open a project, it becomes the first tab in the window. Any other files that you open, such as unit files, become subsequent tabs in the window.

You can open a copy of any editor page, which opens a separate window.

To open a Code Editor window, you can do one of the following:

- Open a file.
- Choose View|New Edit Window.

The New Window command opens a copy of the current page in the Code Editor.
If you have modified the code and not saved the changes, Delphi opens the Save As dialog box, where you can enter a file name.

See also

Adding components to a form

Using the Code Editor

Behind the scenes in the Code Editor

See also

When you add a component to a form, Delphi generates an instance variable, or field, for the component and adds it to the form's type declaration. For example, look at the following code sample, adding a pushbutton component to a blank form.

```
type
  TForm1 = class(TForm)
    Button1: TButton;
  end;
```

Adding the pushbutton changes the form's **type** declaration (`Tform1 = class (Tform)`) by adding the field for the button itself (`Button1: Tbutton;`). You can view similar code being added to the Code Editor, either in your current project or in a new project.

To view code being added in the Code Editor,

1. Drag the form's title bar until you can see the entire Code Editor.
2. Scroll in the Code Editor until the **type** declaration part is visible.
3. Add a component to the form while watching what happens in the Code Editor.

Note: Do not edit any code that Delphi generates. Edit only code that you create.

See also

Adding components to a form

Using the Code Editor

Getting Help in the Code Editor

See also

Context-sensitive Help is available from nearly every portion of the Code Editor. The context is determined by the current position of the cursor.

To get context-sensitive Help from the Code Editor window, do one of the following:

- Place the cursor on the code for which you want Help, then press CTRL+F1.
- Right-click the Code Editor, then choose Topic Search from the SpeedMenu.

If Help is not available for the specific topic you selected, Help displays a message reading, "Help Topic Does Not Exist." If this message appears, you have the three options:

- Return to the main Help screen.
- Select another topic for Help to search.
- Return to a previously viewed topic.

See also
Using the Code Editor

Viewing pages in the Code Editor window

See also

When a page of the Code Editor is displayed, you can scroll through all the data it contains, not just particular sections of your code.

To view a page in the Code Editor, do one of the following,

- If the Code Editor is already the active window, click the tab corresponding to the page you want to view.
- Choose View|Units
- Choose View Unit from the Project Manager SpeedMenu.
- Click the View Unit button on the Project Manager SpeedBar.

See also

Project Manager

Using the Code Editor

Code Editor SpeedMenu

See also

The Code Editor SpeedMenu contains commands for navigating, modifying, and debugging your source code. This menu is unique to the Code Editor, and the commands contained in the menu pertain only to the Code Editor.

The SpeedMenu commands are listed below. To view detailed information on a Code Editor SpeedMenu command, click that command in the list below:

Close Page

Open File At Cursor

Browse Symbol At Cursor

Topic Search

Toggle Marker

Go to Marker

Toggle Breakpoint

Run to Cursor

Evaluate/Modify

Add Watch at Cursor

Read Only

To open the Code Editor SpeedMenu, do one of the following:

- Right-click anywhere in the Code Editor window.
- Press Alt+F10 when the Code Editor window is active.

See also
Using the Code Editor

Close Page (Code Editor SpeedMenu)

Code Editor SpeedMenu

Choose Close Page from the Code Editor SpeedMenu to close the current page in the Code Editor window.

If you have modified code, not saved the changes, and this is the last page open in a file, Delphi opens the Save As dialog box, where you can enter a new file name.

If you are closing the last page in the project and have not saved it yet, Delphi opens the Save As dialog box, where you can enter a name for the project.

Open File At Cursor (Code Editor SpeedMenu)

Code Editor SpeedMenu

Choose Open File At Cursor from the Code Editor SpeedMenu to open the file at the current cursor position.

Delphi searches for files with the default extension of .PAS, unless another file extension is explicitly specified. Similarly, Delphi uses the directory settings for unit and include files specified in the Directories/Conditionals page of the Options|Project dialog box.

To change directory settings for unit and include files,

1. From the main Delphi menu, choose Options|Project.
2. Click the Directories/Conditionals page.
3. Set unit and include directories as you want.
4. Choose OK to put your choices into effect.

Browse Symbol At Cursor (Code Editor SpeedMenu)

See also Code Editor SpeedMenu

Choose Browse Symbol At Cursor from the Code Editor SpeedMenu to open the Symbol Inspection window for the highlighted symbol.

The symbol can be any object, unit, or variable symbol defined in your source code (not only in the current file, but also in any source file compiled and linked as part of the same project).

To browse symbols, you need to compile with the Local Symbols and Symbol Information options on the Compiler Options page of the Project dialog box enabled.

See also
ObjectBrowser

Topic Search (Code Editor SpeedMenu)

Code Editor SpeedMenu

Choose Topic Search from the Code Editor SpeedMenu to display a Help window for the word or token at the cursor in the Code Editor.

If no Help topic exists, the Borland Search dialog box is displayed, with the closest match highlighted.

Toggle Marker (Code Editor SpeedMenu)

See also Code Editor SpeedMenu

Choose Toggle Marker from the Code Editor SpeedMenu to set or delete a marker at the current cursor position.

To set a marker in the Code Editor,

1. Place the insertion point cursor on the line where you want to set a marker.
2. Open the Code Editor SpeedMenu.
3. Choose Toggle Marker.
   A list of available markers (0-9) appears.
4. Choose the marker you want to insert at the current cursor position.
   The marker is displayed in the left margin of the marked line.

To remove individual markers,

1. Place the insertion point cursor on the line with the marker you want to remove.
2. Open the Code Editor SpeedMenu.
3. Choose Toggle Marker.
   A list of available markers (0-9) appears.
4. Choose the marker you want to remove at the current cursor position.
   The marker is removed from the left margin of the marked line.

To remove all markers,

1. Choose the Close Page command from the Code Editor SpeedMenu.
   All markers are removed when the page closes. Note that markers are not saved with the unit file.

See also
Go to Marker

Go To Marker (Code Editor SpeedMenu)

See also Code Editor SpeedMenu

Choose Go To Marker from the Code Editor SpeedMenu to move the cursor to a predefined marker.

To go to a Code Editor marker,

1. Open the Code Editor SpeedMenu.
2. Choose Goto Marker.
   A list of available markers (0-9) appears.
3. Choose the marker to which you want the cursor to move.
   The cursor moves to the marked position.

Note: If you select an undefined marker from the marker list, the cursor remains in its current position.

To remove individual markers,

1. Place the insertion point cursor on the line with the marker you want to remove.
2. Open the Code Editor SpeedMenu.
3. Choose Toggle Marker.
   A list of available markers (0-9) appears.
4. Choose the marker you want to remove at the current cursor position.
   The marker is removed from the left margin of the marked line.

To remove all markers,

1. Choose the Close Page command from the Code Editor SpeedMenu.
   All markers are removed when the page closes. Note that markers are not saved with the unit file.

See also
Toggle Marker

Toggle Breakpoint (Code Editor SpeedMenu)

See also Code Editor SpeedMenu

Choose Toggle Breakpoint from the Code Editor SpeedMenu to toggle a breakpoint on and off at the current cursor position.

If no breakpoint is set when you choose this command, Delphi sets one and turns it on. If a breakpoint is already set, choosing this command toggles the breakpoint off.

To modify breakpoint properties,

- Right-click an existing breakpoint in the Breakpoint List window and choose Edit Breakpoint.

See also

Setting breakpoints
Breakpoints
The Delphi Debugger

Run To Cursor (Code Editor SpeedMenu)

See also Code Editor SpeedMenu

Choose Run To Cursor to run the loaded program up to the location of the cursor in the Module window.

When you run to the cursor, your program is executed at full speed, then pauses and places the execution point on the line of code containing the cursor.

You can use Run To Cursor to run your program and pause before the location of a suspected problem. You can then use Run|Step Over or Run|Trace Into to control the execution of individual lines of code.

An alternative way to perform this command is:

- Choose Run|Run To Cursor.

See also

Controlling program execution

About the Debugger

Evaluate/Modify (Code Editor SpeedMenu)

See also Code Editor SpeedMenu

The Evaluate/Modify command opens the Evaluate/Modify dialog box, which lets you evaluate or change the value of an existing expression.

An alternate way to perform this command is:

- Choose Run|Evaluate/Modify.

See also

Evaluating and modifying expressions
The Delphi Debugger

Add Watch At Cursor (Code Editor SpeedMenu)

See also Code Editor SpeedMenu

The Add Watch At Cursor command opens the Watch Properties dialog box, where you can create and modify watches. After you create a watch, use the Watch List window to display and manage the current list of watches.

Alternate ways to perform this command are:

- Choose Run|Add Watch from the Code Editor SpeedMenu.
- Choose Add Watch from the Watch List SpeedMenu.
- Right-click an existing watch in the Watch List window and choose Edit Watch from the Watch List SpeedMenu.

See also

Setting watches
Watches
About the debugger

Read Only (Code Editor SpeedMenu)

Code Editor SpeedMenu

Choose Read Only from the Code Editor SpeedMenu to make the current open file read only. When a file is read only, you cannot make any changes to the file.

When you mark a file as read only this command is checked on the Code Editor SpeedMenu and "Read only" is displayed in the Code Editor status line.

Code Editor Control menu

See also

The Code Editor's Control-menu box is on the far left of the Code Editor window. Click it once or press Alt+Spacebar to display the Control menu. The commands that appear on this menu affect the Code Editor window.

The commands available on the Control menu are:

Restore

Move

Size

Minimize

Maximize

Close

Switch To

See also
Using the Code Editor

Control | Restore

Code Editor Control Menu

Choose Restore to return the Code Editor window to its previously displayed size and position (except the maximized or minimized sizes).

The Restore command is available only if the Code Editor window is maximized or minimized.

Shortcut

Click the upward/downward double triangle in the upper right corner of the Code Editor.

Control | Move

Code Editor Control Menu

Choose Move to change the position of the Code Editor window using keyboard commands, rather than by dragging it with the mouse.

To move the Code Editor window,

1. Choose Control|Move.
2. Use your keyboard's arrow keys to move the window to the location you want.
3. Press Enter.

Note: The Move command is available only if the Code Editor window is not maximized.

Shortcut

- Drag the Code Editor's title bar.

Control | Size

Code Editor Control Menu

Choose Size to change the size of the Code Editor window using the keyboard, rather than by dragging the window borders.

To resize the Code Editor window,

1. Choose Control|Size.
2. Use your keyboard's arrow keys to move the window borders until the window is the size you want.
3. Press Enter.

Note: The Size command is available only if the Code Editor window is not maximized.

Shortcut

- Drag the Code Editor's window borders. Use the top or bottom border to resize the window vertically, use the left or right border to resize the window horizontally, or use any border corner to resize the window diagonally.

Control | Minimize

Code Editor Control Menu

Choose Minimize to shrink the Code Editor window, together with all other Delphi windows, into a single Delphi icon.

The Minimize command is available only if the Code Editor is not already minimized.

Shortcut

- Click the downward-pointing triangle in the upper right corner of the Code Editor.

Control | Maximize

Code Editor Control Menu

Choose Maximize to enlarge the Code Editor window to fill your entire screen.

The Maximize command is available only if the Code Editor window is not already maximized.

Shortcut

- Click the upward-pointing triangle in the upper right corner of the Code Editor.

Control | Close

Code Editor Control Menu

Choose Close to close all Code Editor pages and the project. If you have modified code and have not saved the changes, Delphi opens the Save As dialog box, where you can enter a new file name.

This command also closes the current project. If you have not saved the project, Delphi opens the Save As dialog box, where you can enter the name of the project.

Shortcuts

- Press Alt+F4.
- Double-click the Code Editor Control-menu box.

Control | Switch To

Code Editor Control Menu

Choose Switch To to display the Windows Task List dialog box, where you can switch from one application to another and rearrange application windows.

Shortcuts

- Press Ctrl+Esc.
- Double-click the Windows desktop, outside of an application.

Write Block To File dialog box

See also

This dialog box enables you to specify the filename and location of an operating system file in which you want to write a block of text you have selected in the Code Editor Window.

When using default key mapping, access this dialog box with: Ctrl+K+W

See also
Read File As Block dialog box

Read File As Block dialog box

SeeAlso

This dialog box enables you to specify the filename and location of an operating system file containing a block of Object Pascal source code which you want to insert in the Code Editor Window at the current cursor position.

When using default key mapping, access this dialog box with: Ctrl+K+R

See also

Write Block To File dialog box

Responding to tab-set changes

Example

When the user selects a tab in a tab-set control by clicking a tab or using the keyboard, the tab set generates an OnClick event. Any other controls that depend on the setting of the tab set need to have their values updated in response to those click events.

For example, with a tab set that contains tabs for each valid disk drive, a click on a different tab indicates a change of drive selection. Other controls, such as directory lists, need to respond to the change.

To respond to changes in a tab set, attach an event handler to the tab set's OnClick event.

Example

The following code, attached to the OnClick event of a tab set named DriveTabSet, sets the Drive property in a directory-outline control to the first letter of the clicked tab:

```
procedure TFMForm.DriveTabSetClick(Sender: TObject);
begin
  with DriveTabSet do
    DirectoryOutline.Drive := Tabs[TabIndex][1];
end;
```

Responding to outline changes

Example

When the user selects an item in an outline by clicking it or using an arrow key, the outline generates a click. Any controls that depend on the currently selected item in the outline need to update themselves in response to those clicks.

For example, in a component such as the directory outline, a click probably indicates a change in the current directory. Related controls, such as file lists, need to respond to this change. However, it is possible that the click was on the directory already selected. Instead of handling the OnClick event, it is more useful to handle the OnChange event, which indicates that something in the directory outline has changed.

To respond to changes in an outline, attach a handler to the outline's OnChange event.

Example

The following code updates both a file list box and a status-bar panel to reflect the current directory in a directory outline component every time the directory outline changes:

```
procedure TFMForm.DirectoryOutlineChange(Sender: TObject);
begin
  FileList.Directory := DirectoryOutline.Directory;
  DirectoryPanel.Caption := DirectoryOutline.Directory;
end;
```

Responding to list box changes

Example

When the user clicks an item in a list box or uses an arrow key to move to an item, that item becomes the selected item, and the list box generates an OnChange event. The application can handle the change and update any dependent controls.

To respond to changes in a list box, attach an event handler to the list box's OnChange event.

For example, a file list box can have an associated panel that displays information about the selected file or files.

Example

The following event handler for a file list box's OnChange event updates a status-bar panel with the name and size of the file currently selected in a file list box.

Note: GetFileSize is a function in the FMXUtils unit installed with the file manager sample application.

```
procedure TFMForm.FileListChange(Sender: TObject);
var
  TheFileName: string;
begin
  with FileList do
  begin
    if ItemIndex >= 0 then{ is there a selected item? }
    begin
      TheFileName := FileName;    { get the file name }
      FilePanel.Caption := TheFileName + ', '    { set panel caption to the name }
        + IntToStr(GetFileSize(TheFileName)) + ' bytes';{ and size }
    end
    else FilePanel.Caption := ''; { blank panel if none selected }
  end;
end;
```

Creating an owner-draw control

Windows list-box and combo-box controls have a style available called "owner draw," which means that instead of using Windows' standard method of drawing text for each item in the control, the control's owner (generally the form) draws each item at run-time. The most common application for owner-draw controls is to provide graphics instead of, or in addition to, text for items.

The following Delphi components support the owner draw style:

- List-box
- Combo-box
- Tab-set

All owner-draw controls contain lists of items. By default, those lists are lists of strings, which Windows displays as text. Delphi enables you to associate an object with each item in a list, and gives you the chance to use that object when drawing items.

To create an owner-draw control in Delphi follow the following three steps:

1. Set the owner-draw style
2. Add graphical objects to a string list
3. Draw owner-draw items

Setting the owner-draw style

Each control that has an owner-draw variant has a property called Style. Style determines whether the control uses the default drawing (called the "standard" style) or owner drawing.

For list boxes and combo boxes, there is also a choice of owner-draw styles, called fixed and variable, as the following table describes. Owner-draw tab sets and string grids are always variable.

Owner-draw style	Meaning	Examples
Fixed	Each item is the same height, with that height determined by the ItemHeight property.	lbOwnerDrawFixed, csOwner-DrawFixed
Variable	Each item might have a different height, determined by the data at run time.	lbOwnerDrawVariable, csOwnerDrawVariable, tsOwnerDraw, all string grids

To set the owner-draw style,

- Set Style property for the owner-draw component.

Adding graphical objects to a string list

See also

Every Delphi string list can hold objects in addition to strings. That is, in addition to its indexed Strings property, which contains strings, a string list also has an Objects property.

Once you have graphical images in an application, you can associate them with the Objects property in a string list. You can either add the objects at the same time as the strings, or associate objects with already-added strings. If you have all the needed data available, you should generally add strings and objects together.

To add a graphical object to an existing string in a string list,

- Assign the object to the Objects property at the same index.

See also

Adding objects to a string list

Adding images to an application

An image control is a non-visual component that contains a graphical image, such as a bitmap. You use image controls to display graphical images on a form, but you can also use them to hold hidden images that you will use in your application. For example, you can store bitmaps for owner-draw controls in hidden image controls.

To add an image to an application,

1. Add an image control to the main form.
2. Set its Visible property to False.
3. Set the Picture property to display the bitmapped image.

Note: The image will be visible on the form but when you run the application they will be invisible.

Drawing owner-draw items

When you set a control's style to owner draw, Windows no longer draws the control on the screen. Instead, it generates events for each visible item in the control. Your application handles the events to draw the items.

To draw the items in an owner-draw control,

1. Size the item.

   If the owner-draw items are all the same size (for example, with a list box style of lsOwnerDrawFixed), you do not need to do this step.

2. Draw the item.

Sizing owner-draw items

Example

Before giving your application the chance to draw each item in a variable owner-draw control, Windows generates a measure-item event. The measure-item event tells the application where the item will appear on the control.

Windows determines what size the item will probably be (generally just large enough to display the item's text in the current font). Your application can handle the event and change the rectangle Windows chose. For example, if you plan to substitute a bitmap for the item's text, you would change the rectangle to be the size of the bitmap. If you want the bitmap and text, you adjust the rectangle to be big enough for both.

To change the size of an owner-draw item, attach an event handler to the measure item event in the owner-draw control. Depending on the control, the name of the event will vary. List boxes and combo boxes use OnMeasureItem. Tab sets use OnMeasureTab.

The sizing event has two important parameters: The index number of the item and the size of that item. That size is variable: the application can make it either smaller or larger. The positions of subsequent items depends on the size of preceding items.

For example, in a variable owner-draw list box, if the application sets the height of the first item to five pixels, the second item starts at the sixth pixel down from the top, and so on. In list boxes and combo boxes, the only aspect of the item the application can alter is height of the item. The width of the item is always the width of the control. In tab sets, only the width of the tabs varies, since the size of the control fixes the height of the tabs. Owner-draw grids allow the application to change both the height and width of each cell.

Example

The following code adjusts the width of an owner-draw tab set's tabs to handle both the default text and a bitmap.

For a tab set, the default width of each tab is the width of the text it contains. To accommodate bitmaps next to the text, you need to handle the measure-item event, increasing the tab width value to include enough space for both the graphic and the text.

The following code, attached to the OnMeasureItem event of an owner-draw tab set, increases the width of each tab to accommodate its associated bitmap:

```
procedure TFMForm.DriveTabSetMeasureTab(Sender: TObject; Index: Integer;
  var TabWidth: Integer); { note that TabWidth is a var parameter}
var
  BitmapWidth: Integer;
begin
  BitmapWidth := TBitmap(DriveTabSet.Tabs.Objects[Index]).Width;
  { increase tab width by the width of the associated bitmap plus two }
  Inc(TabWidth, 2 + BitmapWidth);
end;
```

Note: You must typecast the items from the Objects property in the string list. Objects is a property of type TObject so that it can hold any kind of object. When you retrieve objects from the array, you must typecast them back to the actual type of the items.

Drawing each owner-draw item

Example

When an application needs to draw or redraw an owner-draw control, Windows generates draw-item events for each visible item in the control.

To draw each item in an owner-draw control, attach an event handler to the draw-item event for that control. The events for owner drawing are:

- OnDrawCell
- OnDrawDataCell
- OnDrawItem
- OnDrawTab

The draw-item event contains

- parameters indicating the index of the item to draw
- the rectangle in which to draw
- information about the state of the item (such as whether the item has focus).

The application handles each event by rendering the appropriate item in the given rectangle.

Note: Most draw-item events do not pass a canvas object as a parameter; you normally use the canvas of the control being drawn. Because the tab set also has to render its tab separators between items, it passes a special canvas for item drawing as a parameter.

Example

The following example draws tabs in a tab set that have bitmaps associated with each string:

```
procedure TFMForm.DriveTabSetDrawTab(Sender: TObject; TabCanvas: TCanvas; R: TRect; Index:
Integer; Selected: Boolean);
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap(DriveTabSet.Tabs.Objects[Index]);
  with TabCanvas do
  begin
    Draw(R.Left, R.Top + 4, Bitmap);      { draw bitmap }
    TextOut(R.Left + 2 + Bitmap.Width,    { position text }
      R.Top + 2, DriveTabSet.Tabs[Index]);       { and draw it to the right of the bitmap }
  end;
end;
```

Manipulating files

There are several common file operations built into Object Pascal's run-time library. The procedures and functions for working with files operate at a high level: You specify the name of the file you want to work on, and the routine makes the necessary calls to the operating system for you.

Previous versions of the Pascal language performed similar operations on files themselves, rather than on file names. That is, you had to locate a file and assign it to a file variable before you could, for example, rename the file. By operating at the higher level, Object Pascal reduces your coding burden and streamlines your applications. The lower-level functions are still available, but you should not need them as often.

Choose a topic for more information.

- Deleting a file
- Renaming a file
- Changing a file's attributes

Deleting a file

Example

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

To delete a file, pass the name of the file to the DeleteFile function. DeleteFile returns True if it deleted the file and False if it did not (for example, if the file did not exist or if it was read-only).

Example

The following code handles a click on a File|Delete menu item by deleting the selected file in a file list box, then updating the list so it reflects the deletion.

```
procedure TFMForm.Delete1Click(Sender: TObject);
begin
  with FileList do
    if DeleteFile(FileName) then Update;
end;
```

Changing file attributes

See also Example

Every file has various attributes stored by the operating system as bitmapped flags. File attributes include such items as whether a file is read-only or a hidden file.

Changing a file's attributes requires three steps:

1. Reading file attributes.
2. Changing individual file attributes.
3. Setting file attributes.

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

Reading file attributes

Operating systems store file attributes in various ways, generally as bitmapped flags.

To read a file's attributes, pass the file name to the FileGetAttr function. The return value is a group of bitmapped file attributes, of type Word.

Changing individual file attributes

Because Delphi represents file attributes in a set, you can use normal bitwise operators to manipulate the individual attributes. Each attribute has a mnemonic name defined in the SysUtils unit.

For example, to set a file's read-only attribute, you would do the following:

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

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

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

Setting file attributes

Delphi enables you to set the attributes for any file at any time.

To set a file's attributes, pass the name of the file and the attributes you want to the FileSetAttr procedure.

See also

FindFirst function

Example

The following code reads a file's attributes into a set variable, sets the check boxes in a file-attribute dialog box to represent the current attributes, then executes the dialog box. If the user changes and accepts any dialog box settings, the code sets the file attributes to match the changed settings:

```
procedure TFMForm.Properties1Click(Sender: TObject);
var
  Attributes, NewAttributes: Word;
begin
  with FileAttrForm do
  begin
    FileDirName.Caption := FileList.Items[FileList.ItemIndex];  { set box caption }
    PathName.Caption := FileList.Directory;      { show directory name }
    ChangeDate.Caption := DateTimeToStr(FileDateTime(FileList.FileName));
    Attributes := FileGetAttr(FileDirName.Caption);     { read file attributes }
    ReadOnly.Checked := (Attributes and faReadOnly) = faReadOnly;
    Archive.Checked := (Attributes and faArchive) = faArchive;
    System.Checked := (Attributes and faSysFile) = faSysFile;
    Hidden.Checked := (Attributes and faHidden) = faHidden;
    if ShowModal <> id_Cancel then       { execute dialog box }
    begin
      NewAttributes := Attributes;       { start with original attributes }
      if ReadOnly.Checked then NewAttributes := NewAttributes or faReadOnly
      else NewAttributes := NewAttributes and not faReadOnly;
      if Archive.Checked then NewAttributes := NewAttributes or faArchive
      else NewAttributes := NewAttributes and not faArchive;
      if System.Checked then NewAttributes := NewAttributes or faSysFile
      else NewAttributes := NewAttributes and not faSysFile;
      if Hidden.Checked then NewAttributes := NewAttributes or faHidden
      else NewAttributes := NewAttributes and not faHidden;
      if NewAttributes <> Attributes then{ if anything changed... }
        FileSetAttr(FileDirName.Caption, NewAttributes);{ ...write the new values }
    end;
  end;
end;
```

Dragging and dropping

Dragging and dropping of items on a form can be a handy way to enable users to manipulate objects in a form. You can let users drag entire components, or let them drag items out of components such as list boxes into other components.

There are four essential elements to drag-and-drop operations:

1. Starting a drag operation
2. Accepting dragged items
3. Dropping items
4. Ending a drag operation

Starting a drag operation

See also Example

Every control has a property called DragMode that controls how the component responds when a user begins dragging the component at run time. If DragMode is dmAutomatic, dragging begins automatically when the user presses a mouse button on the control. A more common usage is to set DragMode to dmManual (which is the default) and start the dragging by handling mouse-down events.

To start dragging a control manually, call the control's BeginDrag method.

BeginDrag takes a Boolean parameter called Immediate. If you pass True, dragging begins immediately, much as if DragMode were dmAutomatic. If you pass False, dragging does not begin until the user actually moves the mouse a short distance. Calling BeginDrag(False) allows the control to accept mouse clicks without beginning a drag operation.

You can also place conditions on whether to begin dragging, such as checking which button the user pressed, by testing the parameters of the mouse-down event before calling BeginDrag.

See also

Accepting dragged items

Example

The following code handles a mouse-down event on a file list box by beginning dragging only if it was the left mouse button pressed:

```
procedure TFMForm.FileListBox1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then { only drag if left button pressed }
    with Sender as TFileListBox do       { treat Sender as TFileListBox }
    begin
      if ItemAtPos(Point(X, Y), True) >= 0 then  { is there an item here? }
        BeginDrag(True);  { if so, drag it }
    end;
end;
```

Accepting dragged items

Example

When a user drags something over a control, that control receives an OnDragOver event, at which time it must indicate whether it can accept the item if the user drops it there. Delphi changes the drag cursor to indicate whether the control can accept the dragged item.

To accept items dragged over a control, attach an event handler to the control's OnDragOver event. The drag-over event has a variable parameter called Accept that the event handler can set to True if it will accept the item.

Setting Accept to True specifies that if the user releases the mouse button at that point, dropping the dragged item, the application can then send a drag-drop event to the same control.

Setting Accept to False specifies that the application will not drop the item on that control. This means that a control should never have to handle a drag-drop event for an item it does not know how to handle.

The drag-over event includes several parameters, including the source of the dragging and the current location of the mouse cursor. The event handler can use those parameters to determine whether to accept the drop. Most often, a control accepts or rejects a dragged item based on the type of the sender, but it can also accept items only from specific instances.

Example

In the following example, a directory outline accepts dragged items only if they come from a file list box:

```
procedure TFMForm.DirectoryOutline1DragOver(Sender, Source: TObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean);
begin
  if Source is TFileListBox then
    Accept := True;
end;
```

Dropping items

Example

Once a control indicates that it can accept a dragged item, it should then also define some way to handle the item should it be dropped. If a user sees the mouse cursor change to indicate that a control will accept the item being dragged, it is reasonable for the user to then expect that dropping the item there will accomplish some task.

To handle dropped items, attach an event handler to the OnDragDrop event of the control accepting the dropped item.

Like the drag-over event, the drag-drop event indicates the source of the dragged item and the coordinates of the mouse cursor over the accepting control. These parameters enable the drag-drop handler to get any needed information from the source of the drag and determine how to handle it.

Example

In the following code, a directory outline accepting items dragged from a file list box can move the file from its current location to the directory dropped on:

```
procedure TFMForm.DirectoryOutline1DragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  if Source is TFileListBox then
    with DirectoryOutline do
      ConfirmChange('Move', FileList.FileName, Items[GetItem(X, Y)].FullPath);
end;
```

Ending a drag operation

Example

When a dragging operation ends, either by dropping the dragged item or by the user releasing the mouse button over a control that does not accept the dragged item, Delphi sends an end-drag event back to the control the user dragged.

To enable a control to respond when items have been dragged from it, attach an event handler to the OnEndDrag event of the control.

The most important parameter in an OnEndDrag event is called Target, which indicates which control, if any, accepted the drop. If Target is **nil**, it means no control accepted the dragged item. Otherwise, Target is the control that accepted the item. The OnEndDrag event also includes the x- and y-coordinates on the receiving control where the drop occurred.

Example

In this example, a file list box handles an end-drag event by refreshing its file list, assuming that dragging a file from the list changed the contents of the current directory:

```
procedure TFMForm.FileListBox1EndDrag(Sender, Target: TObject; X, Y: Integer);
begin
  if Target <> nil then FileList.Update;
end;
```

Reusing forms as DLLs

See also

When you create a form that you want to use in multiple applications, especially when the applications are not Delphi applications, you can build the form into a dynamic-link library (DLL). A DLL is a compiled executable file, so applications written with tools other than Delphi can call them. For example, you can call a DLL from applications created with C++, Paradox, or dBASE.

DLLs are standalone files that contain the overhead of the component library (about 100K). You can minimize this overhead by compiling several forms into a single DLL. For example, suppose you have a suite of applications that all use the same dialog boxes for checking passwords, displaying shared data, or updating status information. You can compile all of these dialog boxes into a single DLL, allowing them to share the component-library overhead.

Building a form into a DLL takes three steps:

1. Declaring interface routines
2. Modifying the project file
3. Accessing routines stored in DLLs

See also
Writing DLLs

Declaring interface routines

See also Example

When you are compiling an application into a DLL, interface routines enable you to access the routine in the DLL from an outside application.

Adding an interface-routine declaration involves declaring a procedure or function in the interface section of the unit to be compiled into the DLL, and following that declaration with the **export** directive.

When writing interface routines that will be called from languages other than Object Pascal, you must declare parameters and return values using types that are available in the calling language. For example, you should pass strings as null-terminated arrays of characters (the Object Pascal type PChar) rather than Object Pascal's native **string** type.

After declaring an **interface** routine, you can define the routine in the **implementation** section of the unit.

See also

Accessing routines stored in DLLs
Functions
Interface
Procedures
Writing DLLs

Example

The following example declares the function GetPassword as a interface routine. The exports section includes the GetPassword routine name to ensure that the function is successfully exported.

```
unit PassForm;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Forms, Controls, Forms, Dialogs, StdCtrls, Buttons;
type
  TPasswordForm = class(TForm)
  ...{ various declarations go here }
  end;
var
  PasswordForm: TPasswordForm;
function GetPassword(APassword: PChar; hAppHandle: THandle): WordBool;
exports GetPassword;
implementation
function GetPassword(APassword: PChar; hAppHandle: THandle): WordBool;
begin
  Application.Handle := hAppHandle; { Associate the DLL's Application handle with the
                loading Application's handle. }
  PasswordForm := TPasswordForm.Create(Application);
  try
    if PasswordForm.ShowModal = mrOK then
    begin
      {Code to validate entered password values here}
      Result := True;
    end;
  finally
    PasswordForm.Free;
  end;
end;
end.
```

Compiling a project into a DLL

See also Example

When you are compiling a project into a DLL, you need to make the following edits in the project file:

1. Change the reserved word **program** in the first line of the file to **library**.
2. Remove the Forms unit from the project's **uses** clause.
3. Remove all lines of code between the **begin** and **end** at the bottom of the file.
4. Below the **uses** clause, and before the **begin..end** block, add the reserved word exports, followed by the names of the interface routines and a semicolon.

   Delphi will not create the list of interface routines to be exported.

After these modifications, when you compile the project, it produces a DLL instead of an application. Applications can now call the DLL to open the wrapped dialog box.

See also

Declaring interface routines

Writing DLLs

Example

The following example shows a typical project file before and after modification:

```
program Password;

uses Forms,
     PassForm in 'PASSFORM.PAS' {PasswordForm};

{$R *.RES}
begin
  Application.CreateForm(TPasswordForm, PasswordForm);
  Application.Run;
end.
```

After modifications:

```
 library Password;            { 1. reserved word changed }

 uses { 2. removed Forms, }
      PassForm in 'PASSFORM.PAS' {PasswordForm};

 exports
   GetPassword;               { 4. add exports clause }
 {$R *.RES}
begin{ 3. remove code from main block }
end.
```

Working with forms

See also

Forms designed in Delphi can be reused among Delphi projects. They can also be saved as dynamic-link libraries (DLLs) so you can load them into projects built with other applications, such as C++, Paradox, or dBase. Delphi makes getting started easy by providing form design tools such as templates.

Before you begin designing the forms that will make up your application's user interface, you need to be comfortable working with forms in a variety of ways. Choose from the following topics for more information:

Creating a new form

Viewing forms and units

Calling forms from forms

Using templates

MDI and SDI forms

Sharing forms with other projects

Saving forms as ASCII

See also

Designing a user interface

Creating a new form

See also

There are several ways to create a new form:

- Start Delphi.

  Delphi generates a blank form, its associated unit, and a project file.

  Note: If you have chosen a different form or project as the default from the Object Repository dialog box, then the default form opens, instead of a blank form.

- Create a new application by choosing File|New|Application.

  Delphi generates a new form and unit file whenever it generates a new application.

- Add a new form to an existing project by choosing File|New|Form.

See also

Adding a form to the Object Repository

Calling forms from forms

See also

Before you can reference a form or any of its methods from a method in different form, you need to make the form containing the method call aware that the other form is part of the project. Adding a form to your project adds it to the **uses** clause in the .DPR file, but not to the **uses** clause of any other units in the project. Because Delphi cannot anticipate the method calls you might write in a given form, you need to modify your unit file's **uses** clause yourself.

For example, if you added the About Box Form Template to your project and then called it from a menu item in your application's main form, you would need to add its associated unit (About, or the name you gave it when you added it) to the main form unit's **uses** clause.

Enabling forms to reference each other

When two forms must reference each other, it's possible to cause a "Circular reference" error when you compile your program. To avoid such an error, do one of the following:

- Place both **uses** clauses, with the unit identifiers, in the **implementation** parts of the respective unit files. (This is the usual means of allowing two forms to reference each other.)
- Place one **uses** clause in an **interface** part and the other in an **implementation** part. (You rarely need to place another form's unit identifier in this unit's **interface** part.)

Do not place both uses clauses in the **interface** parts of their respective unit files. This will generate the "Circular reference" error at compile time.

See also

Sharing forms with other projects

Sharing forms with other projects

Before you begin designing the forms that will make up your application's user interface, you should think about whether you plan to make these forms available to other developers or users. The easiest way to share a form is to simply add an existing form to a project. Delphi provides Form Templates for just that purpose. Form Templates are predesigned forms that you can easily reuse in your projects.

Modifying a shared form

When you share a form among projects, any changes you make to the form at design time affect the shared form. If you have several projects that all use the same form, changes to the shared form that you make in any of the projects show up in all of them. This might be your intention. If it is not, however, there are several ways to prevent this. You can:

- Save the form under a different name and use the renamed form in your project instead of the original.

  This is similar to using a Form Template in your project.
- Make the changes to the form at run time instead of at design time.
- Make the shared form into a component that developers can customize at design time. (For more information, see Creating a new component.)

A third way to reuse a form is to make it into a DLL. For more information, see Building a dialog box into a DLL.

Saving a form under a different name

See also

You choose File|Save As to save a form under a different name or location. However, the .DFM file is not listed separately from the unit file in the Save As dialog box. Even if you make design-time modifications to a form without changing any of the underlying source code, only the .PAS file is displayed when you go to save the file. The two files are inseparable; saving one saves the other.

Saving a form under a different name is a good way to ensure that modifications you make to the form do not affect any other projects that might also be using the form.

To save a form under a different name,

1. Select the form you want to save.
2. Choose File|Save As.
3. In the Save As dialog box, specify a name and a directory for the file.
4. Choose OK.

See also

Adding a form to the Object Repository

Saving files

Saving projects

Viewing forms and units

When you are working with forms at design time, you can simultaneously have many forms open. You can switch among several open forms by bringing the form you want to work on to the front. Similarly, you can view the unit source code for the form you are working in.

To view a specific form in a project,

1. Choose View|Forms to display View Form dialog box.
2. In the View Form dialog box, select the form you want to give focus to, then choose OK.

To view a specific unit in a project,

1. Choose View|Units to display the View Unit dialog box.
2. In the View Unit dialog box, select the unit you want to view, then choose OK.

You can also use the Delphi Project Manager to navigate among the units and forms in your project.

Using templates

See also

Delphi provides you with many predesigned forms and projects to help you design your application interface. By using Form Templates, simple function calls, and Delphi components, you can create a wide variety of forms, from the simplest to the most complex.

Project Templates give you a selection of application designs that you can use as a starting point when building your own applications.

Form Templates provide an array of predesigned forms when developing your user interface.

The following topics discuss ways you can add predesigned forms to your application:

Displaying a message box

Using simple input forms

Adding a form to your project

Creating a form by using the database form expert

See also
Using the Object Repository

Displaying a message box

Example

Message boxes display text to the user and must be manually cleared from the screen before the application can continue. Message boxes can be used for a variety of purposes, including:

- Displaying warning information to the user
- Displaying error messages
- Requesting confirmation before closing a file or exiting the application

The Delphi Dialogs unit contains several functions that display predesigned dialog boxes to the user, based on the parameters you specify. The MessageDlg function displays a message box that contains text you specify, and various captions, symbols, and buttons.

MessageDlg displays the message box in the center of the screen by default. You can use the MessageDlgPos function to create a message box that appears at a screen location you specify.

Example

The following example calls a message dialog box when you click Button1.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MessageDlg('Save changes?', mtConfirmation,mbYesNoCancel , 0);
end;
```

Using simple input forms

See also Example

While MessageDlg displays an informational form that requires only a button click from the user, the following two functions display forms requiring a single line of input from the user.

- InputBox
- InputQuery

Both InputBox and InputQuery create a dialog box with OK and Cancel buttons.

Use InputBox if you simply want the string the user enters returned if the user chooses OK, and not returned if the user chooses Cancel, or presses Esc to exit the dialog box.

Use InputQuery if you want to be able to interpret the user's action (OK, Cancel or Esc) as a Boolean variable that the program can recognize.

Example

For an illustration of how InputBox works, try the following.

1. Start a new, blank project.
2. Add a Button component and a Label component to Form1.
3. Generate the following OnClick event for Button1:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := InputBox('Password Entry Form', 'Enter Password', '')
end;
```

   Leaving the ADefault parameter blank means no text appears in the input box initially.
4. Run the program and choose Button1.

   The input box appears.
5. Type some text in the input box, and choose OK.

   The text appears in Label1.
6. Choose Button1 again to display the input box.
7. Type some text, and this time, choose Cancel or press Esc to exit the dialog box.

   The text is not displayed in Label1; instead, the ADefault string is returned. In this case, the string is blank, so the text in Label1's Caption property is cleared, but no other text replaces it.

See also

Masking password characters

Masking password characters

See also Example

You can mask the characters that a user enters into an edit or memo field. Use the PasswordChar property of the Edit, DBEdit, or MaskEdit components to display any characters the user enters as special characters, such as asterisks (*) or pound signs (#).

To see an example, open the Password Dialog Form Template.

Note: The PasswordChar property of the Edit component in the Password Dialog Form Template, Password, has already been set to *. When the user enters text in this dialog box at run time, only the asterisk character is displayed, so that the user's password is not visible onscreen. (You can enter any character as the PasswordChar property value.)

Example

The following example displays the Password Dialog Form Template when Button1 is clicked.

1. Start a new, blank project and add the Password Dialog Form Template to it.
2. Add a button to Form1, and write the following OnClick event handler:

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

3. Add Password to Unit1's **uses** clause, and run the application.
4. Choose Button1.

   The Password dialog box appears.
5. Type some text into the edit box.

   Only asterisks appear.

See also

TEdit component
TDBEdit component
TMaskEdit component
Adding a form to your project

Adding a form to the Object Repository

Once you've designed a custom dialog box, you might want to reuse it in other projects. The best way to do this is to add the form to the Object Repository.

Saving a form as an object is similar to saving a copy of the form under a different name. When you save a form as a object, however, it then appears in the Object Repository. You specify the bitmap and description of the object that appears in this list.

To add your current form to the Object Repository,

1. Right-click the form to open the Form SpeedMenu.
2. From the Form SpeedMenu, choose the Add To Repository command.

   The The Add To Repository dialog box appears.
3. In the Title edit box, specify a name for the object.
4. In the Description edit box, type a brief description of this object.
5. Choose the Page on which the form should appear in the New Items dialog box.
6. You can specify an Author of the form, which shows only in the detailed view of the Object Repository.
7. To specify an icon for the object, choose the Browse button.

   The Select Bitmap dialog box appears.
8. Locate and select the bitmap (if any) you want to use, and choose OK to exit the Select Bitmap dialog box.
9. Choose OK to accept your specifications, and exit the Add To Repository dialog box.

The next time you choose File|New|New Form, your template appears in the templates list, with the bitmap you chose to represent it, and the description you entered.

Add To Repository dialog box

You use the Add To Repository dialog box to add new forms to the Object Repository. From the Form SpeedMenu, choose Add To Repository to open the Add To Repository dialog box.

Add To Repository dialog box

Forms

Displays the names of the forms in the current project.

Title

Use this box to specify the name of the form to be added to the Repository.

Description

Use this box to type a description of the form. The description is dislayed when you select the View Details option from the SpeedMenu in the New Items dilaog box.

Page

Displays a list of the current pages in the Object Repository. Use this option to select the Repository page to which you want to add your form.

Author

Use this box to type the name of the creator of the form you want to add to the Repository.

Browse button

Use the Browse button to change the icon for the form you want to add to the Repistory. The icon represents the bitmap that will appear in the New Items and Repository Options dialog box. To change the bitmap, click the Browse button to open the Select Bitmap dialog box.

You can use a bitmap of any size, but it will be cropped to 60 x 40 pixels.

Saving forms as ASCII

Delphi enables you to save .DFM form files as ASCII text. You can then modify a form by using any standard text editor to edit the text file. Later, you can load the ASCII file in the Code Editor and convert it back into the binary .DFM format. This can be useful when using Delphi projects in a team development environment, or whenever version control is an issue, since most version-control mechanisms depend on a text-based comparison.

To convert a form file to ASCII format,

1. Choose File|Open. (You need not have a project open.)
2. Select the .DFM file you want to convert to ASCII.

   If the form's unit file (.PAS) is open, you are prompted to save changes and close it before the .DFM file opens in the Code Editor.
3. Choose File|Save As.
4. In the Save As dialog box, rename the file using a .TXT file extension.

   Using the .TXT file extension automatically causes the .DFM file to convert to ASCII text.
6. Repeat these steps for any other .DFM files you want to convert to ASCII.

Note You cannot have the same form open simultaneously in both visual and text formats.

To see an example of a form converted to an ASCII file, see Sample ASCII Form File.

Converting text files back to .DFM format

After editing an ASCII form file, you can convert the text-based form back into its regular binary .DFM format. Once you understand the structure and content of a form file, it's even possible for you to create a form by writing its structure as a text file and converting that to binary format. However, it's normally far more efficient to create forms with the visual tools that Delphi provides.

To convert an ASCII-based form file back into binary (.DFM) format,

1. Choose File|Open. (You need not have a project open.)
2. Select the text file you want to convert to binary .DFM format.

   The file opens in the Code Editor.
3. Choose File|Save As.
4. In the Save As Type combo box, choose Form File (*.DFM).
5. Name the file with a .DFM extension in the File Name edit box.
6. Repeat this process for any other files you want to convert to binary .DFM format.

Sample ASCII form file

Here is the text output of the Password Dialog Form Template. It is a summary listing of the objects that make up the form, and the main visual properties of those objects.

```
object PasswordDlg: TPasswordDlg
  Left = 200
  Top = 99
  ActiveControl = Password
  AutoScroll = False
  BorderStyle = bsDialog
  Caption = 'Password Dialog'
  Font.Color = clBlack
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = [fsBold]
  PixelsPerInch = 96
  Position = poScreenCenter
  ClientHeight = 93
  ClientWidth = 236
  object Label1: TLabel
    Left = 8
    Top = 9
    Width = 92
    Height = 13
    Caption = 'Enter password:'
  end
  object Password: TEdit
    Left = 8
    Top = 27
    Width = 220
    Height = 20
    TabOrder = 0
    PasswordChar = '*'
  end
  object OKBtn: TBitBtn
    Left = 66
    Top = 59
    Width = 77
    Height = 27
    TabOrder = 1
    Kind = bkOK
    Margin = 2
    NumGlyphs = 2
    Spacing = -1
  end
  object CancelBtn: TBitBtn
    Left = 152
    Top = 59
    Width = 77
    Height = 27
    TabOrder = 2
    Kind = bkCancel
    Margin = 2
    NumGlyphs = 2
    Spacing = -1
  end
end
```

MDI and SDI forms

See also Example

Any form you design can be implemented in your application as a Multiple Document Interface (MDI) or Single Document Interface (SDI) form.

It's probably more common to design an MDI form as such from the beginning, but you can change forms you create into MDI or SDI forms simply by changing their FormStyle property.

The following table displays the possible FormStyle property settings and their purpose.

FormStyle	Creates	Comments
fsNormal	A standalone (SDI) form	
fsMDIForm	An MDI parent or frame form	This form can contain other forms at run time.
fsMDIChild	an MDI child form	This form is contained by the MDI parent at run time.
fsStayOnTop	A standalone form that stays on top of other open forms at run time.	

See also

Multiple Document Interface (MDI) applications

Example

To demonstrate how MDI parent forms contain child forms, try the following.

1. Start a new, blank project and add a new, blank form.
2. Set the FormStyle property for Form1 to fsMDIForm, and for Form2 to fsMDIChild.
3. Add a Panel component to the lower right corner of Form1, and change its Alignment property to AlTop.
4. Add a SpeedButton component to the Panel, and generate the following OnClick event handler for it:

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  with TForm2.Create(Self) do show;
end;
```

5. Add Unit2 to Unit1's **uses** clause.
6. Choose Options|Project|Forms and remove Form2 from the autocreate list.
7. Run the program, and click SpeedButton1.

   Form2 opens inside Form1. You can resize and move either form, but the child form will always stay within the borders of the parent.

Adding forms to your project

The Object Repository contains many kinds of forms, including starndard forms, dialog boxes, data modules, and any forms you have added.

To add a form from the Object Repository to your project,

1. With a project open, choose File|New.

   The New Items dialog box appears.
2. Choose the forms page.

   The forms page opens with the default new form highlighted.
3. Select the form you want to add.
4. Choose one of the sharing options (Copy, Inherit, or Use) from the radio buttons near the bottom of the dialog box.

   For details on sharing options, see Object Repository usage options.
5. Choose OK.

   Delphi adds the form and its associated unit to the associated unit to the project you have open. You can use this form the way you would use any form in a project. You can customize it, add components, code to it, and so on.

   Any modifications you make might affect the the original item in the Object Repository, depending on the sharing option you choose.

Note: With no project open, it is still possible to choose File|New and select a form from the Object Repository. The form's unit file then opens as a reference file. If you susequently open a project or create a new project, the open template form *is not part of that project.* To save it as part of the project, you must explicitly add it to the project.

Glossary

A

abstract
accelerator key
actual parameter
actual variable
alias
ancestor
application
array
ASCII

B

base type
batch operation
BDE
BDE Configuration Utility
BLOB
block
Boolean
Borland Database Engine
breakpoints
byte

C

callback routines
call stack
canvas
case variant
char
child
class
class method
client
client area
column
compile
compiler directive
compile time
compile-time error
complete evaluation
component
conditional symbol
const parameter
constant
constant address expression
container application
container component
control

D

data

data access component
data-aware
data control component
data type
database
database server
dataset
DDE client
DDE conversation
DDE server
default ancestor
default event
default new form
default new project
derive
descend
descendant
design time
detail table
dispatch
drag
dynamic
dynamic data exchange (DDE)
dynamic-link library (DLL)

E

embedding
encapsulate
end user
enumerated data type
exception
exception handler
execution point
expressions
event
event handler

F

feature
field
file buffer
file type
filter
filter program
focus
form
formal parameter
function
function header

G

global heap
global variable
glyph
grandchild

H

IJK

L

M

N

O

object file
object instance variable
object type
OLE
OLE container
OLE object
OLE server
ordinal
override
owner

P

parameter
parent
pixel
pointer
power set
primary index
private
private part
procedure
procedure header
program
project
project directory
project file
property
protected
protected block
public
published

Q

qualified identifier
qualified method identifier
qualifier
query

R

raise
real
record
record type
recursion
relational database
report
root class
routine
row
run time
run-time error
run-time library
run-time only

S

T

U

V

W

XYZ

abstract
A method that is declared but not implemented. Descendant types must override the abstract method.

accelerator key

Accelerator keys enable the user to access a menu command or component from the keyboard, by pressing Alt+ the appropriate letter, indicated in your code by the preceding ampersand. The letter after the ampersand appears underlined in the menu or component caption.

actual parameter
A variable, expression, or constant that is substituted for a formal parameter in a procedure or function call.

actual variable
A variable that a program can use at run time, as distinguished from the definition of that variable within the program. A location in memory used for storage purposes, as distinguished from an identifier.

alias
A name that specifies the location of database tables. If the database is on a server, an alias also specifies connection parameters for the server.

ancestor
An object from which another object is derived. An ancestor class can be a parent or a grandparent. See default ancestor.

application

An application is the executable file and all related files that a program needs to function which serve a common purpose or purposes, as distinguished from the design and source code of the project. Often used synonymously with 'program'. Compare with program and project.

array
A group of data elements identical in type that are arranged in a single data structure and are randomly accessible through an index.

ASCII

An acronym for "American Standard Code for Information Interchange" and used to describe the byte values assigned to specific characters. Examples: The capital letter A has an ASCII value of 65. The ASCII code for a space is 32.

In Pascal, you can reference a character by its ASCII code prefixed with a number sign (#). Example: To put the symbol for American cents into a character C, for example, you could code "c := #155;".

base type
The type referred to in a pointer declaration, an array declaration, or the enumeration type used in a set declaration. A type declaration builds a new type by combining or referencing one or more other base types, which could themselves be arbitrarily complex.

batch operation
Operations that you perform with the TBatchMove component on groups of records, or on datasets, to add, delete, or copy groups of records in a single operation.

BDE

Borland Database Engine; also referred to in some documentation as IDAPI. Delphi uses this database engine to access and deliver data. BDE maintains information about your PC's environment in the BDE configuration file (usually called IDAPI.CFG). Use the BDE Configuration Utility to change the settings in this configuration file.

BDE Configuration Utility

A program that enables you to change the settings in the BDE configuration file, usually called IDAPI.CFG. The executable file is named BDECFG.EXE.

BLOB
Binary large object. Many database tables use specific field types to contain BLOB data. Delphi lets you access BLOB data that exists as plain text with the TDBMemo component, and BLOB data that exists as a graphic with the TDBImage component.

block

The associated declaration and statement parts of a program or subprogram. Examples: In the var block of the routine declare an integer variable. Follow the **then** of your **if..then** statement with a **begin** to start a block of code that will be executed only if the condition is met.

Boolean
A data type that can have a value of either True or False. Data size = byte.

Borland Database Engine
See BDE.

breakpoints
A location you mark in your program where you want the program to pause during a debugging session. Once the program's execution has been paused, you can examine the state of your program at that point in its execution. The state of your program includes the values of variables and data structure elements and the routines on the call stack.

byte
A data type capable of holding a value from 0 to 255. A sequence of 8 bits.

callback routines
Routines in your application that are called by Windows and not by your application. For example, EnumFonts is a Windows routine that will call the given callback function for every font installed in the system.

call stack

The list of calls that were made to reach the present location, and which consequently show the path by which the program must return. Available during debugging.

canvas
The graphical drawing surface of an object. The canvas has a brush, a pen, a font, and an array of pixels. The canvas encapsulates the Windows device context.

case variant

1. The element of a case statement that is examined to determine what code will be executed. In a case statement beginning "Case I of", I is the case variant.
2. In record type definitions, case variants allow instances of that record to treat the same area of memory as different fields.

char
A Pascal type that represents a single character.

child

1. A child class is any class that is descended from another. For example, in "type B = class(A)", B is a child of A. Compare with grandchild.
2. The child of a window appears inside that window and cannot draw outside of its bounds. This is called a child or child window.

class

A list of features representing data and associated code assembled into single entity. A class includes not only features listed in its definition but also features inherited from ancestors. The term "class" is interchangeable with the term "object type."

class method

Class methods provide behavior for a class that is global in nature, or otherwise does not require instance data. A class method is called by using the class name followed by the method (TClass.SomeMethod) and can be called with an instance or without. As such, a class method cannot rely on any properties, fields or instance methods in its executions.

client

Generically, any thing that requests the services of something else. In Object Pascal, a client is any code that uses one or more features of an object or unit. In Windows, a client is code that makes use of the Windows API.

In database systems, a workstation connected to an intelligent "back-end" server from which it can request data. The client workstation can process the data locally and write it back to the server.

client area

In Windows, the area of a control which a program (that is, a client of Windows' services) is allowed to draw on. On a window, for example, that would usually exclude the frame and title bar.

column
The vertical component of a table, sometimes called a field. A column contains one value for each row in a table. See also row.

compile

The act of translating a block of source code into machine instructions. (As opposed to "interpret" which is the line-by-line translation of source code to machine instructions.)

Also see linking.

compiler directive
An instruction to the compiler that is embedded within the program; for example, {$R+} turns on range checking.

compile time
The period of time when the compiler is actively compiling source code.

compile-time error
An error detected by the compiler during compilation, such as a syntax error or unknown identifier.

complete evaluation

Every operand in a Boolean expression built from the **and** and **or** operators evaluates, even if the expression result can be determined before the entire expression is evaluated. This is useful when operands are routines that can alter the meaning of a program. Opposite of short-circuit Boolean evaluation.

component

1. The elements of a Delphi application, iconized on the Component palette. Components, including forms, are objects you can manipulate. Forms are components that can contain other components (forms are not iconized on the Component palette).
2. In Delphi, any class descended from TComponent is, itself, a component. In the broader sense, a component is any class that can be interacted with directly through the Delphi Form Designer. A component should be self-contained and provide access to its features through properties.
3. In traditional Pascal, the word "component" is also sometimes used synonymously with feature, as in "The record consists of several components: three string fields and two byte fields."

conditional symbol

Used with conditional compiler directives to specify a condition that is either true or false. You define (set to true) or undefine (set to false) conditional symbols with the $DEFINE and $UNDEF directives.

constant
An identifier with a fixed value in a program. At compile time, all instances of a constant in source code are replaced by the fixed value. Contrast with typed constant.

constant address expression

An expression that takes the address, the offset, or the segment of a global variable, a typed constant, a procedure, or a function.

Constant address expressions cannot reference local variables (stack-based) or dynamic (heap-based) variables, because their addresses cannot be computed at compile time.

const parameter

A const (constant) parameter is one that is passed by reference but that cannot be changed by the procedure. Const is more efficient in performance and memory usage a than a value parameter. See value parameter and variable parameter.

container application
An application that contains an embedded OLE object. (See OLE.)

container component
Any of several component classes that have the inherent ability to contain other components. Examples include TForm, TPanel, TNotebook, and TGroupbox. A container component is the parent of the components it contains.

control
A visual component. Specifically, any descendant of TControl.

data

1. Information stored in a database. Data may be a single item in a field, a record that consists of a series of fields, or a set of records. Delphi applications can retrieve, add, modify, or delete data in a database.
2. Generally, any information that has intrinsic value regardless of the means used to access it.

data access component
A Delphi component that enables you to connect to a database and access its data. Data access components are visible on a form only at design time, not at run time.

data-aware
Able to display and update data stored in an underlying table. All Delphi data control components are data-aware.

data control component

A Delphi component that enables you to create the interface of a database application. Many data controls are data-aware versions of component classes available on the Standard page of the Component palette.

data type
A fundamental unit of data definition that defines what kind of data can be stored in memory or in data tables, and what operations can be performed on that data.

database
A collection of data in tables.

database server
A system that manages relational databases. For example, SQL Server is a type of database server.

dataset
A collection of data determined by a TTable, TQuery, or TStoredProc component. A dataset defined by TTable includes every row in a table, and a dataset defined by a TQuery contains a selection of rows and columns from a table.

DDE client

In a DDE conversation, the client is the application that requests data. The DDE client is often called the destination.

DDE conversation

A link between a DDE client application and a DDE server application which provides a means for both applications to continuously and automatically send data back and forth.

DDE server

In a DDE conversation, the server is the application that updates the DDE client. The DDE server is often called the source.

default ancestor
The ancestor of any class that does not specify an ancestor: TObject.

default event

For a given component, the event whose event handler is automatically generated or displayed in the unit source code when you double-click the component at design time. For example, the OnClick event is the default event for a Button component.

default new form

The Form Template that is used to create a new form in the IDE at design time when you choose File | New Form. In a new installation, the Blank Form template is used. You can change the specified Form Template in the Environment Options dialog box (Options | Environment | Gallery).

default new project

The Project Template that is used to open a new Delphi project in the IDE at design time when you choose File | New Project. In a new installation, the Blank Project template is used. You can change the specified Project Template in the Environment Options dialog box (Options | Environment | Gallery).

derive
To create a new class based on an existing class. The new class inherits all of the features of the existing class, which is called its parent or, more generically, an ancestor.

descend

To acquire, in the process of being created, all the characteristics of another class. A class that descends from another is a descendant of the parent class. The process of creating a descendant class is deriving.

See also ancestor, derive, descendant, inheritance, and parent.

descendant
An object derived from another object. A descendant is type compatible with all of its ancestors.

design time
Period when you can use Delphi to design your application, using the form, the Object Inspector, Component palette, Code Editor, and so forth; as opposed to run time, when the application you design is actually running.

detail table

In multi-table relationships, the table whose records are subordinate to those of the master table. In a data model, the detail table is the one being pointed to by another table. For example, in the following data model, all of the tables except CUSTOMER.DB are detail tables.

dispatch
The means of resolving calls to object methods. Dispatching can be either static, virtual, or dynamic.
Do not confuse with TObject.Dispatch which dispatches message procedure calls, not virtuals.

drag

To move an object from one location to another by using your mouse.

To drag an object, click it and continue to hold down the left mouse button while you move the mouse pointer to a new location on your screen. When you are satisfied with the new location, release the mouse button.

dynamic
A form of virtual method which is more space efficient (but less speed efficient) than simple virtual.

dynamic data exchange (DDE)
The process of sending data to and receiving data from other applications through a predefined message protocol. You can use this to exchange data with other applications, or you can control other applications through the use of commands and macros.

dynamic-link library (DLL)
An executable module (extension .DLL) that contains code or resources that can be accessed by other DLLs or applications. In the Windows environment, DLLs permit multiple applications to share code and resources.

embedding
The act of placing one thing within another. In Windows, specifically the capability of one application to provide some or all of the services of another application integrated with its own services. For example, a word processor might allow a spreadsheet to be embedded into a document, allowing the user to write text around the spreadsheet and perhaps even change the spreadsheet while still working in the word processor. See OLE Container.

encapsulate
To provide access to one or more features through an interface that protects clients from relying upon or having to know the inner details of the implementation.

enumerated data type
A user-defined ordinal type that consists of an ordered list of identifiers.

end user

A member of an application's intended audience and, by extension, everyone in that audience. Synonymous with user, but emphasizes the fact that the programmer is not the user.

In Delphi documentation end user refers to a user of an application you develop using Delphi unless otherwise noted.

exception
An event or condition that, if it occurs, breaks the normal flow of execution. Also, an exception is an object that contains information about what error occurred and where it happened.

exception handler
Code designed to resolve the situation in the run-time environment that raised the exception and/or to restore the environment to a stable state afterwards.

execution point

The execution point indicates the next line in your program that will be executed when you run your program through the integrated debugger. The execution point is indicated by highlighted line of code in the Code Editor.

expressions
Part of a statement that represents a value or can be used to calculate a value.

event

A user action, such as a button click, or a system occurrence such as a preset time interval, recognized by a component.

Each component has a list of specific events to which it can respond. Code that is executed when a particular event occurs is called an event handler.

event handler

A form method attached to an event. The event handler executes when that particular event occurs.

When you use the Object Inspector to attach code to a component event, Delphi generates a procedure header and a **begin..end** block for you. For example, this is the code Delphi generates for a button click event:

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

end;
```

The code you write inside the **begin..end** block executes whenever Button1 is clicked.

features
A generic term used to refer the fields of a record, the types, constants, variables and routines of a unit, and the fields, properties, and methods of a class.

field

1. One possible element of a structured data type (that is, a record or object), a field is an instance of a specific data type. (Compare with property.)
2. In database terminology, a column of information in a table. A collection of related fields makes up one record. See also record.

file buffer
An area of memory set aside to expedite the transfer of data to and from a file.

file type
A file type refers to the specific data type that a file holds.

filter
Anything used to check or alter data. For example, the file filter in the Save dialog box can be set to show only Pascal files.

filter program
A program that takes output from another program as input and produces an altered, reduced, or verified version of that output.

focus
The component or window that is active in a running application is said to have "focus." Any keyboard input the user enters is directed to that component or window.

formal parameter

An identifier in a procedure or function declaration heading that represents the arguments that will be passed to the subprogram when it is called.

See parameter name for information on a given parameter.

form

To an end user, a form is merely another window. In Delphi, a form is a window that receives components (placed by the programmer at design time, or created dynamically with code at run time), regardless of the intended run-time functionality of the window.

function
A subroutine that computes and returns a value.

function header

Text that gives the name of a routine followed by a list of formal parameters, followed by the function's return type. In a unit, a routine may have a header entered into the interface part, and then again in the implementation part. The second appearance of the header may be an exact duplicate of the header in the interface part, or may be only the name of the routine.

global heap

Memory available to all applications.

Although global memory blocks of any size can be allocated, the global heap is intended only for large memory blocks (256 bytes or more). Each global memory block carries an overhead of at least 20 bytes, and under the Windows standard and 386 enhanced modes, there is a system-wide limit of 8192 global memory blocks, only some of which are available to any given application.

Note: Delphi suballocates small allocations from large global memory blocks to reduce the liklihood of hitting the system limit. (See HeapLimit, HeapBlock.)

global variable

A variable used by a routine (or the main body of a program) that was not declared by that routine (or a **var** part of the main body) is considered a global variable by that code. A variable global to one part of a program may be inaccessible to another part of the same program, and hence considered local in that context.

glyph
A bitmap that displays on a BitBtn or SpeedButton component with the component's Glyph property.

grandchild
A class descended from another through one or more intermediate classes. Example: In the following type definition "type E = class(D)", E is the child of D. If D is descended from class C, then E is a grandchild of class C, as well as C's parent, C's parent's parent, and so on, until the root class is reached. C and its ancestors are E's grandparents.

grandparent
A class from which others are descended through one or more intermediate classes. See grandchild.

grid

1. The evenly spaced dots on the form that aid in placing components during design time (not visible at run time). Control through Options | Environment | Preferences.
2. In database terminology, an object on a form that enables you to view and edit all the records in a dataset in a spreadsheet-like format. You create a grid with a TDBGrid component.

handling exceptions
Making a specific response to an exception, which then clears the error condition and destroys the exception object.

header

Text that gives the name of a routine followed by a list of formal parameters, followed in the case of a function by the function's return type. In a unit, a routine may have a header entered into the interface part, and then again in the implementation part. The second appearance of the header may be an exact duplicate of the header in the interface part, or may be only the name of the routine.

heap
An area of memory reserved for the dynamic allocation of variables.

heap suballocator

When allocating a memory large block, the heap manager simply allocates a global memory block using the Windows GlobalAlloc routine.

When allocating a small block, the Object Pascal heap manager allocates a larger global memory block and then divides (suballocates) that block into smaller blocks as required. Allocations of small blocks reuse all available suballocation space before the heap manager allocates a new global memory block, which, in turn, is further suballocated.

help context
A number assigned individually to the controls in a program so that when the user activates Help, the Help system can query the focused control and use the help context as a reference to supply information appropriate to what the user is doing.

hint
Pop-up text that appears when the mouse pointer passes over an object in the user interface at run time. Specified in the Hint property of many visual components.

host type
The particular server being used for a process or series of processes, hence "hosting" the activities.

IDAPI
See BDE.

identifier
A programmer-defined name for a specific item (a constant, type, variable, procedure, function, unit, program, or field).

implementation
The second, private part of a unit that defines how the elements in the **interface** part (the public portion) of the unit work.

include file (.INC)

An include file (.INC) is a source-code file that is included in a compilation using the {$I filename} compiler directive. Include files are seldom part of a Delphi project, but can optionally be used.

index

1. A position within a list of elements.
2. In database terminology, a sort order for a table associated with a specific field or fields, used to locate records quickly. An index performs the following tasks:

- Determines the location of records.
- Keeps records in sorted order.
- Speeds up search operations.

index type
Specifies the type of elements in an array. Valid index types are all the ordinal types except Longint and subranges of Longint.

inheritance
The assumption of the features of one class by another.

instance
A variable of an object type. More generally, a variable of any type. Actual memory is allocated.

integer
A numeric variable type that is a whole number in the range -32,768 to +32,767.

integrated debugger

The integrated debugger is contained within the Integrated Development Environment. This debugger lets you debug your source code without leaving Delphi. The functionality of this debugger can be reached through the Run and View menus.

interface
The first, public part of a unit that describes the constants, types, variables, procedures, and functions that are available within it.

key

A field or group of fields in a table, used to order records. A key has three effects:

- The table is prevented from containing duplicate records.
- The records are maintained in sorted order based on the key fields.
- A primary index is created for the table.

label
An identifier that marks the target for a **goto** statement.

language driver

Determines a table's sort order and available character set. The BDE Configuration Utility enables you to specify the default language driver for tables.

late binding
When the addess used to call virtual methods or dynamic methods is determined at run time.

linking
The process of turning compiled source code into an executable file. At the linking stage resources are bound into the executable.

literal value
A value that appears in the actual source code, such as the string "Hello, World" or the numeral 1 (as opposed to a calculated value or a declared constant).

local heap

Memory available only to your application or library.

It exists in the upper part of an application's or library's data segment.

The total size of local memory blocks that can be allocated on the local heap is 64K minus the size of the application's stack and static data. For this reason, the local heap is best suited for small memory blocks (256 bytes or less). The default size of the local heap is 8K, but you can change this with the **$M** compiler directive.

local symbol information

Information used by the IDE to debug a routine. Local symbol information must be enabled in the Project Options dialog box (Options | Project). Enabled by default in new Delphi installations.

local variable
A variable declared within a procedure or function.

lock
A device that prevents other users from viewing, changing, or locking a table while one user is working with it.

logic error

Logic errors occur when your program statements are valid, but the actions they perform are not the actions you intended. For example, logic errors occur when variables contain incorrect values, when graphic images don’t look right, or when the output of your program is incorrect.

Longint (type)

A 4-byte integer, able to store integers in the range -2,147,483,648 to +2,147,483,647.

lookup table

A secondary table that enables database systems to use a small code field to enable many records in a primary table to refer to information stored in the lookup table.

This can be used as a means of ensuring that values entered in a primary table are legitimate values, thus safeguarding data integrity.

loop
A statement or group of statements that repeat until a specific condition is met.

main form

At design time, the first form created in or added to a project. The form designated as the main form can be changed in the Project Options dialog box (Options | Project | Forms). The main form is usually the first displayed at run time, and usually the principal form displayed throughout the execution of the program.

master table

In a multi-table relationship, the primary table of your data model. If you have only one table in your data model, that table is the master table. In a multi-table data model, the master table is the one pointing to other tables. For example, in the following data model, all of the tables except VENDORS.DB are master tables.

method
Procedure or function associated with a particular object.

method identifier
The identifying string or dynamic index of a method.

method pointer
A pointer to a specific method in a specific object.

multiple document interface (MDI) application
An application whose interface consists of a main application window, called the frame window, that can contain multiple child windows, or documents. The child window's document title merges with the parent window's title bar when the child window is maximized.

modal
The run-time state of a form designed as a dialog box in which the user must close the form before continuing with the application. A modal dialog box restricts access to all other areas of the application. See Help for the ShowModal method for more information.

modeless
The run-time state of a form designed as a dialog box in which the user can switch focus away from the dialog box without first closing it. See Help for the Show method for more information.

module
A self-contained routine or group of routines. A unit is an example of a module.

nil

A pointer value referencing nothing. A pointer must be assigned a memory address in order to be meaningfully and safely used.

Note: Referencing a pointer having a nil value causes a General Protection Fault exception.

nonvisual component
A component that appears at design time as a small picture on the form, but either has no appearance at run time until it is called (like TSaveDialog) or simply has no appearance at all at run time (like TTimer).

nonwindowed control
A nonwindowed control is a control that cannot receive the focus, that cannot be the parent of other controls, and which does not have a window handle.

object files (.OBJ)

An intermediate machine-code file usually produced with an assembler. It is linked with a project or unit using the $L filename compiler directive.

Functions residing in .OBJ files are declared EXTERNAL in Pascal declarations. Object files (.OBJ) are seldom a part of a Delphi project.

object instance variable
The identifier created internally for an instance of an object.

object type
A class.

OLE

Object Linking and Embedding is a method for sharing complex data among applications. With OLE, data from a server application is stored in a container application. The data is stored in an OLE Object.

OLE container

An application that can contain an OLE object. In Delphi, an OLE container application has a TOLEContainer component.

OLE object

The data shared by an OLE server and OLE container. An OLE object can be linked or embedded in the container application. The data for linked objects are stored in an external file; embedded objects are stored in the container application.

Examples of OLE objects are documents, spreadsheets, pictures, and sounds.

OLE server
An application that can create and edit an OLE object.

ordinal (type)
Any Object Pascal type consisting of a closed set of ordered elements.

override
Redefine an object method in a descendant object type.

owner
An object responsible for freeing the resources used by other (owned) objects.

parameter
A variable or value that is passed to a function or procedure.

parent

1. The immediate ancestor of a class, as seen in its declaration. Example: In "type B = class(A)", class A is the parent of class B.
2. Parent property: the component that provides the context within which a component is displayed.

pixel
Any of the individual colored dots that make up an image on the screen. Derived from the words "picture element."

pointer
A variable that contains the address of a specific memory location.

power set
The set of all possible subsets of values of the base type including the empty set.

primary index

An index on the key fields of a table. An index performs the following tasks:

- Determines the location of records.
- Keeps records in sorted order.
- Speeds up search operations.

A primary index typically has a requirement of uniqueness--that is, no duplicate keys can exist.

private
The keyword indicating the beginning of a class declaration.

private part

Elements declared in this part of a class declaration can be used exclusively within the module that contains the class declaration. Outside that module they are unknown and inaccessible.

procedure
A subprogram that can be called from various parts of a larger program. Unlike a function, a procedure returns no value.

procedure declaration
The procedure declaration is the first occurrence of the procedure header that appears in a unit or project.

procedure header

Text that gives the name of a routine followed by a list of formal parameters. In a unit, a routine may have a header declared in the interface part, and then again in the implementation part. The second appearance of the header may be an exact duplicate of the header in the interface part, or may be only the name of the routine.

program
An executable file. Less formally, a program and all the files it needs to run. Contrast with application.

project
The complete catalogue of files and resources used in building an application or DLL. More specifically, the main source code file of the programming effort, which lists the units that the application or DLL depends on.

project directory
The directory in which the project file resides.

project file

The file that contains the source code for a Delphi project. This file has a .DPR extension. It lists all the unit files used by the project and contains the code to launch the application.

property
A feature that provides controlled access to methods or fields of an object. A published property may also be stored to a file.

protected
Used in class type definitions to make features visible only to the defining class and its descendants.

protected block
The **try** block of a **try...except** or **try...finally** statement.

public
Used in class type definitions to make features visible to clients of that class.

published
Used to make features in class type definitions streamable. Streamable features are visible at design time.

qualified identifier

An identifier that contains a period, that is, includes a qualifier. A qualified identifier forces a particular feature (of an object, record or unit) to be used regardless of other features of the same name that may also be visible within the current scope.

qualified method identifier

An object-type identifier, followed by a period (.), followed by a method identifier. Like any other identifier, you can prefix a qualified method identifier with a unit identifier and a period.

qualifier
An identifier, followed by a period (.), that precedes a method or other identifier to specify a particular symbol reference.

query
A way to retrieve data from your tables. A query can examine the data in a single table or in several tables.

raise

Raising an exception means constructing an exception object to signal an error or other exception condition. The application then must handle the exception.

real
A number represented by floating-point or scientific notation.

record

1. An instance of a record type.
2. In database terminology, a horizontal row in a table that contains a group of related fields of data.

record type
A structured data type that consists of one or more fields.

recursion
A programming technique in which a subroutine calls itself. Use care to ensure that a recursion eventually exits. Otherwise, an infinite recursion will cause a stack fault.

relational database

A database management model in which data is stored as rows (records) and columns (fields), and in which the data in one table can access the data in other tables by means of a common data field. The database structure can be used to create one-to-many and many-to-one relationships with data elements.

report
Organized summary or detail information that is presented to the end user either as a printed document or an online display.

root class
A class that itself has no ancestors, and from which all other classes are descended. In Delphi, the root class is TObject.

routine
A procedure or function.

row
The horizontal component of a table, sometimes called a record. A row contains one value for each column in a related group of columns in a table. See also column.

run time
Period when the application you design is running.

run-time error
An error that occurs when the application runs, as opposed to a compile-time error.

run-time library
The standard procedures and functions available to all Object Pascal programs.

run-time only
Routines, properties, events, or components that can be be modified, called, or seen only while your application is running (as opposed to design time).

scalar type
Any Object Pascal type consisting of ordered components.

scope
The visibility of an identifier to code within a program or unit.

separator
A blank (space) or a comment. Object Pascal treats a comment as a space.

separator bar
A line inserted between menu items. A dash character (-) entered in the Caption property of a new item in the menu designer creates a separator bar at the current position.

set
A collection of zero or more elements of a certain scalar or subrange base type

short-circuit evaluation
Strict left-to-right evaluation of a Boolean expression where evaluation stops as soon as the result of the entire expression is evident. This model guarantees minimum code execution time, and usually minimum code size.

Shortint
A one byte type capable of holding any whole number value from -128 to +127.

sizing handles
The small black rectangles that appear on the perimeter of a component, form or window when selected. You drag them to resize the object.

source code
The line-by-line statements written by the developer of a computer program using an appropriate editing tool and following the syntax rules for a particular programming language.

splash screen
A form you design to "introduce" your application, and which appears immediately at run time while the application main form and secondary forms are being loaded in memory, or while a database server connection is being established. See also main form.

SpeedMenu
A local menu on an object which you can access by right-clicking with a pointing device.

SQL
Structured Query Language, abbreviated SQL and commonly pronounced "sequel." A relational database language used to define, manipulate, search, and retrieve data in databases.

SQL table
A table on a database server defined by SQL.

stack
An area of memory reserved for storing local variables. Also keeps track of program execution and subroutine calls.

statement
The simplest unit in a program; statements are separated by semicolons.

static
Resolved at compile time, as are calls to procedures and methods.

step over
A debugger command that executes a program one line at a time, stepping over procedures while executing them as a single unit. Contrast with trace into.

string
A sequence of characters that can be treated as a single unit of data.

string list

A flexible collection of strings and (potentially) the objects associated with them.

subrange
Any specified contiguous portion of a scalar type.

switch directive

A compiler directive that turns compiler features on or off depending on the state (+ or -) of the switch. For example, {F+} turns the Force Far calls directive on; {F-} turns it off.

symbol
Any identifier. Symbols include reserved words.

table
A structure made up of rows (records) and columns (fields) that contains data.

tag field
A Longint storage for a specific instance of a component to be used as wanted by the programmer.

TDBDataset

A descendant of TDataset that includes the functionality needed to connect to a database, handle passwords, and perform other tasks associated with database connectivity.

You cannot instantiate an object of TDataset directly; you instantiate TTable, TQuery, or another TDataset descendant.

template
A predesigned project or form that serves as a starting point for your application design.

trace into

A debugger command that executes a program one line at a time, tracing into procedures which were compiled with debug information and following the execution of each line. Contrast with step over.

typecasting

The forcing of the compiler to treat an expression of type X as though it were an expression of type Y.

Using AS to typecast object instances causes generation of code to validate the compatibility of the typecast at run time. Normal typecasts are evaluate at compile time and are not validated at run time.

type
A description of how data should be stored and accessed. Contrast with variable--the actual storage of the data.

type compatibility

An instance may be used in place of or assigned to another type it is said to be compatible with.

Integer types are all cross-compatible. A descendant class instance is type-compatible with a variable of an ancestor type. Sibling classes are not type-compatible, nor are ancestors type-compatible with their descendants.

typed constant
A variable that is given a default value upon startup of the application. All global variables occupy a constant space in memory.

type definition
The specification of a non-predefined type. Defines the set of values a variable can have and the operations that can be performed on it.

unit

A independently compileable code module consisting of a public part (the interface part) and a private part (the implementation part).

Every form in Delphi has an associated unit.

The source code of a unit is stored in a .PAS file. A unit is compiled into a binary symbol file with a .DCU extension. The link process combines .DCU files into a single .EXE or .DLL file.

untyped file
Low-level I/O (input/output) channels that let you directly access any disk file regardless of its internal format.

untyped pointer
A pointer that does not point to any specific type. An untyped pointer cannot be referenced without a typecast. (Also see typecast.)

unqualified identifier

An identifier that contains no periods, that is, an identifier with no qualifier. The semantics of an unqualified identifier depend on the current scope. Example: "Create" is an unqualified identifier that will call any routine called "Create" within the current scope (or cause a compile error if no such routine is visible) but "TForm.Create" will call the specific "Create" method which is a feature of TForm. See qualifier.

use count

An internal variable that Windows uses to determine whether or not a DLL should stay in memory. A DLL stays in memory while its use count is greater than zero.

Windows increments Use Count every time an application loads a DLL and decrements whenever an application frees the DLL.

user-defined
Said of a type that is defined by a programmer and not inherently part of the Pascal language. This includes any type definitions you may code or definitions provided by Borland, or any other source.

value parameter

A procedure or function parameter that is passed by value; that is, the value of a parameter is copied to the local memory used by the routine and therefore, changes made to that parameter are local.

See variable parameter, const parameter.

variable parameter

A subroutine parameter that is passed by reference. Changes made to a variable parameter remain in effect after the subroutine has ended. See value parameter, const parameter.

variable
An identifier that represents an address in memory, the contents of which can change at run time.

virtual
Calls are resolved at run time by name.

visual component
A component that is visible, or can be made visible on a form at run time.

watches

A watch expression lets you track the values of program variables or expressions as you step over or trace into your code. Use the Watch List window to view the currently set watches.

As you step through your program, the value of the watch expression will change if your program updates any of the variables contained in the watch expression.

window handle
A number assigned by Windows to a control that must be used to request services for that control from the Windows API.

windowed control
A control that can receive the focus, that can own other controls, and which does have a window handle.

word
In Delphi, A location in memory occupying 2 adjacent bytes; the storage required for a variable of type integer or word. Also, a predefined data type with a range of 0 to 65535.

wrapper
An object, routine, group of objects, or group of routines designed to encapsulate some functionality for the programmer usually for some perceived benefit. VCL is an object-oriented wrapper for the Windows API.

z-order
The conceptual distance of an object from the surface of the screen. Whether or not a control is covered by other controls depends on its z-order relative to those controls.

Responding to the mouse

See also Example

There are four kinds of mouse actions you can respond to in your applications.

Three of these are uniquely mouse actions:

- Mouse-button down
- Mouse move
- Mouse-button up

The fourth kind of action, a click (a complete press-and-release, all in one place) is a bit different, as it can also be generated by some kinds of keystrokes (such as pressing Enter in a modal dialog box).

Responding to a mouse-down action

Whenever the user presses a mouse button, an OnMouseDown event goes to the object the pointer is over. The object can then respond to the event.

To respond to a mouse-down action,

- Define the object handler for an OnMouseDown event.

Responding to a mouse-up action

Whenever the user releases a mouse button, an OnMouseUp event occurs. The event usually goes to the object the mouse cursor is over when the user presses the button, which is not necessarily the same object the cursor is over when the button is released. This enables you, for example, to draw a line as if it extended beyond the border of the form.

To respond to mouse-up actions,

- Define the form's handler for the OnMouseUp event.

Responding to a mouse-move action

An OnMouseMove event occurs periodically when the user moves the mouse. The event goes to the object that was under the mouse pointer when the user pressed the button.

To respond to mouse movements,

- Define the form's handler for the OnMouseMove event.

Mouse-move events occur even when the user has not pressed the mouse button. You need to write code so the application will act on mouse-move events only when the mouse button is down.

To track whether there is a mouse button pressed,

- You need to add an object field to the form object.

See also

Generating a new event handler

What is in a mouse event?

Example

Example for responding to mouse-down event
Example for repsonding to mouse-down event
Example for responding to mouse-down event

Example

The following example displays the string 'Here!' at the location on a form clicked with the mouse:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.TextOut(X, Y, 'Here!'); { write text at (X, Y) }
end;
```

Example

The following example draws a line to the point of the mouse-button release:

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);      { draw line from PenPos to (X, Y) }
end;
```

Example

The following example draws a line on a form to the location of the OnMouseMove event:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);      { draw line to current position }
end;
```

What is in a mouse event?

See also

Three mouse events are defined in Delphi:

- OnMouseDown
- OnMouseMove
- OnMouseUp

When a Delphi application detects a mouse action, it calls whatever event handler you have defined for the corresponding event, passing five parameters. Use the information in those parameters to customize your responses to the events. The five parameters are as follows:

Parameter	Meaning
Sender	The object that detected the mouse action
Button	Indicates which button was involved: mbLeft, mbMiddle, or mbRight.
Shift	Indicates the state of the Alt, Ctrl, and Shift keys at the time of the mouse action
X, Y	The coordinates where the event occurred

Most of the time, the most important information in a mouse-event handler is the coordinates, but sometimes you also need to check Button to determine which mouse button caused the event.

Note: Delphi uses the same criteria as Microsoft Windows in determining which mouse button has been pressed. Thus, if you have switched the default "primary" and "secondary" mouse buttons (so that the right mouse button is now the primary button), clicking the primary (right) button will record mbLeft as the value of the Button parameter.

See also

Responding to the mouse

Adding a field to a form object

See also Example

When you add a component to a form, Delphi adds a field that represents that component to the form object, and you can refer to the component by the name of its field. You can add your own fields to forms by editing the type declaration at the top of the form's unit.

Adding your own fields provides you with a way to declare variables that are global to all the event handlers associated with the declaring form.

To add a field to an object,

- Edit the object's type definition by specifying the field identifier and type after the **public** directive at the bottom of the declaration.

Delphi puts all fields that represent components and all methods that respond to events before the **public** directive.

See also

Adding a method to a form object

Modifying the form's type declaration

Example

The following example adds a field called Drawing of type Boolean to the declaration of Form1.

```
type
  TForm1 = class(TForm)
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure FormMouseMove(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  public
    Drawing: Boolean;{ field to track whether button was pressed }
  end;
```

Creating a tool bar

See also

A tool bar is a panel, usually across the top of a form, below the menu bar, that can hold any number of components, especially buttons. Tool bars provide an easy, visible way to present options to users of your applications.

To add a tool bar to a form,

1. Add a Panel component to the form.
2. Set the panel's Align property to alTop. When aligned to the top of the form, the panel maintains its height, but matches its width to the full width of the form's client area, even if the window changes size.
3. Add speed buttons or other components to the bar.

You can add as many tool bars to a form as you want. If you have multiple panels aligned to the top of the form, they "stack" vertically in the order added.

See also

Adding hidden tool bars
Hiding and showing tool bars
TPanel

Adding a speed button to a tool bar

See also

Speed buttons are graphical buttons on tool bars used to represent shortcuts for menu commands and macros.

When you add a speed button to a panel, the panel becomes the container of the speed button, so moving or hiding the panel also moves or hides the speed button.

To add a speed button to a tool bar,

- Place the speed button component on the panel that represents the tool bar.

The default height of the tool bar is 41, and the default height of speed buttons is 25. If you set the Top property of each button to 8, they will be vertically centered. The default grid setting will snap the speed button to that vertical position for you.

See also

Assigning a glyph to a speed button
Creating a group of speed buttons
Setting the initial condition of a speed button
Using the speed button component

Assigning a glyph to a speed button

See also

Each speed button needs a graphical image called a glyph to indicate to the user what the button does. If you supply the speed button only one image, the button manipulates that image to indicate whether the button is pressed, unpressed, selected, or disabled. You can also supply separate, specific images for each state if you prefer. See Specifying different glyphs for the different button states.

You normally assign speed-button glyphs at design time, although you can assign different glyphs at run time.

To assign a glyph to a speed button,

1. Select the speed button.
2. In the Object Inspector, select the Glyph property.
3. Double-click the Value column beside Glyph.

   Delphi opens the Picture editor.
4. Click Load in the Picture editor.

   Delphi opens a file open dialog box.
5. Select the .BMP file that contains the image you want on the button, and click OK.

   The file open dialog box closes, and the Picture editor displays the selected bitmap image.
6. Click OK to close the Picture editor.

See also

Adding a speed button to a tool bar
Creating a group of speed buttons
Setting the initial condition of a speed button

Setting the initial condition of a speed button

See also

The appearance of a speed button gives the user information about its state and purpose. Because speed buttons have no caption, it is important that you use the right visual cues to assist users. Part of this involves specifying a glyph for the speed button.

When the user chooses a speed button at run time, by default, the button appears to be pressed down, and then returns to its raised position when the user releases it. You can specify that a speed button remain pressed (in the "down" position) when the user chooses it, by making it into a toggle or by grouping it. A toggle button switches between a pressed and a raised position each time the user chooses it.

To make a non-grouped speed button a toggle,

1. Set its AllowAllUp property to True.
2. Set its GroupIndex property to a non-zero value.

You can specify that a speed button toggle appear pressed when it first is displayed in the application, for example to indicate that this is the default selection in a group of buttons.

To make a speed button initially appear in the pressed state,

- Set its Down property to True.

Just as you can for any component, you can specify that a speed button is not available, initially or at any time while the application is running, by disabling it.

To make a speed button appear disabled,

- Set its Enabled property to False.

Note: Setting Enabled to False automatically dims any bitmap displayed on the speed button. If the speed button does not contain a graphic, setting Enabled to False does not provide any visual cue to the user that the button is not available. However, when the user chooses it at run time, the button will not respond.

See also

Adding a speed button to a tool bar
Assigning glyph to a speed button
Creating a group of speed buttons

Creating a group of speed buttons

See also

Creating a group of speed buttons does not require anything other than associating them visually in the form or within a container component. You can, however, create a group of speed buttons whose behavior at run time is interdependent.

When the user chooses a speed button at run time, by default, the button appears to be pressed down, and then returns to its raised position when the user releases it. By grouping speed buttons, you can specify that a speed button in the group remain pressed (in the "down" position) after the user chooses it.

Note: You can also specify that a non-grouped speed button remain pressed, by making it into a toggle.

If you group speed buttons, you can specify that one speed button in the group must always be pressed at any given time. Alternatively, you can allow all the buttons in the group to be raised.

To associate any number of speed buttons into a group,

- Select all the buttons you want in the group, and with the whole group selected, set the GroupIndex property to a nonzero value. (Zero is the default GroupIndex value, and therefore does not create a grouped relationship between speed buttons.)

  You can add a speed button to an existing group at any time, by setting its GroupIndex property to the same value as the GroupIndex property of the group. All speed buttons with the same GroupIndex value on the form, or within the same container on the form will interact as a group at run time.

To allow all buttons in a group to appear raised,

- Set the AllowAllUp property to True for any speed button in the group.

  When you set AllowAllUp to True for a speed button in a group, Delphi automatically sets AllowAllUp to True for all other speed buttons in the same group.

See also

Adding a speed button to a tool bar
Assigning glyph to a speed button
Grouping speed buttons within container components
Setting the initial condition of a speed button

Grouping speed buttons within container components

See also

Placing components you wish to group inside a container component (such as a panel,) makes working with them as a group at design time -- for example, moving, copying, deleting -- easier. You can also use properties of the container component to create visual effects such as raised or lowered borders.

Speed buttons do not need to be inside a container to act as a group: Setting the GroupIndex property is what determines which buttons interact together at run time. You can create several groups of speed buttons by setting several different sets of nonzero GroupIndex values. All speed buttons with the same GroupIndex value on the form, or within the same container on the form will interact as a group at run time.

See also

Adding a speed button to a tool bar
Adding a tool bar to a form
Assigning glyph to a speed button
Creating a group of speed buttons
Setting the initial condition of a speed button

Responding to a button click

See also

Button components have two possible click events:

- OnClick
- OnDblClick

To respond to the user's click on a button,

- Define a handler for the button's OnClick event. If you are not sure how to generate an event handler, see Generating a New Event Handler.

By default, Delphi generates an OnClick event handler named after the component clicked. For example, the default OnClick event handler for a speed button named LineButton would be:

```
procedure TForm1.LineButtonClick(Sender: TObject);
begin

end;
```

You then fill in the action that you want to occur when the user clicks that button.

See also

Handling keyboard events
Handling mouse events
Handling drag and drop events

Adding a method to a form object

See also Example

Any time you find that a number of your event handlers use the same code, you can make your application more efficient by moving the repeated code into a method that all the event handlers can share.

To add a method to a form,

1. Add the method declaration to the form object.

   You can add the declaration in either the **public** or **private** parts at the end of the form object's declaration. If the code is just sharing the details of handling some events, it is probably safest to make the shared method **private**.

2. Write the method implementation in the **implementation** part of the form unit.

   The header for the method implementation must match the declaration exactly, with the same parameters in the same order.

See also

Adding a field to a form object

Modifying the form's type declaration

Example

The following example eliminates repetitive shape-drawing code from mouse-event handlers by adding a method to the form called DrawShape and calling it from each of the handlers.

1. Add the declaration of DrawShape to the form object's declaration.

```
type
  TForm1 = class(TForm)
    ...  { many fields and methods omitted for brevity }
  public
    { Public declarations }
    procedure DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
  end;
```

2. Write the implementation of DrawShape in the **implementation** part of the unit.

```
implementation
{$R *.DFM}
... { many other method implementations omitted for brevity }
procedure TForm1.DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
begin
  with Canvas do
  begin
    Pen.Mode := AMode;
    case DrawingTool of
      dtLine:
        begin
          MoveTo(TopLeft.X, TopLeft.Y);
          LineTo(BottomRight.X, BottomRight.Y);
        end;
      dtRectangle: Rectangle(TopLeft.X, TopLeft.Y, BottomRight.X,
BottomRight.Y);
      dtEllipse: Ellipse(TopLeft.X, TopLeft.Y, BottomRight.X,
BottomRight.Y);
      dtRoundRect: RoundRect(TopLeft.X, TopLeft.Y, BottomRight.X,
BottomRight.Y,
        (TopLeft.X - BottomRight.X) div 2, (TopLeft.Y - BottomRight.Y) div
2);
    end;
  end;
end;
```

3. Modify the other event handlers to call DrawShape.

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  DrawShape(Origin, Point(X, Y), pmCopy);    { draw the final shape }
  Drawing := False;
end;

procedure TForm1.FormMouseMove(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    DrawShape(Origin, MovePt, pmNotXor);     { erase the previous shape }
    DrawShape(Origin, Point(X, Y), pmNotXor);{ draw the current shape }
  end;
end;
```

Adding hidden tool bars

See also

Tool bars do not have to be visible all the time. In fact, it is often convenient to be able to have a number of tool bars available, but show them only when the user wants to use them. Often you create a form that has several tool bars, but hide some or all of them.

To create a hidden tool bar,

1. Add a tool bar to the form. (Be sure to set Align to alTop.)
2. Set the panel's Visible property to False.

Although the tool bar remains visible at design time so you can modify it, it remains hidden at run time until the application specifically makes it visible.

See also

Hiding and showing tool bars

Hiding and showing tool bars

See also

Often, you want an application to have multiple tool bars, but you do not want to clutter the form with all of them at once. Or some users might not want tool bars at all. As with all components, tool bars are shown or hidden at run time as specified by your code.

To hide or show a tool bar at run time,

- You set its Visible property to False or True, respectively, usually in response to particular user events or changes in the operating mode of the application.

A common practice is to put some kind of close button on each tool bar. When the user clicks that button, the application hides the corresponding tool bar.

See also
Adding hidden tool bars

Adding a status line

See also

A status line is much like a tool bar, except it usually sits at the bottom of a window instead of the top, and instead of having components on it, the status line usually displays text. Delphi's Panel component makes an excellent status line.

Often you will want to subdivide the status line into multiple status areas and provide each with a three-dimensional bevel effect. To do this, you can use panel components nested within the status line panel.

Adding a status bar to an application involves the following tasks:

- Placing a status-line panel
- Subdividing a panel
- Creating 3-D panels
- Updating the status line

See also
Adding a tool bar

Placing a status-line panel

See also

In general, status lines appear at the bottom of a form. Aligning the status-line panel to the bottom of the form takes care of both placement and resizing for you.

To add a status-line panel to a form,

1. Place a panel component on the form.
2. Set the panel's Align property to alBottom.
3. Clear the panel's caption.

Once you have added the status-line panel, you can subdivide it into separate status panels. If you are not going to subdivide, you probably want to set the BevelInner and BorderWidth properties to create a 3-D effect. Panels that serve only as containers for smaller panels generally do not change those properties.

You can also align the text within a panel. By default, panels center their captions, but often in a status line, you want to set Align to alLeft.

See also

Subdividing a panel
Creating 3-D panels
Updating the status line

Subdividing a panel

See also

Often you want to divide a panel, particularly one used for a status line, into multiple, independent areas. Although you can achieve a similar effect by carefully formatting the text in a single panel, it is more efficient to use individual panels instead.

To create panels within another panel,

1. Place a new panel within the panel.
2. Set any 3-D effects upi want for the new panel.
3. Set the Align property of the new panel to alLeft.
4. Move the right side of the new panel to adjust its width.
5. Clear the new panel's caption.
6. Repeat steps 1 to 5 as needed for additional panels.

As you add new left-aligned panels to the original panel, they align to each others' right sides. For the last panel, you probably want to set Align to alClient, rather than alLeft, so that the last panel takes up all remaining space in the original panel.

See also

Placing a status-line panel
Creating 3-D panels
Updating the status line

Creating 3-D panels

See also

Panel components used for status-line information usually have a 3-D effect. By default, all panels have an outer bevel to give them a slightly raised appearance. Interior text, however, usually looks better if it is set off by lowering, making the panel look like a frame around the text.

To create the lowered-text effect, you change two properties: BevelInner and BorderWidth.

- To create the "engraved" look for the panel, set BevelInner to bvLowered.
- To change the space between the inner and outer bevels, change the BorderWidth property. A BorderWidth of 2 gives a good appearance.

The combination of inner and outer bevels creates a frame around the text.

See also

Placing a status-line panel
Subdividing a panel
Updating the status line

Updating the status line

See also Example

Once you have a status line in a form, you need to update the status-line information. You can set the caption text of a panel at any time to display information you want to provide the user, such as the location of the cursor in a graphics application, or the selected cell in a spreadsheet application.

To update a status-line panel,

- Set the panel's Caption property to display the information you want to provide to the user.

Add this code to any event handlers that affect the status a particular panel reflects.

See also

Placing a status-line panel
Subdividing a panel
Creating 3-D panels
OnHintEven

Example

The basic technique for updating a working status line involves the following steps:

1. Add a method identifier to the **public** section of your main form's declaration:

```
procedure ShowHint(Sender: TObject);
```

2. In the event handler for the main form's OnCreate event, code the following assignment statement:

```
Application.OnHint := ShowHint;
```

3. Write the code for the ShowHint method:

```
procedure TForm1.ShowHint(Sender: TObject);
begin
  Panel1.Caption := Application.Hint;   {Panel1 is status line}
end;
```

For more detailed information on creating a working status line, see the example for the OnHint event.

Drawing on a bitmap

See also

The times when you want to draw directly on a form are relatively rare. More often, an application should draw on a bitmap, since bitmaps are very flexible for operations such as copying, printing, and saving. Delphi's Image component can contain a bitmap, making it easy to put one or more bitmaps into a form.

There are two things you need to do to make the drawing code you write apply to the bitmap instead of the form:

- Use the Image component's Canvas instead of the form's canvas.
- Attach your event handlers to the appropriate events in the Image component.

Once you move the application's drawing to the bitmap in the Image component, it will be easy to add printing, Clipboard operations, and loading and saving bitmap files.

In addition, the bitmap need not be the same size as the form: It can be either smaller or larger. By adding a scroll-box component to the form and placing the image inside it, you can draw on bitmaps that are much larger than the form or even larger than the screen.

Adding a scrollable bitmap for drawing takes two steps:

- Adding a scrollable region
- Adding an image component

See also

Loading an image at design time

Providing an area for drawing at run time

Adding a scrollable region

See also

The are many times when an application needs to display more information than will fit in a particular area. Some components, such as list boxes and memos, can automatically scroll their contents. But other components, and sometimes even forms full of components, need to be able to scroll. Delphi provides a way to create such scrolling regions with the ScrollBox component.

A scroll box is much like a panel or a group box, in that it can contain other components. However, a scroll box is normally invisible unless it is needed. If the components contained in the scroll box cannot all fit in the visible area of the scroll box, it automatically displays one or two scroll bars, enabling users to move components outside the visible region into a position where they can be seen and used.

To create a scrolling region,

- Place a ScrollBox component on a form and set its boundaries to the region you want to scroll.

You often use the Align property of a scroll box to allow the scroll box to adjust its area to a form or a part of a form. For example, setting Align to alClient causes the scroll box to occupy the entire client area of the form.

See also

Drawing on a bitmap

Adding an image component to the form

Adding an Image component

See also

The Delphi Image component is a kind of place-holder component. It allows you to specify an area on a form that will contain a picture object, such as a bitmap or a metafile. You can either set the size of the image manually, or allow the Image component to adjust to the size of its picture at run time.

You can use an Image component to hold a bitmap that is not necessarily displayed all the time, or which an application needs to use to generate other pictures.

Placing the component

You can place an Image component anywhere on a form. If you set AutoSize to True, then you want to take into consideration the top left corner, as that remains stable even as the component resizes itself. If the Image component is a non-visible holder for a bitmap, you can place it anywhere, just as you would a non-visual component.

Setting the initial picture

If the Image component will always hold a particular picture, you can set its Picture property a4t design time. You can also load the picture into the component from a file at run time. Or, you can use the Image component to provide an area that the user can draw on, such as in a graphics application. To provide a blank bitmap for drawing, you should create it at run time. See Providing an Area for Drawing at Run Time

See also

Using the Image component

Loading an image

Providing an area for drawing at run time

See also Example

You can provide an area for the user to draw in at run time by using the Image component to contain a blank bitmap.

To create a blank bitmap when the application starts,

1. Attach a handler to the OnCreate event for the form that contains the Image component.
2. Create a bitmap object.
3. Assign it to the Image component's Picture.Graphic property.

Assigning the bitmap to the picture's Graphic property gives ownership of the bitmap to the picture object. It will therefore destroy the bitmap when it finishes with it, so you should not destroy the bitmap object. You can assign a different bitmap to the picture, at which point the picture disposes of the old bitmap and assumes control of the new one.

See also

Using the Image component

Loading a picture from a file

Replacing a picture

Example

The following code, attached to Form1's OnCreate event, creates a blank bitmap 200 pixels wide by 200 pixels tall, and places the blank bitmap into the image component on the form.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Bitmap: TBitmap;   { temporary variable to hold the bitmap }
begin
  Bitmap := TBitmap.Create;{ construct the bitmap object }
  Bitmap.Width := 200;     { assign the initial width... }
  Bitmap.Height := 200;    { ...and the initial height }
  Image.Picture.Graphic := Bitmap;     { assign the bitmap to the image
component }
end;
```

Printing graphics

See also Example

Printing graphic images from a Delphi application is a simple task. The only requirement for printing is that you add the Printers unit to the **uses** clause of the form that will call the printer. The Printers unit declares a printer object called Printer that has a canvas that represents the printed page.

To print a graphic image,

- Copy the image to the printer's canvas.

You can use the printer's canvas just as you would any other canvas. In particular, that means you can copy the contents of a graphic object, such as a bitmap, to the printer directly.

Example

The following code copies the image of a form to the printer in response to a click on a button named PrintButton:

```
procedure TForm1.PrintButtonClick(Sender: TObject);
begin
  with Printer do
  begin
    BeginDoc;  { start printing }
    Canvas.Draw(0, 0, Image);    { draw Image at top left corner of
printed page }
    EndDoc;    { finish printing }
  end;
end;
```

See also

Using the printer object

Printing the contents of a memo

Working with graphics files

Graphic images that exist only for the duration of one running of an application are of very limited value. Often, you either want to use the same picture every time, or you want to save a created picture for later use. Delphi's Image component makes it easy to load pictures from a file and save them again.

Choose a topic for more information.

- Loading a picture from a file
- Saving a picture to a file
- Replacing a picture

Loading a picture from a file

See also Example

The ability to load a picture from a file is important if your application needs to modify the picture or if you want to store the picture outside the application so a person or another application can change the picture without changing code.

To load a graphics file into an Image component,

- Call the LoadFromFile method of the Image component's Picture object.

See also

Using the Image component

Example

The following code is an OnClick event handler for a menu item called Open. This code retrieves a file name from an open file dialog box and then loads that file into an Image component named Image:

```
procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    CurrentFile := OpenDialog1.FileName;
    Image.Picture.LoadFromFile(CurrentFile);
  end;
end;
```

Saving a picture to a file

See also Example

When you have created or modified a picture, you often want to save the picture in a file for later use. The Delphi Pictureobject can save graphics in several formats, and application developers can create and register their own graphic-file formats so that picture objects can store them as well.

To save the contents of an Image component to a file,

- Call the SaveToFile method of the Image component's Picture object.

The SaveToFile method requires the name of a file to save into. If the picture is newly created, it might not have a file name, or a user might want to save an existing picture in a different file. In either case, the application needs to get a file name from the user before saving, as shown in the example code.

See also

Using the Image component

Loading a picture from a file

Example

The following pair of event handlers, written for menu items called Save and Save As, respectively, handles resaving named files, saving unnamed files, and saving existing files under new names.

```
procedure TForm1.Save1Click(Sender: TObject);
begin
  if CurrentFile <> '' then
    Image.Picture.SaveToFile(CurrentFile)    { save if already named }
  else SaveAs1Click(Sender);     { otherwise get a name }
end;

procedure TForm1.Saveas1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then    { get a file name }
  begin
    CurrentFile := SaveDialog1.FileName;     { save the user-specified
name }
    Save1Click(Sender);    { then save normally }
  end;
end;
```

Replacing a picture

See also Example

You can replace the picture in an Image component at any time. If you assign a new graphic to a picture that already has a graphic, the new graphic replaces the existing one.

To replace the picture in an Image component,

- Assign a new graphic to the Image component's Picture object.

Assigning a new bitmap to the picture object's Graphic property causes the picture object to destroy the existing bitmap and take ownership of the new one. Delphi handles the details of freeing the resources associated with the previous bitmap automatically.

See also

Using the Image component
Providing an area for drawing at run time
Loading a picture from a file
Saving a picture to a file

Example

The following code is an OnClick event handler for a menu item called New. This code opens a dialog box, NewBMPForm, that enables the user to choose a size other than the default size used for the existing bitmap area.

```
procedure TForm1.New1Click(Sender: TObject);
var
  Bitmap: TBitmap;   { temporary variable for the new bitmap }
begin
  with NewBMPForm do
  begin
    ActiveControl := WidthEdit;  { make sure focus is on width field }
    WidthEdit.Text := IntToStr(Image.Picture.Graphic.Width);   { use
current dimensions... }
    HeightEdit.Text := IntToStr(Image.Picture.Graphic.Height); { ...as
default }
    if ShowModal <> idCancel then{ continue if user does not cancel dialog
box }
    begin
      Bitmap := TBitmap.Create;  { create fresh bitmap object }
      Bitmap.Width := StrToInt(WidthEdit.Text);    { use specified width }
      Bitmap.Height := StrToInt(HeightEdit.Text);  { use specified height }
      Image.Picture.Graphic := Bitmap; { replace graphic with new bitmap }
      CurrentFile := '';   { indicate unnamed file }
    end;
  end;
end;
```

Using the Clipboard with graphics

See also Example

You can use the Windows Clipboard to copy and paste graphics within your applications or to exchange graphics with other applications. Delphi's Clipboard object makes it easy to handle different kinds of information, including graphics.

Before you can use the Clipboard object in your application, you must add the ClipBrd unit to the **uses** clause of any unit that needs to access Clipboard data.

Copying graphics to the Clipboard

You can copy any graphical image, including the contents of Image components and graphics on forms, to the Clipboard. Once on the Clipboard, the picture is available to all Windows applications.

To copy a picture to the Clipboard,

- Assign the picture to the Clipboard object using the Assign method.

Cutting graphics to the Clipboard

Cutting a graphic to the Clipboard is exactly like copying it, but you also erase the graphic from the source.

To cut a graphic from a picture to the Clipboard,

- First copy it to the Clipboard, then erase the original. To erase the original, for example, you might set the area to white.

Pasting graphics from the Clipboard

If the Windows Clipboard contains a graphic, you can paste it into any image object, including Image components and the surface of a form.

To paste a graphic from the Clipboard,

1. Call the Clipboard's HasFormat method to see whether the Clipboard contains a graphic.

   HasFormat is a Boolean function. It returns True if the Clipboard contains an item of the type specified in the parameter. To test for graphics, you pass CF_BITMAP.

2. Assign the bitmap on the Clipboard to the destination, using the Assign method.

See also

Using the clipboard with text

Example

Example for copying graphics to the Clipboard
Example for cutting graphics to the Clipboard
Example for pasting graphics to the Clipboard

Example

This code copies the picture from an Image component named Image to the Clipboard in response to a click on an Edit|Copy menu item:

```
procedure TForm1.Copy1Click(Sender: TObject);
begin
  Clipboard.Assign(Image.Picture);
end;
```

Example

This example removes the selected graphic from the form and copies it onto the Clipboard when the user clicks the Edit|Cut menu item.

```
procedure TForm1.Cut1Click(Sender: TObject);
var
  ARect: TRect;
begin
  Copy1Click(Sender);{ copy picture to Clipboard }
  with Image.Canvas do
  begin
    CopyMode := cmWhiteness;     { copy everything as white }
    ARect := Rect(0, 0, Image.Width, Image.Height);{ get bitmap rectangle }
    CopyRect(ARect, Image.Canvas, ARect);    { copy bitmap over itself }
    CopyMode := cmSrcCopy; { restore normal mode }
  end;
end;
```

Example

The following code pastes a picture from the Clipboard into an Image component in response to a click on an Edit|Paste menu item:

```
procedure TForm1.PasteButtonClick(Sender: TObject);
var
  Bitmap: TBitmap;
begin
  if Clipboard.HasFormat(CF_BITMAP) then     { check to see if there is a
picture }
   begin
     Bitmap := TBitmap.Create;   {Create a bitmap to hold the contents of
the Clipboard}
     try
       Bitmap.Assign(Clipboard); {get the bitmap off the clipboard using
Assign}
       Image.Canvas.Draw(0, 0, Bitmap);{copy the bitmap to the Image}
     finally
       Bitmap.Free;
     end;
   end;
end;

end;
```

Drawing graphics at run time

Drawing graphics in a Delphi application is done on the object's canvas rather than directly on the object.

The canvas is a property of the object, and is itself an object with its own properties. The canvas object has four important properties:

- A pen for drawing lines
- A brush for filling shapes
- A font for writing text
- An array of pixels to represent the image

Canvases are available only at run time, so you do all your work with canvases by writing code.

To use graphics effectively at run time, you need to understand the following concepts:

Drawing vs. painting

Using the pixel array

Drawing lines and shapes

Drawing vs. painting

When working with graphics, you often encounter the terms drawing and painting. It is important that you understand the difference between the two.

Drawing

Drawing is the creation of a single, specific graphic element, such as a line or a shape, with code. In your code, you tell an object to draw a specific graphic in a specific place on its canvas by calling a drawing method of the canvas.

Painting

Painting is the creation of the entire appearance of an object. At certain times, Windows determines that objects on the screen need to refresh their appearance, so it generates OnPaint events. OnPaint events happen when, for example, a window that was hidden is brought to the front.

Painting usually involves drawing because in response to OnPaint events, an object generally draws some graphics.

An OnPaint event occurs whenever Windows determines that a form needs to be redrawn, such as when another window covering the form goes away. FormPaint is the default name Delphi generates for a handler for a form's OnPaint event.

You can handle OnPaint events when you want to paint a background on a form or when you want to regenerate an image.

Using the pixel array

See also Example

The Pixels property of a canvas represents the individual colored points that make up the image onscreen.

You rarely need to access Pixels directly, unless you need to find out what color a particular pixel is or to set that pixel's color.

Pixels are the drawing surface of the canvas, and the pen and brush are convenient ways to manipulate groups of pixels.

Manipulating pixels

The pixels of a canvas are a two-dimensional array, with each element in the array having a particular color. You can either read or set individual pixels.

The upper left corner of the canvas is the origin of the coordinate system.

To retrieve the color of a particular pixel,

- Read the Pixels property of the canvas, using as indexes the x- and y-coordinates of the pixel you want to read.

To set the color of a particular pixel,

- Assign a color value to the Pixels property, using as indexes the x- and y-coordinates of the pixel.

See also

Using pens

Using brushes

Example

Example for reading a pixel

Example for setting a pixel

Example

The following example responds to a click on a check box by changing the color of the text in the check box to the color of the pixel at the point (10, 10) in the form:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  CheckBox1.Font.Color := Canvas.Pixels[10, 10];
end;
```

Example

The following example responds to a click on a button by changing a random pixel on the form to red:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Canvas.Pixels[Random(ClientWidth), Random(ClientHeight)] := clRed;
end;
```

Using pens

Use the Pen property of a canvas to control how lines appear, including lines drawn as the outlines of shapes. Even though the pen is a property of an object, it itself is also an object.

Draw a straight line by changing a group of pixels that lie between two points.

The tasks below are unique to pens. The following is a list of tasks that either have multiple steps or require some programming knowledge.

Drawing lines and shapes

Moving the pen

Property tasks

The following table summarizes tasks that you can perform simply by setting properties. To find out how to set a property, see Setting properties.

Task	Property
Change the pen color	Color
Change the pen width	Width
Change the pen style	Style
Change the pen mode	Mode

Moving the pen

The PenPos property stores the current position of the pen. Pen position affects only the drawing of lines; for shapes and text, you specify all the needed coordinates.

To set the pen position,

- Call the MoveTo method of the canvas. The upper left corner of the canvas is the coordinate position (0, 0).

Drawing a line with the LineTo method also moves the current position to the endpoint of the line.

Using brushes

Use the Brushproperty of a canvas to control how you fill areas, including the interior of shapes. Filling an area with a brush changes a large number of adjacent pixels in a specified way.

The tasks below are unique to brushes.

Property tasks

The following table summarizes tasks that you can perform simply by setting properties. If you do not know how to set a property, see Setting properties.

Task	Property
Changing the fill color	Color
Changing the brush style	Style
Using a bitmap as a brush pattern	Bitmap

Drawing lines and shapes

See also Example

A canvas has methods for drawing many kinds of shapes.

The canvas draws the outline of the shape with its pen, then fills the interior (when applicable) with its brush.

To draw lines and shapes,

Call the associated method for that graphic.

The following table summarizes the shapes the canvas can draw and the methods needed to draw each shape.

To draw	Method	Comments
Arc	Arc	The Arc method draws an arc along the perimeter of the ellipse bounded by the rectangle.
Chord	Chord	The Chord method draws a line connecting two points on the ellipse bounded by the rectangle.
Ellipse	Ellipse	The Ellipse method draws an ellipse that touches all four sides of the rectangle.
Line	LineTo	LineTo draws a line from the current pen position to the point you specify and makes the endpoint of the line the current position. The canvas draws the line using its pen.
Pie Slice	Pie	
Polygon	Polygon	Polygon takes an array of points as its only parameter and connects the points with the pen, then connects the last point to the first to close the polygon. After drawing the lines, Polygon uses the brush to fill the area inside the polygon.
Polyline	PolyLine	The parameter passed to the PolyLine method is an array of points. Drawing a polyline is like performing a MoveTo on the first point and a LineTo on each successive point.
Rectangle	Rectangle	The Rectangle method draws the bounding rectangle.
Rounded Rectangle	RoundRect	The first four parameters are a bounding rectangle, and the last two indicate how to draw the rounded corners.

See also

Drawing vs. painting
Using pens
Using brushes

Example

The following method draws crossed diagonal lines across a form whenever the form gets painted:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  with Canvas do
  begin
    MoveTo(0, 0);
    LineTo(ClientWidth, ClientHeight);
    MoveTo(0, ClientHeight);
    LineTo(ClientWidth, 0);
  end;
end;
```

Examples

Example for drawing arcs
Example for drawing chords
Example for drawing ellipses
Example for drawing pie slices
Example for drawing polygons
Example for drawing polylines
Example for drawing rectangles
Example for drawing rounded rectangles
Example for drawing straight lines

Example

The following method draws a rhombus in a form:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  with Canvas do
    PolyLine([Point(0, 0), Point(50, 0), Point(75, 50), Point(25, 50),
Point(0, 0)]);
end;
```

This example takes advantage of Delphi's ability to automatically create an open-array parameter You can pass any array of points, but an easy way to construct an array quickly is to put its elements in brackets and pass the whole thing as a parameter.

Example

The following method draws a rectangle filling a form's upper left quadrant, then draws an ellipse in the same area:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Rectangle(0, 0, ClientWidth div 2, ClientHeight div 2);
  Canvas.Ellipse(0, 0, ClientWidth div 2, ClientHeight div 2);
end;
```

Example

The following method draws a rounded rectangle in a form's upper left quadrant, rounding the corners as if they were sections of a circle with a diameter of 10 pixels:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.RoundRect(0, 0, ClientWidth div 2, ClientHeight div 2, 10, 10);
end;
```

Example

The following example draws a right triangle in the lower left half of a window:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Polygon([Point(0, 0), Point(0, ClientHeight),
    Point(ClientWidth, ClientHeight)]);
end;
```

Add Fields dialog box

Use the Add Fields dialog box to create a persistent field component for a dataset:

To create a persistent field component for a dataset:

1 Right-click the Fields editor list box.

2 Choose Add fields from the SpeedMenu. The Add Fields dialog box appears

The Available fields list box displays all fields in the dataset which do not have persistent field components.

3 Select the fields for which you want to create persistent field components.

4 Click OK.

Each time you open the dataset, Delphi no longer creates dynamic field components for every column in the underlying database. It only creates persistent components for the fields you specified.

Each time you open the dataset, Delphi verifies that each non-calculated persistent field exists or can be created from data in the database. If it cannot, Delphi raises an exception warning you that the field is not valid, and does not open the dataset.

To delete a persistent field component:

1 Select the field(s) to remove in the Fields editor list box.

2 Press the Delete key.

Note: Fields you remove are no longer available to the dataset and cannot be displayed by data aware controls.

Fields editor

Use the Fields editor at design time to create persistent lists of the field components used by the datasets in your application. Persistent fields component lists are stored in your application, and do not change even if the structure of a database underlying a dataset is changed. All fields in a dataset are either persistent or dynamic.

To start the fields editor:

- Double-click the dataset component.

The Fields editor appears.

Using the Fields editor

The Fields editor contains a title bar, navigator buttons, and a list box.

Title bar

The title bar displays the both the name of the data module or form containing the dataset, and the name of the dataset itself.

Navigation buttons

Use these buttons to scroll one-by-one through the records in an active dataset at design time. You can also jump to the first or last record. The buttons are dimmed if the data set is not active or empty.

List box

The List box displays the names of persistent fields components for the dataset. The first time you invoke the Fields editor for a new dataset, the list is empty because the fields components for the dataset are dynamic, not persistent. If you invoke the Fields editor for a dataset that already has persistent fields components, you see the fields component names in the list box.

To add fields to the list of persistent fields for a dataset:

1 Right-click the list box to invoke the SpeedMenu for the Fields editor
2 Choose Add Fields.

Associate attributes dialog box

You can apply attribute sets to fields without having to recreate the settings manually if:

- If several fields in datasets used by your application share common formatting properties

  and

- You have saved those property settings as attribute sets in the Data Dictionary.

If you change the attributes in the Data Dictionary, those changes are automatically applied to every field associated with the set the next time field components are added to the dataset.

To apply an attribute set to a field component:

1 Double-click the dataset to invoke the Fields editor.

2 Select the field for which to apply an attribute set.

3 Right-click the Fields editor list box to invoke the SpeedMenu.

4 Choose Associate attributes.

5 Select or enter the attribute set to apply from the Attribute set name dialog box.

If there is an attribute set in the Data Dictionary that has the same name as the current field, that set name appears in the edit box.

Save attributes dialog box

When several fields in the datasets used by your application share common formatting properties, it is more convenient to set the properties for a single field, then store those properties as an attribute set in the Data Dictionary. Attribute sets stored in the data dictionary can be easily applied to other fields.

To create an attribute set based on a field component in a dataset:

1 Double-click the dataset to invoke the Fields editor.

2 Select the field for which to set properties.

3 Set the desired properties for the field in the Object Inspector.

4 Right-click the Fields editor list box to invoke the SpeedMenu.

5 Choose Save Attributes to save the current field's property settings as an attribute set in the Data Dictionary.

The name for the attribute set defaults to the name of the current field. You can specify a different name for the attribute set by choosing Save attributes as instead of Save attributes from the SpeedMenu.

Note: You can also create attribute sets directly from the Database Explorer. When you create an attribute set from the data dictionary, the set is not applied to any fields, but you can specify two additional attributes in the set: a field type and a data-aware control that is automatically placed on a form when a field based on the attribute set is dragged onto the form. For more information, see the online help for the Database Explorer.

Retrieve attributes

To retrieve an attribute set for a field component that is different in name from the field component name, choose Retrieve attributes to specify the name of the attribute set to retrieve.

Save as attributes

To save an attribute set and assign it a name that differs from the currently selected field component name, choose Save as attributes, and then enter the new attribute set name.

Unassociate attributes

To remove an attribute set assignment for a selected field component, choose Unassociate attributes.

Fields Editor edit options

Use these SpeeeMenu options to edit fields in the Field editor.

Cut

Use this option to remove selected field from the editor and place them on the Windows clipboard.

Copy

Use this option to copy selected fields to the Wincdows clipboard.

Paste

Use this option to paste the clipboard contents into an application.

Delete

Use this option to delete selected fields without copying them to the clipboard.

Select All

Use this option to select all the fields in the fields

DBGrid Columns editor

At design time, use the DBGrid Columns editor to create a set of personal column objects for the grid. At run time, the **State** property for a grid with persistent column objects is automatically set to **csCustomized**.

Column properties determine how data is displayed in the cells of that column. Most column properties obtain their default values from properties associated with another component, called the **default source**, such as a grid or an associated field component.

To create persistent columns for a grid control:

1 Select the grid component in the form.

2 Right-click to invoke the SpeedMenu, and choose Columns editor.

Columns editor options

The Columns editor contains a list box of defined columns, insertion and deletion buttons, and two tab pages of column properties: one for properties of the column object and one for properties of the column title object.

Columns List box

The Columns list box displays the persistent columns that have been defined for the selected grid. When you first bring up the Columns editor, this list is empty because the grid is in its default state, containing only dynamic columns.

New button

To create a persistent column for a grid, click on the New button in the Columns editor. The new column will be selected in the list box. If you want to associate a field with this new column, use the FieldName combo box to select a field from the grid's dataset.

Delete button

Use this button to delete a selected column.

Add All Fields button

To create columns for all the fields in the grid's dataset, click the Add All Fields button. If the grid already contained persistent columns, a dialog box asks if you want to delete the existing columns, or append to the column set.

Delete All Columns button

Use this button to delete a persistent column from a grid.

Note: If you delete all the columns from a grid, the grid reverts to its **csDefault** state and automatically builds dynamic columns for each field in the dataset.

The Column properties page

In the Columns editor, properties are divided into two pages: Column Properties and Title Properties. The following table summarizes the properties you can set on the Column Properties page.

Property	Purpose
Alignment	Left justifies, right justifies, or centers the field data in the column. Default source: TField.Alignment
ButtonStyle	cbsAuto: (default) Displays a drop-down list if the associated field is a lookup field, or if the column's PickList property contains data.
	cbsEllipsis: Displays an ellipsis (...) button to the right of the cell. Clicking on the button fires the grid's OnEditButtonClick event.
	cbsNone: The column uses only the normal edit control to edit data in the column
Color	Specifies the background color of the cells of the column. For text foreground color, see the font property. Default Source: TDBGrid.Color
DropDownRows	The number of lines of text displayed by the drop-down list. Default: 7
FieldName	Specifies the field name that is associated with this column. This can be blank.
ReadOnly	True: The data in the column cannot be edited by the user.
	False: (default) The data in the column can be edited.
Width	Specifies the width of the column in screen pixels. Default Source: derived from TField.DisplayWidth

Font button

Use this button to select the font attributes of the column text. Default Source: TDBGrid.Font

PickList button

Use this button to specify a list of strings to display in a drop-down list in the column. When the user chooses from the list, the corresponding field's value is set to what the user chose. PickList is similar to using a lookup field, but is more useful for lists of

items that rarely change or are fixed in the application.

Note: The table grid in your form must be linked to a data source in order for the PickList feature to work.

To define a pick list column:

1 Select the column in the **Columns** list box.
2 Set **ButtonStyle** to **cbsAuto**,
3 Click the PickList button.
The String List editor appears.
4 Enter a list of string values that you want to appear in the PickList.
5 Click OK.
The DBGrid Column Editor appears.
6 Click OK.

Restore Defaults button

You can discard all property changes in the selected column by clicking the Restore Defaults button. The column's properties will return to their default values.

The Title properties page

The following table summarizes the properties on the Title Property page.

Property	Purpose
Alignment	Left justifies (default), right justifies, or centers the caption text in the column title.
Caption	Specifies the text to display in the column title. Default Source: TField.DisplayLabel
Color	Specifies the background color used to draw the column title cell. Default Source: TDBGrid.FixedColor

Font button

Use this button to select the font attributes of the text in the column title. Default Source: TDBGrid.Title.Font

Update SQL editor

Use the Update SQL editor to create SQL statements for updating a dataset.

The Update SQL editor

The Update SQL editor has two pages, the Options page and the SQL page.

The Options page

The Options page is visible when you first invoke the editor.

Table Name combo box

Use the Table Name combo box to select the table to update. When you specify a table name, the Key Fields and Update Fields list boxes are populated with available columns.

Update Fields list box

The Update Fields list box indicates which columns should be updated. When you first specify a table, all columns in the Update Fields list box are selected for inclusion. You can multi-select fields as desired.

Key Fields list box

The is used to specify the columns to use as keys during the update. Generally the columns you specify here should correspond to an existing index, especially for local Paradox and dBASE tables, but having an index is not a requirement.

Select Primary Keys button

Click the Primary Key button to select key fields based on the primary index for a table.

Generate SQL button

After you specify a table, select key columns, and select update columns, click the Generate SQL button to generate the preliminary SQL statements to associate with the update component's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties.

Dataset Defaults button

Use this button to restore the default values of the associated dataset. This will cause all fields in the Key Fields list and the Update Fields list to be selected and the table name to be restored.

The SQL page

To view and modify the generated SQL statements, select the SQL page. If you have generated SQL statements, then when you select this page, the statement for the ModifySQL property is already displayed in the SQL Text memo box. You can edit the statement in the box as desired.

Important

Keep in mind that generated SQL statements are intended to be starting points for creating update statements. You may need to modify these statements to make them execute correctly. Test each of the statements directly yourself before accepting them.

Use the Statement Type radio buttons (Modify, Insert, and Delete) to switch among generated SQL statements and edit them as desired.

To accept the statements and associate them with the update component's SQL properties, click OK.

New Field dialog box

See also

Use the New Field dialog box to create new persistent fields as additions to or replacements of the other peristent fields in a dataset. There are three type of persistent fields you can create:

- **Data fields**, which usually replace existing fields (for example to change the data type of a field), are based on columns in the table or query underlying a dataset.
- **Calculated fields**, which displays values calculated at run time by a dataset's OnCalcFields event handler.
- **Lookup fields**, which retrieve values from a specified dataset at run time based on search criteria you specify.

These types of persistent fields are only for display purposes. The data they contain at run time are not retained either because they already exist elsewhere in your database, or because they are temporary. The physical structure of the table and data underlying the dataset is not changed in any way.

To create a new persistent field component:

1 Right-click the Fields editor list box.

2 Choose New field from the SpeedMenu.

The New Field dialog box appears.

The New Field dialog box

The New Field dialog box contains three group boxes: Field properties, Field type, and Lookup definition.

Field type group

The Field type radio group enables you to specify the type of new field component to create. The default type is Data. If you choose Lookup, the Dataset and Source Fields edit boxes in the Lookup definition group box are enabled.

Field properties group

The Field properties group box enables you to enter general field component information. Enter the component's field name in the Name edit box. The name you enter here corresponds to the field component's **FieldName** property. Delphi uses this name to build a component name in the Component edit box. The name that appears in the Component edit box corresponds to the field component's **Name** property and is only provided for informational purposes (**Name** contains the identifier by which you refer to the field component in your source code). Delphi discards anything you enter directly in the Component edit box.

Type combo box

The Type combo box in the Field properties group enables you to specify the field component's data type. You must supply a data type for any new field component you create. For example, to display floating point currency values in a field, select **Currency** from the drop-down list. The Size edit box enables you to specify the maximum number of characters that can be displayed or entered in a string-based field or the size of **Bytes** and **VarBytes** fields. For all other data types, Size is meaningless.

Lookup definition group

The Lookup definition group box is only used to create lookup fields. For more information, see Defining a lookup field.

Defining a data field

A data field replaces an existing field in a dataset. For example, for programmatic reasons you might want to replace a **TSmallIntField** with a **TIntegerField**. Because you cannot change a field's data type directly, you must define a new field to replace it.

Important

Even though you define a new field to replace an existing field, the field you define must derive its data values from an existing column in a table underlying a dataset.

To create a data field in the New Field dialog box:

1. Enter the name of an existing persistent field in the Name edit box. Do not enter a new field name.
2. Choose a new data type for the field from the Type combo box. The data type you choose should be different from the data type of the field you are replacing.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type **TStringField**, **TBytesField**, and **TVarBytesField**.
4. Select Data in the Field type radio group if it is not already selected.
5. Choose OK. The New Field dialog box closes, the newly defined data field replaces the existing field you specified in Step 1, and the component declaration in the data module or form's **type** declaration is updated.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector.

Defining a calculated field

A calculated field displays values calculated at run time by a dataset's **OnCalcFields** event handler. For example, you might create a string field that displays concatenated values from other fields.

To create a calculated field in the New Field dialog box:

1. Enter a name for the calculated field in the Name edit box. Do not enter the name of an existing field.
2. Choose a data type for the field from the Type combo box.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type **TStringField**, **TBytesField**, and **TVarBytesField**.
4. Select Calculated in the Field type radio group.
5. Choose OK. The newly defined calculated field is automatically added to end of the list of persistent fields in the Field editor list box, and the component declaration is automatically added to the form's **type** declaration in the source code.
6. Place code that calculates values for the field in the **OnCalcFields** event handler for the dataset.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector.

Programming a calculated field

After you define a calculated field, you must write code to calculate its value. Otherwise it always has a null value. Code for a calculated field is placed in the **OnCalcFields** event for its dataset.

Defining a lookup field

A lookup field displays values it searches for at run time based on search criteria. In its simplest form, a lookup field is passed the name of a field to search, a field value to search for, and the field in the lookup dataset whose value it should display.

For example, consider a mail-order application that enables an operator use a lookup field to determine automatically the city and state that correspond to a zip code a customer provides. In that case, the column to search on might be called **ZipTable.Zip**, the value to search for is the customer's zip code as entered in **Order.CustZip**, and the values to return would be those in the **ZipTable.City** and **ZipTable.State** columns for the record where **ZipTable.Zip** matches the current value in the **Order.CustZip** field.

To create a lookup field in the New Field dialog box:

1. Enter a name for the lookup field in the Name edit box. Do not enter the name of an existing field.
2. Choose a data type for the field from the Type combo box.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type **TStringField**, **TBytesField**, and **TVarBytesField**.
4. Select Lookup in the Field type radio group. Selecting Lookup enables the Dataset and Key Fields combo boxes.
5. Choose from the Dataset combo box drop-down list the dataset in which to look up field values. The lookup dataset must be different from the dataset for the field component itself, or a circular reference exception is raised at run time. Specifying a lookup dataset enables the Lookup Keys and Result Field combo boxes.
6. Choose from the Key Fields drop-down list a field in the current dataset for which to match values. To match more than one field, enter field names directly instead of choosing from the drop-down list. Separate multiple field names with semicolons.
7. Choose from the Lookup Keys drop-down list a field in the lookup dataset to match against the Source Fields field you specified in step 6. To specify more than one field, enter field names directly instead. Separate multiple field names with semicolons.
8. Choose from the Result Field drop-down list a field in the lookup dataset to return as the value of the lookup field you are creating. To return values from more than one field in the lookup dataset, enter field names directly instead. Separate multiple field names with semicolons.

Index Files editor

For dBASE tables that use non-production indexes set the **IndexFiles** property to the name of the index file(s) to use before you set **IndexName**. At design time you can click the ellipsis button in the **IndexFiles** property value in the Object Inspector to invoke the Index Files editor.

To see a list of available index files, choose Add, and select one or more index files from the list. A dBASE index file can contain multiple indexes. To select an index from the index file, select the index name from the **IndexName** drop-down list in the Object Inspector. You can also specify multiple indexes in the file by entering desired index names, separated by semicolons.

You can also set **IndexFiles** and **IndexName** at run time. For example, the following code sets the **IndexFiles** for the **AnimalsTable** table component to ANIMALS.MDX, and then sets **IndexName** to NAME:

```
AnimalsTable.IndexFiles := 'ANIMALS.MDX';
AnimalsTable.IndexName := 'NAME';
```

Query Parameters editor

At design time, the easiest and safest way to enter query parameters is to invoke the Query Parameters editor. The Query Parameters editor lists parameters in the correct order, and lets you assign values to them.

To invoke the Query Parameters editor:

1 Select the query component.

2 Right-click the component to invoke the SpeedMenu.

3 Choose Define Parameters.

Using the Query Parameters editor

For queries without previously defined parameters, the Parameter name list box is empty. When you define parameters, this list box displays them.

Data type combo box

The Data type combo box lists the data type for a parameter selected in the list box. If you add a parameter to the list, you must set its data type.

Value edit box

The Value edit box enables you to enter a value for a selected parameter, and the Null Value check box enables you set a Null value for the selected parameter if its data type permits Null values.

Note Defining parameters at design time also prepares the query for execution. A query with parameters must be prepared before it can be executed at run time. To signal the end of parameter definition, choose OK.

StoredProc Parameters editor

Many stored procedures require you to pass them a series of input arguments, or *parameters*, to specify what and how to process. You specify input parameters in the *Params* property. The order in which you specify input parameters is significant, and is determined by the stored procedure definition on the server.

At design time, the easiest and safest way to enter input parameters is to invoke the StoredProc Parameters editor. The StoreProc Parameters editor lists input parameters in the correct order, and lets you assign values to them.

To invoke the StoredProc Parameters editor:

1 Select the stored procedure component.

2 Right-click the component to invoke the SpeedMenu.

3 Choose Define Parameters.

The StoredProc Parameters editor

Parameter name list box

The Parameter name list box displays all input, output, and result parameters for the procedure. Information on input and output parameters is retrieved from the server. For some servers (such as Sybase), parameter information may not be accessible. In this case, the Parameter name list box is empty, and you must add the names and order of input and output parameters in order to use the procedure.

Parameter type combo box

The Parameter type combo box describes whether a parameter selected in the list box is an input, output, or results parameter. If a server's stored procedure allows it, a single parameter may be both an input and output. If you add a parameter to the list, you must set the parameter type for it.

Data type combo box

The Data type combo box lists the data type for a parameter selected in the list box. If you add a parameter to the list, you must set its data type.

Value edit box

The Value edit box enables you to enter a value for a selected input parameter, and the Null Value check box enables you set a Null value for the selected input parameter if its data type supports Null values.

Add button

The Add button enables you to add parameters to a stored procedure definition when your server does not pass the information to Delphi. The Delete button enables you to remove parameters you have added, and the Clear button removes all parameters from the list. Do not add, delete, or clear parameters for servers that pass parameter information to Delphi except if you are working with Oracle overloaded stored procedures.

Note: Defining parameters at design time also prepares the stored procedure for execution. A stored procedure must be prepared before it can be executed at run time. To signal the end of parameter definition, choose OK.

Field Link designer

At design time, double-click on the **MasterFields** property in the Object Inspector to invoke the Field Link designer.

Using the Field Link designer

The Field Link Designer provides a visual way to link master and detail tables.

Available Indexes combo box

The Available Indexes combo box shows the currently selected index used to join the tables. Unless you specify a different index name in the table's **IndexName** property, the default index used for the link is the primary index for the table. Other available indexes defined on the table can be selected from the drop-down list.

To link master and detail tables:

1 Select the field to use to link the detail table in the Detail Fields list

2 Select the field to link the master table in the Master Fields list.

3 Choose Add.

The selected fields are be displayed in the Joined Fields list box. For example,

```
OrderNo -> OrderNo
```

Note: For tables on a database server, the Available Indexes combo box will not appear, and you must manually select the detail and master fields to join in the Detail Fields and Master Fields list boxes.

ListView Columns Editor

Use the **ListView Columns** editor at design time to control the number of columns and their headings in a listview component. When the **ViewStyle** property of a listview component is set to vsReport, and **ShowColumnHeaders** is set to True, the headings specified with the **ListView Columns** editor are visible at run time.

To invoke the ListView Columns editor:

- Double-click the Columns property value in the Object Inspector.

Using the ListView Columns editor

The ListView Columns editor contains a Columns list box, New and Delete buttons, and a Column Properties group box containing a Caption edit box, an Alignment combo box, and a Width group box containing three radio buttons and an edit box. When you enter or change column properties, the Apply button is enabled so that you can activate changes immediately.

Columns list box

The Columns list box displays the names of existing column headings for a listview component. For new components this list is always empty. To add a column, click New, and then specify the column's properties. To delete a column, click Delete.

Caption edit box

Specify the heading for a new or existing column in this box. As you enter the name, it appears in the Columns list box.

Alignment combo box

Choose the alignment for a column heading in this box. Choices are Left Justify, Center, and Right Justify. Note that the first column heading is always left justified; you cannot change the alignment for this column.

Width group box

The controls in the Width group box specify the display width of a column. Select the method for calculating width using the radio buttons. Width Value enables you to enter a specific width in the Width edit box. Header Width specifies that the column's width is as wide as the text that appears in the column heading. Item Width specifies that column's width should be as wide as the widest item that appears in the column.

ListView Items Editor

Use the ListView Items editor at design time to add or delete the items displayed in a listview component. You can add or delete new items and sub-items, and you can set the caption, image index, and state index for each item in the ListView Items editor.

To invoke the ListView Items editor:

- Double-click the Items property value in the Object Inspector.

Using the ListView Items editor

The ListView Items editor contains an Items group box with an Items list box, a New Item button, a New SubItem button, and a Delete button. When you first add a listview control to a form, the Items list box is empty and the New SubItem and Delete buttons are disabled. When you enter or change item properties for a selected item, the Apply button is enabled so that you can activate changes immediately.

The ListView Items editor also contains an Item Properties group box for setting the properties of the listview item currently selected in the Items list box. The Item Properties group box contains a Caption edit box, an Image Index edit box, and a State Index edit box.

Items group box

Create and delete listview items and subitems in the Items group box. To create a new item, click New Item. A default item caption appears in the Items list box. Specify an item's properties, including its caption, in the Items Properties group box. When you create a new item, or select an existing item, the New SubItem button is enabled so that you can nest items within other items in the listview. If the Items list box contains items, the Delete button is also enabled. To delete an item, select it in the Items list box and click Delete.

Item Properties group box

Set the properties for a selected item in the Item Properties group box. Enter a name for the item in the Caption edit box. As you enter the name, it changes in the Items list box.

To display an image to the left of an item that is not currently selected, specify the index number of the image in the Image Index edit box. To suppress image display, set Image Index to -1 (the default).

To display an image to the left of an item that is currently selected, specify the index number of the image in the State Index edit box. The index number represents an index into the StateImages property of the listview component. To suppress image display, set State Index to -1 (the default).

TreeView Items Editor

Use the TreeView Items editor at design time to add items to a treeview component, delete items from a treeview component, or load images from disk into a treeview component. You can specify the text associated with individual treeview items, and set the image index, selected index, and state index for items.

To invoke the TreeView Items editor:

- Double-click the Items property value in the Object Inspector.

Using the TreeView Items editor

The TreeView Items editor contains an Items group box with an Items list box, a New Item button, a New SubItem button, a Delete button, and a Load button. When you first add a treeview control to a form, the Items list box is empty, and the New SubItem and Delete buttons are disabled. When you enter or change item properties for a selected item the Apply button is enabled so that you can activate changes immediately.

The TreeView Items editor also contains an Item Properties group box for setting the properties of the treeview item currently selected in the Items list box. The Item Properties group box contains a Text edit box, and Image Index edit box, a Selected Index edit box, and a State Index edit box.

Items group box

Create, load, and delete treeview items and subitems in the Items group box. To load a set of existing treeview items from disk, click Load. To create a new item, click New Item. Default text for the item appears in the Items list box. Specify an item's properties, including its text, in the Items Properties group box.

When you create a new item, or select an existing item, the New SubItem button is enabled so that you can nest items within other items in the treeview. If the Items list box contains items, the Delete button is also enabled. To delete an item, select it in the Items list box and click Delete.

Item Properties group box

Set the properties for a selected item in the Item Properties group box. Enter text for the item in the Caption edit box. As you enter the name, it changes in the Items list box.

To display an image to the left of an item that is not currently selected, specify the index number of the image in the Image Index edit box. To suppress image display, set Image Index to -1 (the default).

To display an image to left of a selected item, specify the index number of the image in the Selected Index edit box. The index is zero-based. To suppress image display, set Selected Index to -1 (the default).

To display an additional image to the left of an item , specify the index number of the image in the State Index edit box. The index number represents an index into the StateImages property of the listview component. The index is zero-based. To suppress image display, set State Index to -1 (the default).

HeaderControl Sections Editor

Use the HeaderControl Sections editor at design time to divide a header control into sections and to assign properties to those sections. A section is a resizeable, moveable header within a header control.

To invoke the HeaderControl Sections editor:

- Double-click the Sections property value in the Object Inspector.

Using the HeaderControl Sections editor

The HeaderControl Sections editor contains a Sections group box, and a Section properties group box. When you first add a header control to a form, the Sections list box is empty, and Delete button is disabled. When you enter or change item properties for a selected item the Apply button is enabled so that you can activate changes immediately.

The Sections group box contains a Sections list box that displays the names of individual sections within a header control, a New button for creating a new section, and a Delete button for deleting sections.

The Section properties group box contains a Text edit box, a Width edit box, Min and Max edit boxes, a Style combo box, an Alignment combo box, and an Allow click check box.

Sections group box

Create and delete sections for a header control in the Sections group box. To create a section, click New. A numbered section header, named "Untitled," appears in the Sections list box, and the Delete button is enabled. Default properties for Width, Min, Max, Style, Alignment, and Allow click appear in the Section properties group box.

To delete a section, select it in the Sections list box and click Delete.

Section properties group box

Set the properties for a selected header control section in the Section properties group box. In the Text edit box enter the text to display in the section within the header control. In the the Width box, specify a default width (in pixels) for the section. The proposed default value for Width is 50. To specify a minimum size allowed for the section, specify the minimum width in the Min edit box. The proposed default value for Min is 0. To specify a maximum size for the section, specify the maximum width in the Max edit box. The default value for Max is 10000.

To specify the drawing style for the section, choose a style in the Style combo box. By default, Style is set to Text, meaning that text only is displayed in the section. To display an image in the header, set Style to OwnerDraw and write the code to draw an object in the header at run time.

To specify the alignment for the text or image displayed in the section at run time, choose an alignment in the Alignment combo box. By default, Alignment is set to Left Justify.

To prevent users from clicking a section, uncheck the Allow click check box. Allow click is checked by default, enabling users to click a section.

StatusBar Panels Editor

Use the StatusBar Panels editor at design time to divide a status bar control into panels and to assign properties to those panels. A panel is a fixed width section within a status bar.

To invoke the StatusBar Panels editor:

- Double-click the Panels property value in the Object Inspector.

Using the StatusBar Panels editor

The StatusBar Panels editor contains a Panels group box, and a Panel properties group box. When you first add a status bar control to a form, the Panels list box is empty, and Delete button is disabled. When you enter or change item properties for a selected item the Apply button is enabled so that you can activate changes immediately.

The Panels group box contains a Panels list box that displays the names of individual panels within a status bar control, a New button for creating a new panel, and a Delete button for deleting panels.

The Panel properties group box contains a Text edit box, a Width edit box, a Style combo box, a Bevel combo box, and an Alignment combo box.

Panels group box

Create and delete panels for a status bar control in the Panels group box. To create a panel, click New. A numbered panel , named "Untitled," appears in the Panels list box, and the Delete button is enabled. Default properties for Width, Style, Bevel, and Alignment appear in the Panel properties group box.

To delete a panel, select it in the Panel list box and click Delete.

Panel properties group box

Set the properties for a selected panel in the Panel properties group box. In the Text edit box enter the text to display in the panel within the status bar control. In the the Width box, specify a default width (in pixels) for the panel. The proposed default value for Width is 50. To specify the drawing style for the panel, choose a style in the Style combo box. By default, Style is set to Text, meaning that text only is displayed in the panel. To display an image in the panel, set Style to OwnerDraw and write the code to draw an object in the panel at run time.

To specify the bevel type for the panel, choose a bevel type in the Bevel combo box. By default, Bevel is set to Lowered.

To specify the alignment for the text or image displayed in the panel at run time, choose an alignment in the Alignment combo box. By default, Alignment is set to Left Justify.

Add page dialog box

Specify the name for a new page to add to the Object Repository in the Add page edit box. After you add a page to the Object Repository, it appears as a separate tab sheet when you choose File|New to invoke the New Items dialog box for the Object Inspector.

Rename page dialog box

Specify a new name for a an existing page in the Object Repository in the Rename page edit box. After you rename a page to the Object Repository, it appears in place of the old name on the existing page.

Instantiating forms at run time

Example

If you move a form into the Available Forms List Box in the Forms page of the Project Options dialog box, you must instantiate that form at run time. You do not instantiate any forms that are in the Auto-Create Forms List Box, as Delphi creates these forms automatically at run time.

Instantiate a form a run time when you cannot determine at design time how many instances of the form will be required when the application is running. For example, if you write an MDI application that allows the user to open any number of MDI child windows, you should instantiate each child form at run time. Instantiating forms at run time can also reduce the memory requirements of the application and reduce the amount of startup time when the application is run.

Note: When creating an application that instantiates forms at run time, make sure that the code of the application does not try to access the instance of the form before it has been created.

To instantiate a form at run time,

1. Add the name of the unit where the form is declared to the **uses** clause of the unit that will instantiate the form. This is necessary only if the form type is declared in a different unit.
2. Declare a memory variable of the type of the form.
3. Assign the return value of the Create constructor of the form type to the memory variable.

   The memory variable specifies the instance of the form type.

Note: The name of the memory variable identifier that the instance of the form is assigned to should not be the name of an existing object or component. While unique instance names are not required at run time, they are recommended so that the instance of the form is not confused with the Name of another object.

Example

The following example declares a memory variable called SpecForm1 of type TSpecForm. Then, an instance of TSpecForm is created and assigned to SpecForm1. Finally, SpecForm1 is displayed with the Show method.

Note: The unit in which TSpecForm is declared (SpecUnit) should be added to the **uses** clause of the unit that contains this code.

```
uses ..., SpecUnit;        { Add the unit name to the uses clause }
:
var
  SpecForm1: TSpecForm;    { Declare the form instance variable }
:
SpecForm1 := TSpecForm.Create(Self);  { Create the form instance }
SpecForm1.Show;                        { Show SpecForm1 }
```

System (brief)

These system keyboard shortcuts apply to the Brief keystroke mapping scheme.

Shortcut	Action or command
F7	Records a keyboard macro
F8	Plays back a keyboard macro
F9	Run|RunMRunRun
F10	Accesses the menu bar
F11	View|Object Inspector
F12	View|Toggle Form/Unit
Alt+F2	Add watch at cursorCELMAddWatchAtCursor
Alt+F7	Run to cursorCELMRunToCursor
Alt+F9	Displays a SpeedMenuDefSpeedMenu
Alt+F10	Project|CompileMProjectCompile
Ctrl+F1	Topic searchCELMTopicSearch
Ctrl+F2	Run|Program ResetMRunProgramReset
Ctrl+F3	View|Call StackMViewsCallStack
Ctrl+F7	Evaluate/modifyCELMEvaluateModify
Ctrl+F8	Toggle breakpointCELMToggleBreakpoint
Ctrl+F9	Project|CompileMProjectCompile
Ctrl+F11	Run|Step overMRunStepOver
Ctrl+Hyphen	File|CloseMFileClose
Ctrl+F12	View|UnitsMViewUnit
Shift+ F12	View|FormsMViewForm
Alt+B	View|Window listMViewWindowList
Alt+E	File|OpenMFileOpen (Note: opens Open dialog box, even when Code Editor window does not have focus)
Alt+H	Displays context-sensitive Help
Alt+N	Displays the next page
Alt+O	File|Save AsMFileSaveAs (Note: opens Save As dialog box, even when Code Editor window does not have focus)
Alt+-	Displays the previous page
Alt+W	File|SaveMFileSave
Alt+X	File|ExitMFileExit
Alt+Z	Accesses the File menuMFile

Clipboard control (brief)

These Clipboard keyboard shortcuts apply to the Brief keystroke mapping scheme.

Shortcut	Command
Ins	Edit|PasteMEditPaste
Plus (+)	Edit|CopyMEditCopy
Minus (-)	Edit|CutMEditCut

Editor (brief)

See also

These editor keyboard shortcuts apply to the Brief keystroke mapping scheme.

Shortcut	Action or command
F5	Search|FindMSearchFind (forward from cursor position)
F6	Search|ReplaceMSearchReplace (forward from cursor position)
Alt+F2	Control|MaximizeCECMMaximize
Alt+F5	Search|FindMSearchFind (backward from cursor position)
Alt+F6	Search|ReplaceMSearchReplace (backward from cursor position)
Alt+F9	Displays the local menu
Shift+F4	Tiles windows horizontally
Shift+F5	Search|Search AgainMSearchSearchAgain
Shift+F6	Repeats the last Search|ReplaceMSearchReplace operation
Esc	Cancels a command at a prompt or enters an escape character at the cursor
Del	Deletes a character or block at the cursor
Star	Edit|UndoMEditUndo
Backspace	Deletes the character to the left of the cursor
Shift+Backspace	Deletes the character to the left of the cursor
Tab	Inserts a tab character
Enter	Inserts a new line with a carriage return
Ctrl+B	Moves to the bottom of the window
Ctrl+C	Centers line in window
Ctrl+D	Moves down one screen
Ctrl+E	Moves up one screen
Ctrl+K	Deletes to the beginning of a line
Ctrl+M	Inserts a new line with a carriage return
Ctrl+S	Performs an incremental search
Ctrl+T	Moves to the top of the window
Ctrl+U	Edit|RedoMEditRedo
Ctrl+Backspace	Deletes the word to the left of the cursor
Ctrl+Enter	Inserts an empty new line
Ctrl+- (dash)	Closes the current page
Alt+A	Marks a non-inclusive block
Alt+B	Displays a list of open files
Alt+C	Mark the beginning of a column block
Alt+D	Deletes a line
Alt+G	Search|Go to line numberMSearchGoToLineNumber
Alt+I	Toggles insert mode
Alt+K	Deletes of the end of a line
Alt+L	Marks a line
Alt+M	Marks an inclusive block
Alt+N	Displays the contents of the next page

Alt+P	Prints the selected block
Alt+Q	Causes next character to be interpreted as an ASCII sequence
Alt+R	Reads a block from a file
Alt+S	Search|FindMSearchFind
Alt+T	Search|ReplaceMSearchReplace
Alt+U	Edit|UndoMEditUndo
Alt+Backspace	Deletes the word to the right of the cursor
Alt+Hyphen	Displays the contents of the previous file buffer
Ctrl+Q+[	Finds the matching delimiter (forward)
Ctrl+Q+Ctrl+[	Finds the matching delimiter (forward)
Ctrl+Q+]	Finds the matching delimiter (backward)
Ctrl+Q+Ctrl+]	Finds the matching delimiter (backward)
Ctrl+O+A	Open file at cursorCELMOpenFileAtCursor
Ctrl+O+B	Browse symbol at cursorCELMBrowseSymbolAtCursor
Ctrl+O+O	Toggles the case of a selection
Ctrl+F1	Help keyword search
Ctrl+F5	Toggles case-sensitive searching
Ctrl+F6	Toggles regular expression searching

See also

Block commands
Bookmark operations
Cursor movement

Block commands (brief)

See also

These block command keyboard shortcuts apply to the Brief keystroke mapping scheme.

Shortcut	Action
Alt+A	Marks a non-inclusive block
Alt+C	Marks a column as a block
Alt+L	Marks a line as a block
Alt+M	Marks an inclusive block
Alt+P	Prints the contents of a block
Alt+R	Reads a block from a file

See also

Bookmark operations
Cursor movement
Editor

Bookmark operations (brief)

See also

These bookmark operations keyboard shortcuts apply to the Brief keystroke mapping scheme.

Shortcut	Action
Alt+0	Sets bookmark 0
Alt+1	Sets bookmark 1
Alt+2	Sets bookmark 2
Alt+3	Sets bookmark 3
Alt+4	Sets bookmark 4
Alt+5	Sets bookmark 5
Alt+6	Sets bookmark 6
Alt+7	Sets bookmark 7
Alt+8	Sets bookmark 8
Alt+9	Sets bookmark 9
Alt+J+0	Goes to bookmark 0
Alt+J+1	Goes to bookmark 1
Alt+J+2	Goes to bookmark 2
Alt+J+3	Goes to bookmark 3
Alt+J+4	Goes to bookmark 4
Alt+J+5	Goes to bookmark 5
Alt+J+6	Goes to bookmark 6
Alt+J+7	Goes to bookmark 7
Alt+J+8	Goes to bookmark 8
Alt+J+9	Goes to bookmark 9

See also

Block commands
Cursor movement
Editor

Cursor movement (brief)

See also

These cursor movement keyboard shortcuts apply to the Brief keystroke mapping scheme.

Shortcut	Action
UpArrow	Moves up one line in the same column position
DownArrow	Moves down one line in the same column position
Home	Moves to the start of a line
End	Moves to the end of a line
Left Arrow	Moves one character to the left
Right Arrow	Moves one character to the right
PgDn	Moves down one screen in the current window
PgUp	Moves up one screen in the current window
Ctrl+Left Arrow	Moves one word to the left
Ctrl+Right Arrow	Moves one word to the right
Ctrl+PgDn	Moves to the end of a file
Ctrl+PgUp	Moves to the beginning of a file
Shift+Tab	Moves backward one tab stop
Shift+Home	Moves to the first column in a window
Shift+End	Moves to the last column in a window
Ctrl+Home	Moves to the top of a screen in the same column position
Ctrl+End	Moves to the bottom of a screen in the same column position
Ctrl+B	Moves to the bottom of the window
Ctrl+C	Moves to the center of the window
Ctrl+D	Scrolls down one screen
Ctrl+E	Scrolls down one screen

See also

Block commands
Bookmark operations
Editor

System (classic)

These system keyboard shortcuts apply to the Classic keystroke mapping scheme.

Shortcut	Action or command
F1	Displays context-sensitive Help
F2	File|Save
F3	File|Open
F4	Run to Cursor
F5	Control|Maximize
F6	Displays the next page
F7	Run|Trace Into
F8	Run|Step Over
F9	Run|Run
F11	View|Object Inspector
F12	View|Toggle Form/Unit
Alt+F3	File|Close
Alt+F10	Displays a SpeedMenu
Alt+X	File|Exit
Alt+0	View|Window LIst
Ctrl+F1	Topic Search
Ctrl+F2	Run|Program Reset
Ctrl+F3	View|Call Stack
Ctrl+F4	Evaluate/Modify
Ctrl+F7	Add Watch at Cursor
Ctrl+F8	Toggle Breakpoint
Ctrl+F9	Project|Compile
Ctrl+F12	View|Units
Shift+F12	View|Forms
Ctrl+Shift+P	Plays back a keyboard macro
Ctrl+Shift+R	Records a keyboard macro
Ctrl+Shift+S	Performs an incremental search
Ctrl+K+D	Accesses the menu bar
Ctrl+K+S	File|Save

Clipboard control (classic)

These Clipboard control keyboard shortcuts apply to the Classic keystroke mapping scheme.

Shortcut	Command
Ctrl+Ins	Edit|Copy
Shift+Del	Edit|Cut
Shift+Ins	Edit|Paste

Editor (classic)

See also

These editor keyboard shortcuts apply to the Classic keystroke mapping scheme.

Shortcut	Action or command
F1	Topic Search
Ctrl+F1	Topic Search
F6	Displays the next page
Shift+F6	Displays the previous page
Ctrl+A	Moves one word left
Ctrl+C	Scrolls down one screen
Ctrl+D	Moves the cursor right one column, accounting for the autoindent setting
Ctrl+E	Moves the cursor up one line
Ctrl+F	Moves one word right
Ctrl+G	Deletes the character to the right of the cursor
Ctrl+H	Deletes the character to the left of the cursor
Ctrl+I	Inserts a tab
Ctrl+L	Search|Search Again
Ctrl+N	Inserts a new line
Ctrl+P	Causes next character to be interpreted as an ASCII sequence
Ctrl+R	Moves up one screen
Ctrl+S	Moves the cursor left one column, accounting for the autoindent setting
Ctrl+T	Deletes a word
Ctrl+V	Turns insert mode on/off
Ctrl+W	Moves down one screen
Ctrl+X	Moves the cursor down one line
Ctrl+Y	Deletes a line
Ctrl+Z	Moves the cursor up one line
Ctrl+Shift+S	Performs an incremental search
End	Moves to the end of a line
Home	Moves to the start of a line
Enter	Inserts a carriage return
Ins	Turns insert mode on/off
Del	Deletes the character to the right of the cursor
Backspace	Deletes the character to the left of the cursor
Tab	Inserts a tab
Space	Inserts a blank space
Left Arrow	Moves the cursor left one column, accounting for the autoindent setting
Right Arrow	Moves the cursor right one column, accounting for the autoindent setting
Up Arrow	Moves up one line
Down Arrow	Moves down one line
Page Up	Moves up one page
Page Down	Moves down one page
Ctrl+Left Arrow	Moves one word left
Ctrl+Right Arrow	Moves one word right
Ctrl+Home	Moves to the top of a screen
Ctrl+End	Moves to the end of a screen
Ctrl+PgDn	Moves to the bottom of a file
Ctrl+PgUp	Moves to the top of a file
Ctrl+Backspace	Move one word to the right
Ctrl+Del	Deletes a currently selected block
Ctrl+Space	Inserts a blank space
Ctrl+Enter	Opens file at cursor
Ctrl+Tab	Moves to the next page

Shift+Tab	Deletes the character to the left of the cursor
Shift+Backspace	Deletes the character to the left of the cursor
Shift+Left Arrow	Selects the character to the left of the cursor
Shift+Right Arrow	Selects the character to the right of the cursor
Shift+Up Arrow	Moves the cursor up one line and selects from the left of the starting cursor position
Shift+Down Arrow	Moves the cursor down one line and selects from the right of the starting cursor position
Shift+PgUp	Moves the cursor up one screen and selects from the left of the starting cursor position
Shift+PgDn	Moves the cursor down one line and selects from the right of the starting cursor position
Shift+End	Selects from the cursor position to the end of the current line
Shift+Home	Selects from the cursor position to the start of the current line
Shift+Space	Inserts a blank space
Shift+Enter	Inserts a new line with a carriage return
Shift+Ctrl+Tab	Moves to the previous page
Ctrl+Shift+Left Arrow	Selects the word to the left of the cursor
Ctrl+Shift+Right Arrow	Selects the word to the right of the cursor
Ctrl+Shift+Home	Selects from the cursor position to the start of the current file
Ctrl+Shift+End	Selects from the cursor position to the end of the current file
Ctrl+Shift+PgDn	Selects from the cursor position to the bottom of the screen
Ctrl+Shift+PgUp	Selects from the cursor position to the top of the screen
Ctrl+Shift+Tab	Moves to the previous page
Alt+Backspace	Edit|Undo
Alt+Shift+Backspace	Edit|Redo
Alt+Shift+Left Arrow	Selects the column to the left of the cursor
Alt+Shift+Right Arrow	Selects the column to the right of the cursor
Alt+Shift+Up Arrow	Moves the cursor up one line and selects the column from the left of the starting cursor position
Alt+Shift+Down Arrow	Moves the cursor down one line and selects the column from the left of the starting cursor position
Alt+Shift+Page Up	Moves the cursor up one screen and selects the column from the left of the starting cursor position
Alt+Shift+Page Down	Moves the cursor down one line and selects the column from the right of the starting cursor position
Alt+Shift+End	Selects the column from the cursor position to the end of the current line
Alt+Shift+Home	Selects the column from the cursor position to the start of the current line
Ctrl+Alt+Shift+Left Arrow	Selects the column to the left of the cursor
Ctrl+Alt+Shift+Rght Arrow	Selects the column to the right of the cursor
Ctrl+Alt+Shift+Home	Selects the column from the cursor position to the start of the current file
Ctrl+Alt+Shift+End	Selects the column from the cursor position to the end of the current file
Ctrl+Alt+Shift+Page Up	Selects the column from the cursor position to the bottom of the screen
Ctrl+Alt+Shift+Page Down	Selects the column from the cursor position to the top of the screen

See also

Block commands

Bookmark operations

Cursor movement

Miscellaneous commands

System (default)

These system keyboard shortcuts apply to the Default keystroke mapping scheme.

Shortcut	Action or command
F1	Displays context-sensitive Help
F4	Run|Go to Cursor
F5	Run|Toggle Breakpoint
F7	Run|Trace Into
F8	Run|Step Over
F9	Run|Run
F11	View|Object Inspector
F12	View|Toggle Form/Unit
Alt+F10	Displays a SpeedMenu
Alt+0	View|Window List
Ctrl+F1	Help|Topic Search
Ctrl+F2	Run|Program Reset
Ctrl+F3	View|Call Stack
Ctrl+F4	Closes current file
Ctrl+F5	Add Watch at Cursor
Ctrl+F7	Evaluate/Modify
Ctrl+F9	Project|Compile
Ctrl+F12	View|Units
Shift+F12	View|Forms
Ctrl+Shift+P	Plays back a key macro
Ctrl+Shift+R	Records a key macro
Ctrl+K+D	Accesses the menu bar
Ctrl+K+S	File|Save

Clipboard control (default)

These Clipboard keyboard shortcuts apply to the Default keystroke mapping scheme.

Shortcut	Command
Ctrl+Ins	Edit|Copy
Shift+Del	Edit|Cut
Shift+Ins	Edit|Paste
Ctrl+C	Edit|Copy
Ctrl+V	Edit|Paste
Ctrl+X	Edit|Cut

Editor (default)

See also

These editor keyboard shortcuts apply to the Default keystroke mapping scheme.

Shortcut	Action or command
F1	Help|Topic Search
Ctrl+F1	Help|Topic Search
F3	Search|Search Again
Ctrl+E	Search|Incremental Search
Ctrl+F	Search|Find
Ctrl+I	Inserts a tab character
Ctrl+N	Inserts a new line
Ctrl+P	Causes next character to be interpreted as an ASCII sequence
Ctrl+R	Search|Replace
Ctrl+S	File|Save
Ctrl+T	Deletes a word
Ctrl+Y	Deletes a line
Ctrl+Z	Edit|Undo
Ctrl+Shift+I	Indents block
Ctrl+Shift+U	Outdents block
Ctrl+Shift+Y	Deletes to the end of a line
Ctrl+Shift+Z	Edit|Redo
Alt+[	Finds the matching delimiter (forward)
Alt+]	Finds the matching delimiter (backward)
End	Moves to the end of a line
Home	Moves to the start of a line
Enter	Inserts a carriage return
Ins	Turns insert mode on/off
Del	Deletes the character to the right of the cursor
Backspace	Deletes the character to the left of the cursor
Tab	Inserts a tab
Space	Inserts a blank space
Left Arrow	Moves the cursor left one column, accounting for the autoindent setting
Right Arrow	Moves the cursor right one column, accounting for the autoindent setting
Up Arrow	Moves up one line
Down Arrow	Moves down one line
Page Up	Moves up one page
Page Down	Moves down one page
Ctrl+Left Arrow	Moves one word left
Ctrl+Right Arrow	Moves one word right
Ctrl+Tab	Moves to the next page
Ctrl+Shift+Tab	Moves to the previous page
Ctrl+Backspace	Deletes the word to the right of the cursor

Ctrl+Home	Moves to the top of a file
Ctrl+End	Moves to the end of a file
Ctrl+Del	Deletes a currently selected block
Ctrl+Space	Inserts a blank space
Ctrl+PgDn	Moves to the bottom of a screen
Ctrl+PgUp	Moves to the top of a screen
Ctrl+Up Arrow	Scrolls up one line
Ctrl+Down Arrow	Scrolls down one line
Ctrl+Enter	Opens file at cursor
Shift+Tab	Moves the cursor to the left one tab position
Shift+Backspace	Deletes the character to the left of the cursor
Shift+Left Arrow	Selects the character to the left of the cursor
Shift+Right Arrow	Selects the character to the right of the cursor
Shift+Up Arrow	Moves the cursor up one line and selects from the left of the starting cursor position
Shift+Down Arrow	Moves the cursor down one line and selects from the right of the starting cursor position
Shift+PgUp	Moves the cursor up one screen and selects from the left of the starting cursor position
Shift+PgDn	Moves the cursor down one line and selects from the right of the starting cursor position
Shift+End	Selects from the cursor position to the end of the current line
Shift+Home	Selects from the cursor position to the start of the current line
Shift+Space	Inserts a blank space
Shift+Enter	Inserts a new line with a carriage return
Ctrl+Shift+Left Arrow	Selects the word to the left of the cursor
Ctrl+Shift+Right Arrow	Selects the word to the right of the cursor
Ctrl+Shift+Home	Selects from the cursor position to the start of the current file
Ctrl+Shift+End	Selects from the cursor position to the end of the current file
Ctrl+Shift+PgDn	Selects from the cursor position to the bottom of the screen
Ctrl+Shift+PgUp	Selects from the cursor position to the top of the screen
Ctrl+Shift+Tab	Moves to the previous page
Alt+Backspace	Edit|Undo
Alt+Shift+Backspace	Edit|Redo
Alt+Shift+Left Arrow	Selects the column to the left of the cursor
Alt+Shift+Right Arrow	Selects the column to the right of the cursor
Alt+Shift+Up Arrow	Moves the cursor up one line and selects the column from the left of the starting cursor position
Alt+Shift+Down Arrow	Moves the cursor down one line and selects the column from the left of the starting cursor position
Alt+Shift+Page Up	Moves the cursor up one screen and selects the column from the left of the starting cursor position
Alt+Shift+Page Down	Moves the cursor down one line and selects the column from the right of the starting cursor position
Alt+Shift+End	Selects the column from the cursor position to the end of the current line
Alt+Shift+Home	Selects the column from the cursor position to the start of the current line
Ctrl+Alt+Shift+Left Arrow	Selects the column to the left of the cursor
Ctrl+Alt+Shift+Rght Arrow	Selects the column to the right of the cursor
Ctrl+Alt+Shift+Home	Selects the column from the cursor position to the start of the current file

Ctrl+Alt+Shift+End	Selects the column from the cursor position to the end of the current file
Ctrl+Alt+Shift+Page Up	Selects the column from the cursor position to the bottom of the screen
Ctrl+Alt+Shift+Page Down	Selects the column from the cursor position to the top of the screen

See also

Block commands
Bookmark operations
Cursor movement
Miscellaneous commands

System (Epsilon)

These system keyboard shortcuts apply to the Epsilon keystroke mapping scheme.

Shortcut	Action or command
F1	Displays context-sensitive Help
F5	Toggle Breakpoint
F8	Run|Step Over
F9	Run|Run
F10	Edit|Redo
F11	View|Object Inspector
F12	View|Toggle Form/Unit
Alt+F9	Project|Compile
Alt+F10	Displays a SpeedMenu
Alt+0	View|Window List
Ctrl+F2	Run|Program Reset
Ctrl+F5	Run|Add Watch
Ctrl+F6	Displays the next page
Ctrl+Shift+F6	Displays the previous page
Ctrl+F7	File|Save As
Ctrl+F9	Project|Compile
Ctrl+F12	View|Units
Shift+F12	View|Forms
Ctrl+X+(	Records a keyboard macro
Ctrl+X+)	Ends a keyboard macro recording
Ctrl+X+e	Plays back the the last keyboard macro recorded
Ctrl+X+E	Plays back the the last keyboard macro recorded
Ctrl+X+b	Displays a list of open files
Ctrl+X+B	Displays a list of open files
Ctrl+X+s	File|Save As
Ctrl+X+S	File|Save As
Ctrl+X+Ctrl+F	File|Open
Ctrl+X+Ctrl+S	File|Save
Ctrl+X+Ctrl+W	File|Save

Clipboard control (Epsilon)

These Clipboard keyboard shortcuts apply to the Epsilon keystroke mapping scheme.

Shortcut	Action or command
Ctrl+Y	Edit|Paste
Alt+w	Edit|Copy
Esc+@w	Edit|Copy
Ctrl+Alt+w	Edit|Copy (appends to current contents)
Esc+Ctrl+w	Edit|Copy (appends to current contents)

Editor (Epsilon)

See also

These editor keyboard shortcuts apply to the Epsilon keystroke mapping scheme.

Shortcut	Action or command
Ctrl+H	Deletes the character to the left of the current cursor position
Backspace	Deletes the character to the left of the current cursor position
Alt+Del	Deletes all text in the block between the cursor and the previous matching delimiter (cursor must be on ')', '}' or ']')
Esc+Del	Deletes all text in the block between the cursor and the previous matching delimiter (cursor must be on ')', '}' or ']')
Ctrl+Alt+H	Deletes the word to the left of the current cursor position
Alt+Backspace	Deletes the word to the left of the current cursor position
Esc+BackSpace	Deletes the word to the left of the current cursor position
Esc+Ctrl+H	Deletes the word to the left of the current cursor position
Ctrl+D	Deletes the currently selected character or character to the right of the cursor
Del	Deletes the currently selected character or character to the right of the cursor
Alt+	Deletes spaces and tabs around the cursor on the same line
Esc+	Deletes spaces and tabs around the cursor on the same line
Ctrl+Alt+K	Deletes all text in the block between the cursor and the next matching delimiter (cursor must be on ')', '}' or ']')
Esc+Ctrl+K	Deletes all text in the block between the cursor and the next matching delimiter (cursor must be on ')', '}' or ']')
Ctrl+X+0	Deletes the contents of the current window
Alt+d	Deletes to word to the right of the cursor
Esc+@d	Deletes to word to the right of the cursor
Ctrl+K	Cuts the contents of line and places it in the Clipboard
Ctrl+Alt+B	Locates the next matching delimiter (cursor must be on ')', '}' or ']')
Esc+Ctrl+B	Locates the next matching delimiter (cursor must be on ')', '}' or ']')
Alt+)	Locates the next matching delimiter (cursor must be on ')', '}' or ']')
Esc+)	Locates the next matching delimiter (cursor must be on ')', '}' or ']')
Alt+Shift+O	Locates the next matching delimiter (cursor must be on ')', '}' or ']')
Ctrl+Alt+F	Locates the previous matching delimiter (cursor must be on ')', '}' or ']')
Esc+Ctrl+F	Locates the previous matching delimiter (cursor must be on ')', '}' or ']')
Alt+c	Capitalizes the current word
Esc+@c	Capitalizes the current word
Ctrl+L	Centers the active window
Ctrl+M	Inserts a carriage return
Ctrl+X+i	Inserts the contents of a file at the cursor
Ctrl+X+I	Inserts the contents of a file at the cursor
Ctrl+O	Inserts a new line after the cursor
Alt+x	Invokes the specified command or macro
Esc+@x	Invokes the specified command or macro
F2	Invokes the specified command or macro
Ctrl+X+Ctrl+X	Exchanges the locations of the cursor position and a bookmark
Ctrl+Shift+-	Displays context-sensitive Help
Alt+Shift+/	Displays context-sensitive Help
Alt+?	Displays context-sensitive Help
Esc+?	Displays context-sensitive Help
Ctrl+_	Displays context-sensitive Help

Ctrl+X+,	Browses the symbol at the cursor
Tab	Inserts a tab
Alt+Tab	Indents to the current line to the text on the previous line
Esc+Tab	Indents to the current line to the text on the previous line
Alt+l	Converts the current word to lowercase
Esc+@l	Converts the current word to lowercase
Ctrl+X+m	Project|Compile
Ctrl+X+M	Project|Compile
Esc+End	Displays the next window in the buffer list
Ctrl+X+n	Displays the next window in the buffer list
Ctrl+X+N	Displays the next window in the buffer list
Esc+Home	Displays the previous window in the buffer list
Ctrl+X+p	Displays the previous window in the buffer list
Ctrl+X+P	Displays the previous window in the buffer list
Ctrl+X+Ctrl+E	Invoke a command processor
Ctrl+Q	Interpret next character as an ASCII code
Ctrl+X+r	Edit|Redo
Ctrl+X+R	Edit|Redo
F10	Edit|Redo
Ctrl+F10	Edit|Redo
Ctrl+X+Ctrl+R	Edit|Redo
Ctrl+X+u	Edit|Undo
Ctrl+X+U	Edit|Undo
F9	Edit|Undo
Ctrl+F9	Edit|Undo
Ctrl+X+Ctrl+U	Edit|Undo
Ctrl+S	Incrementally searches for a string entered from the keyboard
Ctrl+R	**I**ncrementally searches backward through the current file
Ctrl+Alt+S	Search|Find (using regular expressions)
Esc+Ctrl+S	Search|Find (using regular expressions)
Ctrl+Alt+R	Search|Find (using regular expressions; backward from cursor)
Esc+Ctrl+R	Search|Find (using regular expressions; backward from cursor)
Alt+Shift+5	Search|Replace
Alt+Shift+7	Search|Replace
Alt+&	Search|Replace
Esc+&	Search|Replace
Alt+%	Search|Replace
Esc+%	Search|Replace
Alt+*	Search|Replace (using regular expressions)
Esc+*	Search|Replace (using regular expressions)
Ctrl+X+Ctrl+N	Search|Find Error
Ctrl+X+g	Search|Go To Line Number
Ctrl+X+G	Search|Go To Line Number
Crl+T	Transposes the two characters on either side of the cursor
Ctrl+X+Ctrl+T	Transposes the two lines on either side of the cursor
Alt+t	Transposes the two words on either side of the cursor
Esc+t	Transposes the two words on either side of the cursor
Esc+T	Transposes the two words on either side of the cursor
Alt+U	Converts a word to all uppercase

Esc+U	Converts a word to all uppercase
Esc+@u	Converts a word to all uppercase
Ins	Toggles insert mode on/off

See also

Block commands

Bookmark operations

Cursor movement

Block commands (Epsilon)

See also

These block command keyboard shortcuts apply to the Epsilon keystroke mapping scheme.

Shortcut	Action
Ctrl+Alt+	Indents a block
Esc+Ctrl+	Indents a block
Ctrl+X+Ctrl+I	Indents a block
Ctrl+X+Tab	Indents a block
Ctrl+W	Cuts a block and places its contents in the Clipboard
Ctrl+X+w	Writes a block to a file
Ctrl+X+W	Writes a block to a file

See also

Bookmark operations

Cursor movement

Editor

Bookmark operations (Epsilon)

See also

These bookmark operations keyboard shortcuts apply to the Epsilon keystroke mapping scheme.

Shortcut	Action
Ctrl+@	Sets a bookmark at the current cursor position
Alt+@	Sets a bookmark at the current cursor position
Esc+@@	Sets a bookmark at the current cursor position
Ctrl+2	Sets a bookmark at the current cursor position
Alt+2	Sets a bookmark at the current cursor position
Ctrl+X+.	Goes to the specified bookmark

See also

Block commands

Cursor movement

Editor

Cursor movement (Epsilon)

See also

These cursor movement keyboard shortcuts apply to the Epsilon keystroke mapping scheme.

Shortcut	Action
Ctrl+B	Moves to the left one character
Left Arrow	Moves to the left one character
Ctrl+F	Moves to the right one character
RightArrow	Moves to the right one character
Alt+m	Moves the cursor to the end of the indentation
Esc+m	Moves the cursor to the end of the indentation
Esc+M	Moves the cursor to the end of the indentation
Alt+b	Moves the cursor to the left one word
Esc+@b	Moves the cursor to the left one word
Ctrl+LeftArrow	Moves the cursor to the left one word
Alt+f	Moves to the cursor to the right one word
Esc+@f	Moves to the cursor to the right one word
Ctrl+RgAr	Moves to the cursor to the right one word
Ctrl+A	Moves to the beginning of the current line
Esc+LeftArrow	Moves to the beginning of the current line
Ctrl+E	Moves to the end of the current line
Esc+RightArrow	Moves to the end of the current line
Alt-,	Moves to the top of the current window
Esc+,	Moves to the top of the current window
Home	Moves to the top of the current window
Alt-.	Moves to the bottom of the current window
Esc+.	Moves to the bottom of the current window
End	Moves to the bottom of the current window
Ctrl+P	Moves the cursor up a line
UpAr	Moves the cursor up a line
Ctrl+N	Moves the cursor down a line
DnAr	Moves the cursor down a line
Alt+Shift-,	Goes to the start of the file
Alt+<	Goes to the start of the file
Esc+<	Goes to the start of the file
Ctrl+Home	Goes to the start of the file
Alt+Shift-.	Goes to the end of the file
Alt+>	Goes to the end of the file
Esc+>	Goes to the end of the file
Ctrl+End	Goes to the end of the file
Ctrl+V	Moves down one page in the current file
PgDn	Moves down one page in the current file
Ctrl+F6	Moves down one page in the current file
Shift+Ctrl+F6	Moves up one page in the current file
Alt+v	Moves up one page in the current file
Esc+@v	Moves up one page in the current file
PgUp	Moves up one page in the current file
Alt+Z	Scrolls the contents of the active window down a line
Esc+Z	Scrolls the contents of the active window down a line
Ctrl+Z	Scrolls the contents of the active window up a line

See also

Block commands

Bookmark operations

Editor

About keyboard shortcuts

See also

Keyboard shortcuts are two- or three-keystroke combinations you can press, while in the Code Editor, to perform a command or access a dialog box. The function of specific keyboard shortcuts depends on which keystroke mapping scheme you select.

Code Editor available keyboard mapping schemes are:

Default	Key bindings that match the CUA standard
Classic	Key bindings that match the Delphi programming environment
Brief	Key bindings that emulate most of the standard Brief keystrokes
Epsilon	Key bindings that emulate a a large part of the Epsilon editor

To select a keymapping,

1. On the Editor display page of the Environment Options dialog box.
2. Select a keyboard mapping scheme from the list of available schemes.
3. Click OK.

To use SpeedSettings to set your keymappings,

1. On the Editor options page of the Environment Options dialog box.
2. Select a keyboard mapping scheme from the Editor SpeedSettings options.
3. Click OK.

Note: Using the Keystroke Mapping list box or the Editor SpeedSettings to change the mapping of your keystrokes can create conflicts with standard Windows keyboard commands.

For example, the Brief keystroke mapping defines Alt+E as File|Open, while the standard Windows action for Alt+E is to activate the Edit menu. The mapped key takes precedence so that Alt+E allows you to open a file.

See also

IDE keyboard shortcuts

Keyboard shortcuts by function

Using the Code Editor

Default keystroke mapping

The Default keystroke mapping scheme provides key bindings that match the CUA standard. For detailed information, choose one of the topics below for a list of keyboard shortcuts:

Clipboard control

Debugger

Editor

- Block commands
- Bookmark operations
- Cursor movement
- Miscellaneous commands

System

Classic keystroke mapping

The Classic keystroke mapping scheme provides key bindings that match the Delphi programming enviroment. For detailed information, choose one of the topics below for a list of keyboard shortcuts:

Clipboard control

Debugger

Editor

- Block commands
- Bookmark operations
- Cursor movement
- Miscellaneous commands

System

Brief keystroke mapping

The Brief keystroke mapping scheme provides key bindings that emulate the Brief editor. For detailed information, choose one of the topics below for a list of keyboard shortcuts:

Epsilon keystroke mapping

The Epsilon keystroke mapping scheme provides key bindings that emulate most of the Epsilon editor. For detailed information, choose one of the topics below for a list of keyboard shortcuts:

Debugger (default, classic, brief and Epsilon)

The Debugger keyboard shortcuts apply to all keystroke mapping schemes:

Default

Classic

Brief

Epsilon

Breakpoint view	
Ctrl+E	Edit Breakpoint
Ctrl+V	View Source
Ctrl+S	Edit Source
Ctrl+D	Delete Breakpoint
Ctrl+A	Add Breakpoint

Call stack view	
Ctrl+V	View Source
Ctrl+E	Edit Source

Watch view	
Ctrl+E	Edit Watch
Ctrl+A	Add Watch
Ctrl+D	Delete Watch

Block commands (default and classic)

See also

These block command shortcuts apply to the Default and the Classic keystroke mappings schemes.

Shortcut	Action or command
Ctrl+K+B	Marks the beginning of a block
Ctrl+K+C	Copies a selected block
Ctrl+K+H	Hides/shows a selected block
Ctrl+K+I	Indents a block by the amount specified in the Block Indent combo box on the Editor options page of the Environment Options dialog box
Ctrl+K+K	Marks the end of a block
Ctrl+K+L	Marks the current line as a block
Ctrl+K+N	Changes a block to uppercase
Ctrl+K+O	Changes a block to lowercase
Ctrl+K+R	Reads a block from a file
Ctrl+K+T	Marks a word as a block
Ctrl+K+U	Outdents a block by the amount specified in the Block Indent combo box on the Editor options page of the Environment Options dialog box.
Ctrl+K+V	Moves a selected block
Ctrl+K+W	Writes a selected block to a file
Ctrl+K+Y	Deletes a selected block
Ctrl+O+C	Marks a column block
Ctrl+O+I	Marks an inclusive block
Ctrl+O+K	Marks a non-inclusive block
Ctrl+O+L	Marks a line as a block
Ctrl+Q+B	Moves to the beginning of a block
Ctrl+Q+K	Moves to the end of a block

See also

- Bookmark operations
- Cusor movement
- Editor (classic)
- Editor (default)
- Miscellaneous commands

Bookmark operations (default and classic)

See also

The following bookmark operations shortcuts apply to the Default and the Classic keystroke mapping schemes.

Shortcut	Action
Ctrl+K+0	Sets bookmark 0
Ctrl+K+1	Sets bookmark 1
Ctrl+K+2	Sets bookmark 2
Ctrl+K+3	Sets bookmark 3
Ctrl+K+4	Sets bookmark 4
Ctrl+K+5	Sets bookmark 5
Ctrl+K+6	Sets bookmark 6
Ctrl+K+7	Sets bookmark 7
Ctrl+K+8	Sets bookmark 8
Ctrl+K+9	Sets bookmark 9
Ctrl+K+Ctrl+0	Sets bookmark 0
Ctrl+K+Ctrl+1	Sets bookmark 1
Ctrl+K+Ctrl+2	Sets bookmark 2
Ctrl+K+Ctrl+3	Sets bookmark 3
Ctrl+K+Ctrl+4	Sets bookmark 4
Ctrl+K+Ctrl+5	Sets bookmark 5
Ctrl+K+Ctrl+6	Sets bookmark 6
Ctrl+K+Ctrl+7	Sets bookmark 7
Ctrl+K+Ctrl+8	Sets bookmark 8
Ctrl+K+Ctrl+9	Sets bookmark 9
Ctrl+Q+0	Goes to bookmark 0
Ctrl+Q+1	Goes to bookmark 1
Ctrl+Q+2	Goes to bookmark 2
Ctrl+Q+3	Goes to bookmark 3
Ctrl+Q+4	Goes to bookmark 4
Ctrl+Q+5	Goes to bookmark 5
Ctrl+Q+6	Goes to bookmark 6
Ctrl+Q+7	Goes to bookmark 7
Ctrl+Q+8	Goes to bookmark 8
Ctrl+Q+9	Goes to bookmark 9
Ctrl+Q+Ctrl+0	Goes to bookmark 0
Ctrl+Q+Ctrl+1	Goes to bookmark 1
Ctrl+Q+Ctrl+2	Goes to bookmark 2
Ctrl+Q+Ctrl+3	Goes to bookmark 3
Ctrl+Q+Ctrl+4	Goes to bookmark 4
Ctrl+Q+Ctrl+5	Goes to bookmark 5
Ctrl+Q+Ctrl+6	Goes to bookmark 6
Ctrl+Q+Ctrl+7	Goes to bookmark 7
Ctrl+Q+Ctrl+8	Goes to bookmark 8
Ctrl+Q+Ctrl+9	Goes to bookmark 9

These shortcuts apply only to the Default scheme:

Shortcut	Action
Shift+Ctrl+0	Sets bookmark 0
Shift+Ctrl+1	Sets bookmark 1
Shift+Ctrl+2	Sets bookmark 2
Shift+Ctrl+3	Sets bookmark 3

Shift+Ctrl+4	Sets bookmark 4
Shift+Ctrl+5	Sets bookmark 5
Shift+Ctrl+6	Sets bookmark 6
Shift+Ctrl+7	Sets bookmark 7
Shift+Ctrl+8	Sets bookmark 8
Shift+Ctrl+9	Sets bookmark 9

Ctrl+0	Goes to bookmark 0
Ctrl+1	Goes to bookmark 1
Ctrl+2	Goes to bookmark 2
Ctrl+3	Goes to bookmark 3
Ctrl+4	Goes to bookmark 4
Ctrl+5	Goes to bookmark 5
Ctrl+6	Goes to bookmark 6
Ctrl+7	Goes to bookmark 7
Ctrl+8	Goes to bookmark 8
Ctrl+9	Goes to bookmark 9

See also

Block commands

Cursor movement

Editor (classic)

Editor (default)

Miscellaneous commands

Cursor movement (default and classic)

See also

These cursor movement shortcuts apply to the Default and the Classic keystroke mapping schemes.

Shortcut	Action
Ctrl+Q+B	Moves to the beginning of a block
Ctrl+Q+C	Moves to end of a file
Ctrl+Q+D	Moves to the end of a line
Ctrl+Q+E	Moves to the top of the window
Ctrl+Q+K	Moves to the end of a block
Ctrl+Q+P	Moves to previous position
Ctrl+Q+R	Moves to the beginning of a file
Ctrl+Q+S	Moves to the beginning of a line
Ctrl+Q+T	Moves to the top of the window
Ctrl+Q+U	Moves to the bottom of the window
Ctrl+Q+X	Moves to the bottom of the window

See also

Block commands

Bookmark operations

Editor (classic)

Editor (default)

Miscellaneous commands

Miscellaneous commands (default and classic)

See also

These miscellaneous commands shortcuts apply to the Default and the Classic keystroke mapping schemes.

Shortcut	Action or command
Ctrl+K+D	Accesses the menu bar
Ctrl+K+E	Changes a word to lowercase
Ctrl+K+F	Changes a word to uppercase
Ctrl+K+S	File|Save
Ctrl+Q+A	Search|Replace
Ctrl+Q+F	Search|Find
Ctrl+Q+Y	Deletes to the end of a line
Ctrl+Q+[	Finds the matching delimiter (forward)
Ctrl+Q+Ctrl+[	Finds the matching delimiter (forward)
Ctrl+Q+]	Finds the matching delimiter (backward)
Ctrl+Q+Ctrl+]	Finds the matching delimiter (backward)
Ctrl+O+A	Open file at cursor
Ctrl+O+B	Browse symbol at cursor
Ctrl+O+G	Search|Go to line number
Ctrl+O+O	Inserts compiler options and directives
Ctrl+O+U	Toggles case

See also

Block commands

Bookmark operations

Cursor movement

Editor (classic)

Editor (default)

Keyboard shortcuts by function

See also

Chose one of these topics for shortcuts for some common menu commands. The shortcuts are different for each keystroke mapping scheme.

Build commands

Debug commands

Edit commands

File commands

Search commands

See also
Keyboard shortcuts
Using the Code Editor

Keyboard shortcuts for the File menu

See also

The table below lists keyboard shortcuts for file commands.

Command	Shortcut	Mapping
File|Open	F3	Classic
	Alt+E	Brief
	Ctrl+X+Ctrl+F	Epsilon
Open File At Cursor	Ctrl+O+A	Default, Classic, Brief
File|Save	Ctrl+K+S	Default, Classic
	Ctrl+S	Default
	F2	Classic
	Alt+W	Brief
	Ctrl+X+Ctrl+S	Epsilon
	Ctrl+X+Ctrl+W	Epsilon
File|Save As	Alt+O	Brief
	Ctrl+F7	Epsilon
	Ctrl+X+s	Epsilon
	Ctrl+X+S	Epsilon
File|Close	Alt+F3	Classic
	Ctrl+Hyphen	Brief
Control|Close	Alt+F4	Default, Classic, Brief, Epsilon
File|Exit	Alt+X	Classic, Brief
File menu	Alt+Z	Brief

For more information about Delphi's keystroke mapping schemes, choose one of the following topics:

Default keystroke mapping

Classic keystroke mapping

Brief keystroke mapping

Epsilon keystroke mapping

See also

Build commands
Debug commands
Edit commands
Keyboard shortcuts
Search commands

Keyboard shortcuts for the Edit menu

See also

The table below lists the keyboard shortcuts for commands on the Edit menu.

Command	Shortcut	Mapping
Edit|Cut	Shift+Del	Default, Classic
	Ctrl+X	Default
	Minus (-)	Brief
Edit|Copy	Ctrl+Ins	Default, Classic
	Ctrl+C	Default
	Plus (+)	Brief
	Alt+w	Epsilon
	Esc+@w	Epsilon
	Ctrl+Alt+w	Epsilon
	Esc+Ctrl+w	Epsilon
Edit|Paste	Shift+Ins	Default, Classic
	Ctrl+V	Default
	Ins	Brief
	Ctrl+Y	Epsilon
Edit|Redo	Ctrl+Shift+Z	Default
	Alt+Shift+Backspace	Default, Classic
	Ctrl+U	Brief
	Ctrl+X+r	Epsilon
	Ctrl+X+R	Epsilon
	F10	Epsilon
	Ctrl+F10	Epsilon
	Ctrl+X+Ctrl+R	Epsilon
Edit|Undo	Alt+Backspace	Default, Classic
	Star	Brief
	Alt+U	Brief
	Ctrl+X+u	Epsilon
	Ctrl+X+U	Epsilon
	F9	Epsilon
	Ctrl+F9	Epsilon
	Ctrl+X+Ctrl+U	Epsilon

For more information about Delphi's keystroke mapping schemes, choose one of the following topics:

Default Keystroke Mapping

Classic Keystroke Mapping

Brief Keystroke Mapping

Epsilon Keystroke Mapping

See also

Build commands
Debug commands
File commands
Keyboard shortcuts
Search commands

Search commands keyboard shortcuts

See also

The table below lists the keyboard shortcuts for commands on the Search menu.

Command	Shortcut	Mapping
Search|Find	Ctrl+Q+F	Default, Classic
	Ctrl+F	Default
	F5	Brief
	Alt+F5	Brief
	Alt+S	Brief
	Ctrl+Alt+S	Epsilon
	Esc+Ctrl+S	Epsilon
	Ctrl+Alt+R	Epsilon
	Esc+Ctrl+R	Epsilon
Search|Replace	Ctrl+Q+A	Default, Classic
	Ctrl+R	Default
	Alt+T	Brief
	F6	Brief
	Alt+F6	Brief
	Alt+&	Epsilon
	Esc+&	Epsilon
	Alt+%	Epsilon
	Esc+%	Epsilon
	Alt+*	Epsilon
	Esc+*	Epsilon
Search|Search Again	F3	Default
	Ctrl+L	Classic
	Shift+F5	Brief
Search|Go To Line Number	Ctrl+O+G	Default, Classic, Brief
	Alt+G	Brief
	Ctrl+X+g	Epsilon
	Ctrl+X+G	Epsilon

For more information about Delphi's keystroke mapping schemes, choose one of the following topics:

Default Keystroke Mapping

Classic Keystroke Mapping

Brief Keystroke Mapping

Epsilon Keystroke Mapping

See also

Build commands

Debug commands

Edit commands

File commands

Keyboard shortcuts

Debug commands keyboard shortcuts

See also

The table below lists the keyboard shortcuts for debug operations.

Command	Shortcut	Mapping
Run|Go to Cursor	F4	Default, Classic
Run|Toggle Breakpoint	F5	Default
Run|Trace Into	F7	Default, Classic
Run|Step Over	F8	Default, Classic, Epsilon
Run|Program Reset	Ctrl+F2	Default, Classic, Brief, Epsilon
Run|Add Watch	Ctrl+F5	Epsilon
Add Watch at Cursor	Ctrl+F5	Default
	Ctrl+F7	Classic
	Alt+F2	Brief
Browse Symbol at Cursor	Ctrl+O+B	Default, Classic, Brief
Evaluate/Modify	Ctrl+F7	Default, Brief
	Ctrl+F4	Classic
Toggle Breakpoint	Ctrl+F8	Classic, Brief
	F5	Epsilon
Run to Cursor	F4	Classic
Run to Cursor	Alt+F7	Brief

For more information about Delphi's keystroke mapping schemes, choose one of the following topics:

Default Keystroke Mapping

Classic Keystroke Mapping

Brief Keystroke Mapping

Epsilon Keystroke Mapping

See also

Build commands
Edit commands
File commands
Keyboard shortcuts
Search commands

Build commands keyboard shortcuts

See also

The table below lists the keyboard shortcuts for build operations.

Command	Shortcut	Mapping
Project|Compile	Ctrl+F9	Default, Classic, Brief
	Alt+F9	Default, Classic, Epsilon
	Alt+F10	Brief
	Ctrl+X+m	Epsilon
	Ctrl+X+M	Epsilon
Run|Run	F9	Default, Classic, Brief, Epsilon

For more information about Delphi's keystroke mapping schemes, choose one of the following topics:

Default Keystroke Mapping

Classic Keystroke Mapping

Brief Keystroke Mapping

Epsilon Keystroke Mapping

See also

Debug commands
Edit commands
File commands
Keyboard shortcuts
Search commands

Keyboard support in the Delphi IDE

See also

Delphi IDE keyboard shortcuts are two- or three-keystroke combinations you can press to perform a command or access a dialog box directly without having to open any menu. To learn about shortcuts in the Code Editor window see Keyboard shortcuts.

To learn about shortcuts in the other windows, select one of the topics listed below:

Form keyboard shortcuts

Project Manager keyboard shortcuts

Object Inspector keyboard shortcuts

Browser keyboard shortcuts

See also

Keyboard shortcuts by function

Using the Code Editor

Form keyboard shortcuts

See also

Listed below are keyboard shortcuts for working with forms.

The IDE supports the movement and resizing of components on a form using the keyboard. The following table shows the keystrokes for selection and move and resize operations. Remember that you must select a component in order to move or resize it.

Keyboard command	Description
Tab	Selects components (in z-order)
Shift+Tab	Selects the previous component (in z-order)
Arrow keys	Selects the nearest component in the direction pressed
Ctrl+arrow keys	Moves the selected component one pixel at a time
Shift+arrow keys	Moves the selected component one pixel at a time
Ctrl+Shift+arrow keys	Moves the selected component one grid at a time (when Snap to Grid is enabled)
Del	Deletes the selected component
Esc	Selects the containing group (usually the form or group box)
F11	Toggles control between the Object Inspector and the last active form or unit
F12	Toggles between the form and its associated unit
Ctrl+F12	Displays the View Unit dialog box
Shift+F12	Displays the View Form dialog box

To add components to a form using the keyboard,

1. Press Alt+V+L to display the Component List dialog box
2. Type the first letter of the name of the component you want to place on the form or press Tab. Then you can use the arrow keys to scroll through the list and make a selection.
3. Press Alt+A or Enter to add the component to the form. Pressing Enter will close the Component List dialog box.

Keys to navigate in the component list	
Home	Displays the first component in the list
End	Displays the last component in the list

To change properties of a component using the keyboard,

1. Select the component you want to modify using Tab or the arrow keys.
2. Press Enter to switch to the Object Inspector.
3. Use the arrow keys to select the property you want to change.
4. Type the new value for that property and press Enter.
5. To return to the form, press Alt+V+F and select it from the list.

See also
Keyboard shortcuts

Project Manager keyboard shortcuts

Listed below are keyboard shortcuts for working with the Project Manager.

Keyboard command	Description
Arrow keys	Selects forms and units
Alt+A	Adds a form or unit to the project
Alt+R	Removes a form or unit from the project
Alt+U	Views the selected unit
Alt+F	Views the selected form
Alt+O	Displays the Project Options dialog box
Alt+D	Updates the current project
Enter	Views the selected unit
Shift+Enter	Views the selected form
Ins	Adds a file to the project
Del	Removes a file from the project

Object Inspector keyboard shortcuts

Listed below are keyboard shortcuts for working with the Object Inspector.

Keyboard command	Description
Ctrl+I	Opens the Object Selector
Up and Down Arrow keys	Selects properties or handlers
Left and Right Arrow keys	Edits the value in the value or event column
Tab	Toggles between the property and value columns in the Object Inspector
Tab+<letter>	Jumps directly to the first property beginning with the letter
Ctrl+Tab	Toggles between the properties and events tabs in the Object Inspector
Page Up	Moves up one screen of properties
Page Down	Moves down one screen of properties
Alt+F10	Toggles expand and contract
Alt+Down	Opens a drop-down list for a property.
Ctrl-Down	Opens the object list drop-down.
Ctrl-Enter	Selects the ellipsis button (if available) in a selected property.

To change properties of a component using the keyboard,

1. Select the component you want to modify using Tab or the arrow keys.
2. Press Enter to switch to the Object Inspector.
3. Use the arrow keys to select the property you want to change.
4. Type the new value for that property and press Enter.
5. To return to the form, press Alt+V+F and select it from the list.

Component menu

Use the Component menu to build a component, install a new component, rebuild the component library or configure the Component palette.

The options on the Component menu are:

New	Opens the Component Expert
Install	Installs new components
Open Library	Opens component library file
Rebuild Library	Recompiles the component library
Configure Palette	Opens the Palette dialog box

Component | New

Choose Component|New to open the Component Expert.

Component Expert
Use this expert to create the basic unit for a new component.

Class Name
Enter the name of the new class you are creating. A general rule is that all Object Pascal classes are prefaced with a T. For example, the name of your new button component could be TMYBUTTON.

Ancestor Class
Use the drop-down list to select a base class, or enter the name of a base class for your new component. Unless you override them in the component declaration, your new component will inherit all the properties, methods, and events from its ancestor class.

Palette Page
Use the drop-down list to select a page, or enter the name of the page on which you want your new component to appear when you add it to the library.

Component | Install

See also

Use Component|Install to display the Install Components dialog box.

Install Components
Use this dialog box to install new components. If a project is open when you choose Component|Install, Delphi prompts you to save any changes made to the current project when you choose the OK button in this dialog box.

Library Filename
Enter the name of the library file you want to load.

Search Path
Enter the path you want the compiler to search when the component library is rebuilt.

Installed Units
Displays the units that are already installed. When you select a unit, its components are displayed in the Component Classes list box.

Component Classes
Displays the components that are installed in the unit selected in the Installed Units list box.

Add
Click Add to add a new unit. This button displays the Add Module dialog box, where you enter the new unit name.

OCX
Click OCX to add a OLE control to the Component palette. This button displays the Import OLE Control dialog box, where you can select an OCX control to install.

Remove
Click Remove to remove the selected unit and its components from the Component palette.

Revert
Click Revert to discard your changes if you get an error while building the library. The file will revert to its previous state.

See also

Adding components to the component library

Removing components from the component library

Add Module dialog box

Use the Add Module dialog box to add a new unit.

To open this dialog box,
Click the Add button in the Install Component dialog box.

Module Name
Enter the unit name in the Module Name edit box.

Browse
Click Browse to use the Add Module Browse dialog box to search for the name of the unit you want to install.

Add Module Browse dialog box

Use the Add Module Browse dialog box to search drives and directories to find the file you want to add.

To open this dialog box,
Click the Browse button on the Add Module dialog box.

File Name
Enter the name of the file you want to load, or enter wildcards to use as filters in the Files list box.

Files
Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type
Choose the type of file you want to open. The default file type is .PAS. All files in the current directory of the selected type appear in the Files list box.

Directories
Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives
Select the current active drive. The directory structure for the current drive appears in the Directories list box.

Import OLE Control dialog box

See also

Use the Import OLE Control dialog box to specify the OCX control you want to install.

To open this dialog box,

Click the OCX button on the Install Components dialog box.

Registered controls

Displays the names of the OLE control libraries that are registered on your system. To select an OLE control library for import, click on the entry in the listbox. When you select a control library, the name of the corresponding .OCX file is shown below the listbox.

Register button

Click on this button to register a new OLE control. An OLE control must be registered before you can import it into Delphi.

Unregister button

Unregisters the currently selected OLE control by removing it from the system registry. Unregistering an OLE control does not actually delete the control, it merely removes its registration information from the system registry.

Unit File name

The name of the import unit generated by Delphi. When you import an OLE control library, Delphi generates an interface unit that contains a class declaration for each OLE control in the library. By default, the import unit is placed in the first directory listed in the library search path.

Browse button

Allows you to browse for a directory in which to place the import unit.

Palette Page

Specifies on which page of the Component palette, Delphi will install the OCX control. The OCX page of the Component palette is the default page for OCX controls. If you enter the name of a page that does not exist, Delphi creates a new page with that name on the Component palette.

Class names

Displays the suggested class names of the OLE controls found in the selected OLE control library. Unless another OLE control library uses the same class names, you should have no reason to change these. If you do make changes, it is strongly suggested that you start each class name with a “T” as is the Delphi convention.

OK and Cancel buttons

When you press OK, Delphi generates an import unit by the specified name and returns to the “Install Components” dialog box, adding the newly generated unit to the list of installed units.

To change the class name,

1. Select the class name.
2. Click the Edit button to open the Edit Class Name dialog box, where you can change the class name.

See also

Installing Visual Basic components

Components|Install

Install OCX dialog box

See also

Use the Install OCX dialog box to specify on which page of the Component palette you want to install the OCX control, in which directory you want to store its associated unit, and a class name for the control.

To open this dialog box,
Click the OCX button on the Install Components dialog box.

To register an OCX control,
Select a OCX file to install and click OK.

See also

Installing Visual Basic components

Install OCX dialog box

Component|Install

Edit Class Name dialog box

Use the Edit Class Name dialog box to change the class name of the OCX control you are installing.

To open this dialog box,
Select a name from the Class Names list box in the Install OCX dialog box and click Edit.

Dialog box option

Class Name
Enter the new class name to use for the OCX control.

Unit File Name dialog box

See also

Use the Unit File Name dialog box to change the directory where Delphi stores the OCX unit file.

To open this dialog box,
Click the Browse button in the Install OCX dialog box.

File Name
Enter the name of the OCX unit file you want to save.

Files
Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the Save File As Type combo box.

Save File As Type
Choose the type of file you want to save. The default file type is a unit file (*.PAS). All files in the current directory of the selected type appear in the Files list box.

Directories
Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the Save File As Type combo box appear in the Files list box.

Drives
Select the current active drive. The directory structure for the current drive appears in the Directories list box.

See also
Installing Visual Basic components
Component|Install

Component | Import OLE Control

Choose Import OLE control to open the Import OLE Control dialog box.

Component | Open Library

See also

Choose Component|Open Library to display the Open Library dialog box.

Open Library dialog box
You use this dialog box to install a dynamic component library (.DCL). If a project is open when you choose Component|Open Library, Delphi prompts you to save any changes made to the current project.

File Name
Enter the name of the file you want to load, or enter wildcards to use as filters in the Files list box.

Files
Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type
Choose the type of file you want to open. The default file type is Dynamic Component Library file (.DCL). All files in the current directory of the selected type appear in the Files list box.

Directories
Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives
Select the current active drive. The directory structure for the current drive appears in the Directories list box.

See also

Saving a customized library

Loading a component library

Component | Rebuild Library

Choose Component|Rebuild Library to recompile the existing component library without leaving the Delphi environment.

In order to recompile the library, Delphi must close the open project. Therefore, before Delphi recompiles the library, Delphi prompts you to save any changes you might have made to the open project.

Database menu

The Database menu contains commands that enable you to create, modify and view your databases.'
Here is a list of the commands on the Database menu:

Explore
SQL Monitor
Database Form Expert

Database | Explore

Choose Database|Explore to open the Database Explorer.

Database | SQL Monitor

Choose Database|SQL Monitor to open the SQL Monitor.

Database | Database Form Expert

See also

Choose Database|Database Form Expert to use the Delphi Database Form Expert to create a form that displays data from a local or remote database.

See also
Using the Database Form Expert

Using the Database Form Expert

See also

The Database Form Designer enables you to easily generate a form that displays data from an external InterBase, Paradox, dBase, or ORACLE database.

The Database Form expert helps you create two types of database forms:

- Simple database forms
- Master/detail forms

The tool automates such form building tasks as:

- Connecting the form to Table and Query components
- Writing SQL statements for Query components
- Placing interactive and non-interactive components on a form
- Defining a tab order
- Connecting DataSource components to interactive components and Table/Query components

See also

Creating a form using the Database Form Expert

Creating a form using the Database Form Expert

See also

You can use the Database Form Expert to create a simple information form.

To build a form using the Database Form Expert,

1. Open the Database Form Expert by choosing Database|Database Form Expert.
2. Select a Form Option and click Next.
3. From the Drive or Alias Name list, select an alias. This alias points to the Delphi sample database. To create an alias, see

   Note: If you have not created an alias, you can still enter a database name by specifying the path to a database in the Form Expert dialog box.
4. Select the fields to use on the generated form.

To use only some of the fields,

1. Press and hold Ctrl.
2. Select each field you want from the Available Fields list.
3. Choose the > button.

To use all of the fields from the Available Fields list,

- Click the button marked >>.

To remove fields from the Selected Fields list,

- Click the buttons marked < or <<.

To reorder the fields in the Selected Fields list,

1. Select a field to move.
2. Choose the Up or Down button to change the field's position in the list.

For the purposes of this exercise, use all the fields from the Available Fields list. Choose Next to proceed.

5. The next Form Expert screen presents options for displaying the selected fields on the form. The Expert explains and illustrates each of your choices.

   For the purposes of this exercise, choose the Vertical option.
6. The Form Expert generates text labels for each of the data entry components in the generated form when you opt for a vertical layout. You can choose the way these labels are displayed in relation to the data entry fields. The screen explains and illustrates your choices.

   For this exercise, choose the Left option, then choose Next to proceed.
7. Choose the Create button to generate the form.

The Form Expert generates a database form based on your choices.

See also
Using TDatabase

Edit menu

Edit commands keyboard shortcuts

Use commands from the Edit menu to manipulate text and components at design time.

The commands on the Edit menu are:

Undo/Undelete	Undoes your last action or last deletion
Redo	Reverses an undo
Cut	Removes a selected item and places it on the Clipboard
Copy	Places a copy of the selected item on the Clipboard, leaving the original in place
Paste	Copies the contents of the Clipboard into the Code Editor window or form
Delete	Removes the selected item
Select All	Selects all the components on the form
Align to Grid	Aligns the selected components to the closest grid point
Bring to Front	Moves the selected component to the front
Send to Back	Moves the selected component to the back
Align	Aligns components
Size	Resizes components
Scale	Resizes all the components on the form
Tab Order	Modifies the tab order of the components on the active form
Creation Order	Modifies the order in which nonvisual components are created
Lock Controls	Secures all components on the form in their current position
Object	Edits or converts an OLE object which you have inserted onto the form

Edit | Undo/Undelete

See also

Choose Edit|Undo in the Code Editor to undo your most recent keystrokes or mouse actions. Choose Edit|Undelete when working with a form to replace an item you just deleted.

Using Undo in the Code Editor

Undo can reinsert any characters you delete, delete any characters you insert, replace any characters you overwrite, or move your cursor back to its prior position.

You can undo multiple successive actions by choosing Undo repeatedly. This undoes your changes by "stepping back" through your actions and reverting them to their previous state. You can specify an undo limit on the Editor Options page of the Project| Environment dialog box.

If you undo a block operation, your file appears as it was before you executed the block operation.

The Undo command does not change an option setting that affects more than one window.

Check Group Undo on the Editor Options page of the Options|Environment dialog box to undo a group of actions.

See also

Edit|Redo

Keyboard shortcuts for the Edit menu

Edit | Redo

See also

Choose Edit|Redo to reverse the effects of your most recent undo.

Redo has an effect only immediately after an Undo command.

See also

Edit|Undo/Undelete

Edit commands keyboard shortcuts

Edit | Cut

See also

Choose Edit|Cut to remove the following items from their current position and place them on the Clipboard:

- Selected text from the Code Editor.
- Components from the form.
- Menus from the Menu designer.

Cut replaces the current Clipboard contents with the selected item.

To insert the contents of the Clipboard elsewhere,

- Choose Edit|Paste.

See also
Edit commands keyboard shortcuts

Edit | Copy

See also

Choose Edit|Copy to place an exact copy of the selected text, component, or menu on the Clipboard and leave the original untouched. Copy replaces the current Clipboard contents with the selected items.

To paste the contents on the Clipboard elsewhere,

- Choose Edit|Paste.

Edit | Paste

See also

Choose Edit|Paste to insert the contents of the Clipboard into the active Code Editor page, the active form, or active menu in the Menu designer.

Note: You can paste only text into the Code Editor window, components onto the form, and menu items into the Menu designer.

When pasting into the Code Editor window, the text is inserted at the current cursor position.

When pasting onto the form, nonvisual components are pasted into the upper left corner of the form, and visual components are pasted into the exact position from which they were cut or copied.

When pasting into the Menu designer, menu items are inserted at the cursor position.

You can paste the current contents of the Clipboard as many times as you like until you cut or copy a new item onto the Clipboard.

See also

Edit|Copy

Edit|Cut

Edit commands keyboard shortcuts

Edit | Delete

Choose Edit|Delete to remove the selected text or component without placing a copy on the Clipboard.
Even though you cannot paste the deleted text, you can restore it by immediately choosing Edit|Undo/Undelete.
Delete is useful if you want to remove an item but you do not want to overwrite the contents of the Clipboard.

Edit | Select All

Choose Edit|Select All to select every component on the active form. When you select all the components only those properties which the components have in common will appear in the Object Inspector.

Edit | Align To Grid

Choose Edit|Align To Grid to align the selected components to the closest grid point.
You can specify the grid size on the Preferences page of the Options|Environment dialog box.

Edit | Bring To Front

See also

Choose Edit|Bring To Front to move a selected component in front of all other components on the form. This is called changing the component's z-order.

Note: The Bring To Front and Send To Back commands do not work if you are combining windowed and non-windowed components. For example, you cannot change the z-order of a label in relation to a button.

Edit | Send To Back

See also

Choose Edit|Send To Back to move a selected component behind all other components on the form. This is called changing the component's z-order.

Note: The Send To Back and Bring To Front commands do not work if you are combining windowed and non-windowed components. For example, you cannot change the z-order of a label in relation to a button.

See also
Changing the Z-order of components

Edit | Align

See also

Choose Edit|Align to open the Alignment dialog box.

Alignment dialog box

Use this dialog box to line up selected components in relation to each other or to the form.

- The Horizontal alignment options align components along their right edges, left edges, or midline.
- The Vertical alignment options align components along their top edges, bottom edges, or midline.

The options for horizontal or vertical aligment are:

Option	Description
No Change	Does not change the alignment of the component
Left Sides	Lines up the left edges of the selected components (horizontal only)
Centers	Lines up the centers of the selected components
Right Sides	Lines up the right edges of the selected components (horizontal only)
Tops	Lines up the top edges of the selected components (vertical only)
Bottoms	Lines up the bottom edges of the selected components (vertical only)
Space Equal	Lines up the selected components equidistant from each other
Center In Window	Lines up the selected components with the center of the window

See also

Alignment Palette

Aligning components

Edit | Size

Choose Edit|Size to open the Size dialog box.

Size dialog box

Use this dialog box to resize multiple components to be exactly the same height or width.

- The Width options change the horizontal size of the selected components.
- The Height options align the vertical size of the selected components.

The options for horizontal or vertical sizing are:

Option	Description
No Change	Does not change the size of the components.
Shrink To Smallest	Resizes the group of components to the height or width of the smallest selected component.
Grow To Largest	Resizes the group of components to the height or width of the largest selected component.
Width	Sets a custom width for the selected components.
Height	Sets a custom height for the selected components.

Edit | Scale

Choose Edit|Scale to open the Scale dialog box.

Scale dialog box

Use this dialog box to proportionally resize the form and all the components on that form.

Dialog box options

Scaling Factor, In Percent

Enter a percentage to which you want to resize the form.

Percentages over 100 grow the form.

Percentages under 100 shrink the form.

Edit | Tab Order

See also

Choose Edit|Tab Order to open the Edit Tab Order dialog box.

Edit Tab Order dialog box

Use this dialog box to modify the tab order of the components on the form or within the selected component if that component contains other components.

The list box displays those components on the active form that can be a tab stop, and their type in their current tab order. The default tab order is determined by the order in which you placed the components on the form.

To change the tabs order of a component,

1. Select the component name.
2. Click the up button to move the component up in the tab order, or click the down arrrow to move its down in the tab order.

   You can also drag the selected component to its new position in the tab order.
3. To save your changes, click OK.

See also
Setting tab order

Edit | Creation Order

Choose Edit|Creation Order to open the Creation Order dialog box.

Creation Order dialog box

Use this dialog box to specify the order in which your application will create nonvisual components. when you load the form at design time or run time.

The list box displays only those nonvisual components on the active form, their type, and their current creation order. The default creation order is determined by the order in which you placed the nonvisual components on the form.

To change the creation order,

1. Select a component name.
2. Click the up button to move the component creation order up, or click the down arrrow to move its creation order down.

   You can also drag the selected component to its new position in the creation order.
3. To save your changes, click OK.

See also

New Items dialog box

Object Repository dialog box

Select Bitmap dialog box

Edit | Lock Controls

Choose Edit|Lock Controls to secure all components on the active form in their current position. When this command is checked, you cannot move or resize a component. However, you can use the Object Inspector to edit the Height, Left, Top, and Width properties for a selected component.

When this command is checked, components are locked. When components are locked, you can choose Lock Controls to unlock them.

Note: Lock Controls has no effect on the form, itself. When you select Lock Controls, you can still resize or move the form.

Edit | Object

Choose Edit|Object to edit or convert an OLE object which you have inserted onto the form. This menu item varies depending on the selected OLE object.

If you choose Convert from the submenu, the Convert dialog box opens.

Convert dialog box

Use this dialog box to specify a different source application for an embedded object.

Dialog box options

Current Type

Displays the type of object that you are converting or activating.

Object Type

Select the type of object to which you want to convert the file.

Convert To

Converts the selected embedded object to the type of information selected in the Object Type box.

Activate As

Opens the embedded object in the type selected in the Object Type box, but returns the object to Current Type after editing.

Display As Icon

Displays the selected embedded object as an icon in a Word document.

Change Icon

Changes the icon that represents an embedded object. This button appears only if you select the Display As Icon check box.

Result

Describes the result of the selected options.

Thank you to the Delphi documentation team for all your hard work and effort.

The team is:

Mike Satenberg	Richard Nelson	Lesley Kew
Linda Jeffries	Laura Wall	Robert Palomo
Patrick King	Matt Prather	Frances Buran
Pat Zylius	Sally Wilcox	JoAnn Findley-Blevens
Holly MacClure		

File menu

See also

Use the File menu to open, save, close, and print new or existing projects and files, and to add new forms and units to the open project.

The commands on the File menu are:

New	Opens the New Items dialog box which contains new items that can be created.
New Application	Creates a new project containing a form, a unit, and a .DPR file.
New Form	Creates and adds a blank form to the project.
New Data Module	Creates and adds a new, blank data module form to the project.
Open	Use the Open dialog box to load an existing project,defProject form,defForm unit,defUnit or text file into the Code Editor.
Reopen	Displays a cascading menu containing a list of most recently closed projects and modules.
Save	Saves the current file using its current name.
Save As	Saves the current file using a new name, including modifications made to project files
Save Project As	Saves the current project using a new name.
Save All	Saves all open files, both current project and modules.
Close	Closes the current project and its associated units and forms.
Close All	Closes all open files.
Use Unit	Adds the selected unit to the uses clause of the active module.
Add to Project	Adds an existing file to the project.
Remove From Project	Removes a file from the project.
Print	Sends the active file to the printer.
Exit	Closes the open project and exits Delphi.

See also
File commands keyboard shortcuts

File | New

See also

The New Items dialog box provides a view into the Delphi Object Repository. The Object Repository contains forms, projects, and Experts. You can use the objects directly, copy them into your projects, or inherit from exisitng objects.

To open the New Items dialog box,

1. Choose File|New to open the New Items dialog box.
2. Select one of the following options:

To add a new form from the Object Repository,

1. Choose File|New to open the New Items dialog box.
2. Choose the tab that contains the item you want add from the Forms or Dialogs page.
3. Select the item in the list view that represents the kind of form you want to add.
4. Choose whether you want to copy, inherit, or use the new form.

To start a new project from a project template,

Choose File|New to display the New Items dialog box.

1. Choose the Projects tab.
2. Select the project template you want and choose OK.
3. In the Select Directory dialog box, specify a directory for the new project's files.

   A copy of the project template opens in the specified directory.

See also

Creating Components
Customizing the Component Library
Using the Component Expert
File|Save
File|Open
Managing Projects
Tools|Repository

File | New Application

Choose File|New Application to create a new Delphi project.

In addition to the standard blank project, you can specify a custom project as the default project.

To create a default project,

1 Choose Tools|Repository to open the Object Repository dialog box.

2 In the Pages list box, click on Projects.

1. A list of projects appears in the Objects list box.

3 Select the project that you want to become the default project.

4 Select the New Project check box.

5 Click OK.

The default project specified in the steps above will be now be used when you use the New Application command.

For more information, see Starting a New Project from a Project Template

You can have only one project open at any time. If a project is open when you choose File|New Project, Delphi prompts you to save any changes made to the current project.

A new project consists of:

- a new project file (PROJECT1.DPR)
- a new form file (FORM1.DFM), and its associated form unit (UNIT1.PAS)

You can change the project and unit file names when you save them.

File | New Form

Choose File|New Form to create a blank form and a new unit and add them to the project.

In addition to the standard blank form, you can specify a custom form as the default form to be added to new projects.

To create a default form,

1 Choose Tools|Repository to open the Object Repository dialog box.

2 In the Pages list box, click on Forms or Dialogs.

1. A list of items appears in the Objects list box.

3 Select the form that you want to become the default form.

4 Select the New Form check box.

5 Click OK.

The default form specified in the steps above will be now be used when you use the New Form command.

When you create a new form, Delphi automatically adds the new form and an associated unit file to the list of files included in the open project. If no project is open, a blank form is created.

If you selected blank form from the New Items dialog box, or you did not specify a default form, the new form is titled FormXX and the new unit is UnitXX.PAS. (XX represents the form/unit number. For example the first form is Form1, the second Form2, and so on.)

You can change the name of the form by editing the Name property from the Object Inspector.

You can change the unit name by saving the file with File|Save File As, or by saving the entire project using File|Save Project As.

Changes made to any form or unit name are reflected throughout the source code anywhere that name appears within that unit.

File | New Data Module

See also

Choose File|New Data Module to add a new data module to your project.

To create a new data module,

- From the File menu, select the New Data Module command.

A data module container appears on the IDE desktop and adds the data module to the project file.

At design time a data module looks like a standard Delphi form with a white background and no alignment grid. As with forms, you can place nonvisual components on a module from the Components palette, and you can resize a data module to accommodate the components you add to it.

See also

Creating a new data module
Naming a data module and its unit file
Reusing data modules in the Object Repository
Accessing a data module from a form
Copying a data module
Inheriting a data module
Inheriting event handlers
Using a data module
Adding a data module to the Object Repository

New Items dialog box

See also

Use the New Items dialog box to select a form, or project template, or expert that you can use as a starting point for your application. The New Items dialog box provides a view into the Delphi Object Repository. The Object Repository contains forms, projects, and Experts. You can use the objects directly, copy them into your projects, or inherit items from exisitng objects.

The New Items dialog box pages

When you first install Delphi, there are six different pages in the New Items dialog box. They are described in the following paragraphs.

The New Page

The new page contains items that you can include in your project. The default items are as follows:

Application	Creates a new project containing a form, a unit, and a .DPR, or provides a way for you to select a template.
Automaton Object	Creates a .pas file which contains an Automation Object template.
Component	Creates a new component using the Component Expert.
Data Module	Creates a new Data Module.
DLL	Creates a new DLL project.
Form	Creates and adds a blank form to the current project, or enables you to select a form template.
Text	Creates a new ASCII text file.
Thread Object	Creates a new Thread Object.
Unit	Creates and adds a new unit to the current project.

The Project Page

If a project is open, the second page in the New Items dialog box is the current project page. The current project title is displayed on the tab. The current project page contains all the forms of the project. You can create an inherited form from any exising project forms.

User Defined Pages

The remaining pages, if any, are user defined pages containing Forms, Projects, Data Modules, or Experts from the Object Repository. By default, the New Items dialog box contains the following user defined pages:

- Forms
- Dialogs
- Data Modules
- Projects

The New Items SpeedMenu

The speed menu provides the following options for items in the New Items dialog box.

- View Large Icons
- View Small Icons
- View List
- View Details
- Arrange by Name
- Arrange by description
- Arrange by Date
- Arrange by Author
- Properties

Note: Selecting Properties from the SpeedMenu opens the Object Repository dialog box. You can use this dialog box to edit, add pages to, and rename items in the Object Repository. You can also access this dialog from the Tools|Repository menu.

Usage Options

There are three options for including a Repository Object in your project.

- Copy the item
- Inherit from the item

- Use the item directly

Copying Items

When you copy an item you make an exact duplicate of the item and add it to your project if it is a form or data module. Any changes to the item in the Object Repository will not be reflected in your copy. Alterations you make to your copy will not affect the original Repository item.

Note: Copying is the only option available for using project templates.

Inheriting Items

The most flexible and powerful option is to inherit an item. When you inherit an item, a new class is derived from the item and is added to your project. When you recompile your project, any changes made to the item in the Object Repository will be reflected in your derived class, unless you have changed a particular aspect. Changes made to your derived class do not affect the shared item in the Object Repository.

Note: Inheriting is available as an option for forms, dialog boxes, and data modules but not for project templates. It is the *only* option available for reusing items from within the same project.

Using Items Directly

Using an item directly is the primarily used with data modules. When you use an item directly, the item is added to your project as if you had created it as part of the project. Design-time changes made to the item appear in *all* projects that directly use the item as well as any projects that inherit from the item.

Note: Using items directly is an option for forms, dialog boxes, and data modules. These items should only be modified at run time to avoid making changes that affect other modules.

The copy option is the only option for form or project experts. Using an expert doesn't actually add shared code; instead it runs a process that generates its own code.

See also
Tools|Repository

Select Directory dialog box

Use the Select Directory dialog box to choose a working directory for your new project.

To open the Select Directory dialog box,

- Select a non-blank project template from the New Items dialog box.

Directory Name
Displays the current directory. If you can enter a directory that does not exist, Delphi will create it.

Directories
Lists the current directory.

Files: (*.*)
List all the files in the current directory. You cannot select any of these files. Delphi displays this file list so you know the contents of the current directory.

Drives
Lists all the available drives. You can select one of the available drives.

Network
If you are running under Windows 95 or Windows NT you can click the Network button to open the Connect Network Drive dialog box.

File | Open

Choose File| Open to display the Open dialog box.

Open dialog box
Use the Open dialog box to load an existing project, form, unit, or text file into the Code Editor.

Simply opening a file does not automatically add it to your current project. To add a file to a project, choose File|Add to Project.

You can open multiple forms, units, or text files but you can only have one project open at any time. If a project is open when you open a project, Delphi prompts you to save any changes made to the current project.

Look In
Lists the current directory. Use the drop down list to select a different drive or directory.

Files
Displays the files in the current directory that match the wildcards in File Name or the file type in Files Of Type. You can display a list of files (default) or you can show details for each file.

File Name
Enter the name of the file you want to load or wildcards to use as filters in the Files list box. When you select a

Files of Type
Choose the type of file you want to open; the default file type is Project file (.DPR). All files in the current directory of the selected type appear in the Files list box.

Directories
Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives
Select the current active drive. The directory structure for the current drive appears in the Directories list box.

File | Reopen

Choose File|Reopen to reopen a recently closed project or module.

When you close a project or a module, it is added to the Reopen list.

To Reopen a project or module,

1 From the File menu, point to Reopen.

2 Click the project or module that you want to reopen.

Note: Only projects or modules that have been closed with the File|Close command appear in the Reopen list. Saved Items will not appear in the list.

File | Save

Choose File|Save to store changes made to all files included in the open project using the current name for each file.

If you try to save a project that has an unsaved project file or unit file, Delphi opens the Save As dialog box, where you enter the new file name.

Note: Open files that are not included in the project file will not be saved. To save these files, select each file in the Code Editor and choose File|Save.

File | Save As

Choose File|Save As to save the active file with a different name or in a different location.

Save project as dialog box
Use the Save As dialog box to change the project file name or to save the project in a new location. If the file name already exists, Delphi asks if you want to replace the existing file.

File Name
Enter a name for the file you are saving.

Files
Displays the files in the current directory that match the file type in the Save File as Type combo box.

Save File As Type
Choose a file extension; the default is .DPR. All files in the current directory of the selected type appear in the Files list box. Note that saving a project file with a different extension does not change the format of the file.

Directories
Select the directory where you want to store the file. In the current directory, files that match the file type in the Save Files As Type combo box appear in the Files list box.

Drives
Select the current active drive. The directory structure for the current drive appears in the Directories list box.

File | Save Project As

Choose File|Save Project As to save the .DPR file to a different name or location. Besides copying and/or renaming the .DPR file, this command saves every other file in the open project to its current location and name. If you have modified forms or units that are used by other projects, and you do not want the current modifications reflected in those other projects, you should first use File|Save File As to copy/rename each unit file before choosing this command to save the project.

Save Project As dialog box

Use the Save Project As dialog box to change the project file name or to save the project in a new location. If the file name already exists, Delphi asks if you want to replace the existing file.

File Name Edit Box

Enter a name for the project file you are saving.

Files List Box

Displays the files in the current directory that match the file type in the Save File as Type combo box.

Save File As Type Combo Box

Choose a file extension; the default is .DPR. All files in the current directory of the selected type appear in the Files list box. Note that saving a project file with a different extension does not change the format of the file.

Directories List Box

Select the directory where you want to store the file. In the current directory, files that match the file type in the Save Files As Type combo box appear in the Files list box.

Drives Combo Box

Select the current active drive. The directory structure for the current drive appears in the Directories list box.

File | Save All

Choose File|Save All to save all open files, including the current project and modules.

To save all files,

1 From the File menu, choose Save All.
2 The Save All dialog box appears with a default name for the item to be saved.
3 Type in a new file name if you do not want to use the default name.
4 Click Save.
The Save As dialog appears again with a default name for the next item to be saved.
5 Repeat steps 3-4 until all modules are saved.

File | Close

Choose File|Close to close the active window.

Closing a form also closes the associated unit file. Before closing the file, Delphi prompts you to save any changes. If you have not previously saved the project, or any file, Delphi opens the Save As dialog box, where you can enter the new file name.

If you close the project file in the Code Editor, you will close the entire project. You can also close the entire project by choosing File|Close when the Project Manager is the active window.

File | Close All

Choose File|Close all to close all open files.

To close all open files,

- From the File menu, choose Close All.

The project file and all modules are closed.

File | Use Unit

Choose File|New Unit to create a new unit and add it to the project.

The new unit is titled UnitXX.PAS. XX represents the unit number. For example, the first form is Unit1, the second Unit2, and so on.

You can change the unit name by saving the file with File|Save File As, or by saving the entire project using File|Save Project As.

Changes made to any unit name are reflected throughout the source code anywhere that name appears within that unit.

Use Unit dialog box

Choose File|Use Unit to open this dialog box.

You use this dialog box to use a unit in the current unit.

To add a unit,

1 From the File menu, select Use Unit.

The Use Unit dialog box appears.

2 In the Use Unit list, click the unit name you want to add.

3 Click OK to add the unit to the current unit.

Use Unit list

This list displays a list of all units in the project that are not being used by the current unit.

File | New | Component

Choose File|New|Component to open the Component Expert dialog box.

Component Expert dialog box
Use this dialog box to create the basic unit for a new component.

Class Name
Enter the name of the new class you are creating. A general rule is that all Object Pascal classes are prefaced with a T. For example, the name of your new button component could be TMYBUTTON.

Ancestor Class
Use the drop-down list to select a base class, or enter the name of a base class for your new component. Unless you override them in the component declaration, your new component will inherit all the properties, methods, and events from its ancestor class.

Palette Page
Use the drop-down list to select a page, or enter the name of the page on which you want your new component to appear when you add it to the library.

File | Open File

See also

Choose File|Open File to display the Open File dialog box.

Open File dialog box
Use the Open File dialog box to load an existing file into Delphi. If you enter the name of a file that does not exist, Delphi opens the Open File dialog box, where you can select what kind of file you want to create.

File Name
Enter the name of the file you want to load or wildcards to use as filters in the Files list box.

Files
Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type
Choose the type of file you want to open; the default file type is Source file (.PAS). All files in the current directory of the selected type appear in the Files list box.

Directories
Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives
Select the current active drive. The directory structure for the current drive appears in the Directories list box.

See also

Adding units and forms to a project

File commands keyboard shortcuts

Open File dialog box

Use the Open File dialog box to specify the type of file you want to create when you enter a new file name into the File Name edit box. Select File|Open File to display this dialog box.

There are three types of files you can create:

- Form
- Unit
- Text file

To choose a type of file to create,

- Select the file type and click OK.

Delphi creates a file of the selected type but does not add it to the project.

Note: If you are creating a file that you want to include in the current project, you need to add it to the project. To add a file to the project, choose File|Add File or Add File from the Project Manager SpeedMenu.

File | Add to Project

See also

Choose File|Add to Project to open the Add To Project dialog box.

Add To Project dialog box
Use the Add To Project dialog box to add an existing unit and its associated form to the Delphi project. When you add a unit to a project, Delphi automatically adds that unit to the **uses** clause of the project file.

File Name
Enter the name of the file you want to load or wildcards to use as filters in the Files list box.

Files
Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type
Choose the type of file you want to open; the default file type is Source file (.PAS). All files in the current directory of the selected type appear in the Files list box.

Directories
Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives
Select the current active drive. The directory structure for the current drive appears in the Directories list box.

See also

Adding units and forms to a project

File|Remove from Project

Viewing units and forms in a project

File | Remove From Project

See also

Choose File|Remove From Project open the Remove From Project dialog box.

Remove From Project dialog box

Use this dialog box to select a module to remove from the current project. When you click OK, Delphi removes the selected module from the **uses** clause of the current project file but does not delete the file from your disk. Any references to the unit that you added must be removed manually.

If you have modified the file you are removing during this editing session, Delphi prompts you to save your changes, just in case you want to use the form or unit in another project. If you have not modified the file, Delphi removes that file from the project without prompting you.

Warning: Do not delete unit files by using other file management programs, or directly from the DOS prompt. Doing so will cause errors.

See also

File|Add to Project

Removing units and forms from a project

Viewing units and forms in a project

File | Print

Choose File|Print to print the active page in the Code Editor or the active form. When you choose File|Print, Delphi displays one of two dialog boxes depending on whether the Code Editor or the form is the active window.

- When the Code Editor is active, Delphi displays the Print Selection dialog box.
- When the form is active, Delphi displays the Print Form dialog box.

Print Form dialog box

Use this dialog box to specify any scaling options when printing a form. The scaling options act accordingly depending on the size of the printer paper. You can change the size of the paper using the Paper Size option in the Printer Setup dialog box.

To display this dialog box, select File|Print when a form is active.

There are three available scaling options:

- Proportional - Scales the form using value of the PixelsPerInch property. Depending on the value of PixelsPerInch, your form may print on more than one page.
- Print To Fit Page - Scales the form so that it will fit onto one page.
- No Scaling - Prints the form using its current onsceen size. If you choose this option, your form might print on more than one page.

Setup

Click the Setup button to display the Printer Setup dialog box.

Print Selection dialog box

Use this dialog box to print the active file from the Code Editor.

File To Print
Lists the file that you are going to print. The file listed is the active page in the Code Editor when you chose File|Print.

Print Selected Block
Sends only the selected block of text to the printer. This option is available only when you have text selected in the file.
If this option is not checked, the entire file will print.

Header/Page Number
Includes the name of the file, current date, and page number at the top of each page.

Line Numbers
Places line numbers in the left margin.

Syntax Print
Uses bold, italic, and underline characters to indicate elements with syntax highlighting.

Use Color
Prints colors that match colors onscreen (requires a color printer).

Wrap Lines
Uses multiple lines to print characters beyond the page width. If not selected, code lines are truncated and characters beyond the page width do not print.

Left Margin
Specifies the number of character spaces used as a margin between the left edge of the page and the beginning of each line.

Setup
Click the Setup to display the Printer Setup dialog box.

Printer Setup

Changes printer options and selects a printer from a list. To display this dialog box, click Setup from the Print Selection dialog box. For more information about setting printer options, see your Windows documentation.

Default Printer
Makes the default printer current.

Specific Printer
Makes a specific printer current. Choose the printer you want from the list box.

Orientation
Prints your document lengthwise or widthwise.

Portrait
Prints text across the narrowest side of the paper (e.g. 8.5" x 11").

Landscape
Prints text along the widest side of the paper (e.g. 11" x 8.5").

Paper Size
Specifies the standard paper sizes supported by your printer.

Paper Source
Specifies which paper tray or paper feeding method your printer uses.

Options
Opens the Options dialog box so you can set options specific to your printer driver. For information on Options, choose the Help button.

File | Exit

Choose File|Exit to close the open project and then close Delphi.
If you exit Delphi before saving your changes, Delphi asks you if you want to save them.

File | (History list)

Choose a file name at the bottom of the File menu to reopen it. When reopened, the file is removed from the list of closed files.

The last three files closed during your editing session in Delphi appear at the bottom of the File menu so you can quickly reopen them.

Help menu

Use the Help menu to access the online Help system, which displays in a special Help window.

The Help system provides information on virtually all aspects of the Delphi environment, Object Pascal language, libraries, and so on.

The Help menu contains two topics:

Help Topics	Displays the Help Topics dialog box
About	Displays a copyright and version number for Delphi

Help | Help Topics

Choose Help|Help Topics to display the Help Topics dialog box.

To find a topic in Help,

- Click the Contents tab to browse through topics by category.
- Click the Index tab to see a list of index entries: either type the word you're looking for or scroll through the list.
- Click the Find tab to search for words or phrases that may be contained in a Help topic.

Help | About

Choose Help|About to display the About Delphi dialog box showing copyright and version information.

Form SpeedMenu

See also

Use the Form SpeedMenu to manipulate components in the form at design time.

To display the Form SpeedMenu,

1. On the form, select the component or components you want to manipulate.

   (Select a single component by clicking it. Select more than one by dragging across them or by holding down Shift while clicking each one.)

2. Right-click anywhere on the form, or press Alt-F10.

The following commands always appear on the Form SpeedMenu:

Align To Grid
Bring To Front
Send To Back
Align
Size
Scale
Tab Order
Creation Order
Add To Repository
View as Text
View as Form

Other commands appear on the Form SpeedMenu when you select certain components on the form. The additional commands are:

Using the Fields editor
Query Builder
Execute
Edit Report
Next Page
Previous Page

Your installation of Delphi may have additional commands installed by a third party. To get help on a command not listed above, display the Form SpeedMenu and select the command. This activates the module named in the command. Then, press F1 to get help on that module.

See also

Form

Query Builder (Form SpeedMenu)

See also

Choose Query Builder from the Form SpeedMenu when the Query component is selected, to open the Visual Query Builder. If a database is not already open, this command opens the Databases dialog box which enables you to select a database.

See also
TQuery component

Execute (Form SpeedMenu)

See also

Choose Execute from the Form SpeedMenu when you have the BatchMove component selected, to perform at design time, the process specified in the Mode property.

The Mode property enables you to perform any of the following tasks:

- Copy a dataset to a table.
- Append a dataset to a table.
- Update a table with data from a dataset.
- Displaying data using data from a control.

To run this process at run time, you must call the Execute method for BatchMove.

See also

Using TBatchMove

TBatchMove component

Edit Report (Form SpeedMenu)

See also

Choose Edit Report from the Form SpeedMenu when you have the Report component selected, to load ReportSmith and open the report you specified using the ReportName and ReportDir properties.

If you have not previously specified a report, Delphi opens the ReportSmith Open Report dialog box which enables you to select a report.

You can also perform this command by double-clicking the Report component on the form.

See also
TReport component

Next Page (Form SpeedMenu)

See also

Choose Next Page from the Form SpeedMenu when you have the either the Notebook component, or TabbedNotebook component selected, to set the ActivePage property to the next page in the notebook.

The next page is determined by the page order which you can specify using the Notebook editor.

See also

Notebook component

TabbedNotebook component

Previous Page (Form SpeedMenu)

See also

Choose Previous Page from the Form SpeedMenu when you have the either the Notebook component, or TabbedNotebook component selected, to set the ActivePage property to the previous page of the notebook.

The previous page is determined by the page order which you can specify using the Notebook editor.

See also

Notebook component

TabbedNotebook component

Align To Grid (Form SpeedMenu)

Choose Align To Grid from the Form SpeedMenu to align the selected components to the closest grid point. You can specify the size of the grid on the Preferences page of the Options|Environment dialog box. This SpeedMenu command works the same as Edit|Align To Grid.

Bring To Front (Form SpeedMenu)

See also

Choose Bring To Front from the Form SpeedMenu to move a selected component in front of all other components on the form. This is called changing the component's z-order.

This SpeedMenu command works the same as Edit|Bring To Front.

Send To Back (Form SpeedMenu)

See also

Choose Send To Back from the Form SpeedMenu to move a selected component behind all other components on the form. This is called changing the component's z-order.

This SpeedMenu command works the same as Edit|Send To Back.

Align (Form SpeedMenu)

See also

Choose Align from the Form SpeedMenu to open the Alignment dialog box.

Alignment dialog box

Use this dialog box to line up selected components in relation to each other or to the form.

- The Horizontal alignment options align components along their right edges, left edges, or center.
- The Vertical alignment options align components along their top edges, bottom edges, or center.

The options for Horizontal or Vertical alignment are

Option	Description
No Change	Does not change the alignment of the component
Left Sides	Lines up the left edges of the selected components (horizontal only)
Centers	Lines up the centers of the selected components
Right Sides	Lines up the right edges of the selected components (horizontal only)
Tops	Lines up the top edges of the selected components (vertical only)
Bottoms	Lines up the bottom edges of the selected components (vertical only)
Space Equally	Lines up the selected components equidistant from each other
Center in Window	Lines up the selected components with the center of the window.

This SpeedMenu command works the same as Edit|Align.

Size (Form SpeedMenu)

Choose Size from the Form SpeedMenu to open the Size dialog box.

Size dialog box

Use this dialog box to resize multiple components to be exactly the same height or width.

- The Horizontal options align the width of the selected components.
- The Vertical options align the height of the selected components.

The options for Horizontal or Vertical sizing are

Option	Description
No Change	Does not change the size of the components.
Shrink To Smallest	Resizes the group of components to the height or width of the smallest selected component.
Grow To Largest	Resizes the group of components to the height or width of the largest selected component.
Width	Sets a custom width for the selected components. To use this option, you must set Horizontal to Enter Value.
Height	Sets a custom height for the selected components. To use this option, you must set Vertical to Enter Value.

This SpeedMenu command works the same as Edit|Size.

Scale (Form SpeedMenu)

Choose Scale from the Form SpeedMenu to open the Scale dialog box.

Scale dialog box

Use this dialog box to proportionally resize the form and all of its components.

Scaling Factor In Percent

Enter a percentage to which you want to resize the form.

Percentages over 100 grow the form.

Percentages under 100 shrink the form.

This SpeedMenu command works the same as Edit|Scale.

Tab Order (Form SpeedMenu)

See also

Choose Tab Order from the Form SpeedMenu to open the Edit Tab Order dialog box.

Edit Tab Order
Use this dialog box to modify the current tab order of the components on the active form or within the selected component if that component can contain other components.

Dialog box options

Controls
Lists the components on the active form in their current tab order. The first component listed is the first component in the tab order.

Up
Click Up to move the component selected in the Controls list box higher in the tab order.

Down
Click Down to move the component selected in the Controls list box lower in the tab order.

See also
Setting the tab order

Creation Order (Form SpeedMenu)

Choose Creation Order from the Form SpeedMenu to open the Creation Order dialog box.

Creation Order dialog box

Use this dialog box to specify the order in which your application will create nonvisual components.

The list box displays only those nonvisual components on the active form, their type, and their current creation order. The default creation order is determined by the order in which you placed the nonvisual components on the form.

To change the creation order,

1. Select the component name.
2. Click the up button to move the component creation order up, or click the down arrrow to move its creation order down.
3. To save your changes, click OK.

This SpeedMenu command works the same as Edit|Creation Order.

Add To Repository (Form SpeedMenu)

Choose Add To Repository from the Form SpeedMenu to open the Add To Repository dialog box. Use this command to easily add any form to the Object Repository.

Once you've designed a custom dialog box, you might want to reuse it in other projects. The best way to do this is to add the form to the Object Repository.

Saving a form as an object is similar to saving a copy of the form under a different name. When you save a form as a object, however, it then appears in the Object Repository.

View as Text (Form SpeedMenu)

Use this command to view a text description of the form's attributes.

Note: This commad changes to View as From when you view the form as text.

View as Form (Form SpeedMenu)

Use this command to view the form as a visual object.

Note: This commad changes to View as Text when you view the form as a visual object.

Select Bitmap dialog box

Use the Select Bitmap dialog box to choose a bitmap to represent your template in the New Items dialog box dialog box.
You can use a bitmap of any size, but it will be cropped to 60 x 40 pixels.

To open this dialog box,

- Click the Browse button in the Add To Repository dialog box.

Dialog box options

File Name
Enter the name of the file you want to use, or enter wildcards to use as filters in the Files list box.

Files
Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type
Choose the type of file you want to open; the default file type is a bitmap file (.BMP). All files in the current directory of the selected type appear in the Files list box.

Directories
Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives
Select the current active drive. The directory structure for the current drive appears in the Directories list box.

Project Manager SpeedMenu

See also

The Project Manager SpeedMenu contains commands that enable you to manage your project.

The commands on the Project Manager SpeedMenu are:

Save Project
Add To Repository
New Unit
New Form
Add File
Remove File
View Unit
View Form
View Project Source
Options
Update

To open the Project Manager SpeedMenu, do one of the following:

- Right-click anywhere on the SpeedBar.
- Press Alt+F10 when this is the active window.

See also

Project Manager

Save Project (Project Manager SpeedMenu)

Choose Save Project from the Project Manager SpeedMenu to store changes made to all files in the open project using each file's current name.

If you try to save a project that has an unsaved project file or unit file, Delphi opens the Save As dialog box, where you enter the new file name.

This SpeedMenu command works the same as File|Save Project.

Add To Repository (Project Manager SpeedMenu)

Choose Add To Repository from the Project Manager SpeedMenu to open the Save Project Template dialog box. Use the Save Project Template dialog box to add a project template to the Object Repository.

Save Project Template dialog box

Use this dialog box to save a project template to the Object Repository.

After saving an application as a template, use the Edit Object Info dialog box to edit the description, delete the template, or change the bitmap.

Dialog box options

Title
Enter the name of the template.

The maximum length for a title is 40 characters.

Description
Enter a description of the template. The description appears under the template name on the Select Template dialog box. The maximum length for a description is 255 characters.

Page
From the drop down list box, choose the name of the page (probably Projects) you want the template to appear on.

Author
Enter text indentifying the author of the application.

Author informaton only appears when you select the View Details option from the SpeedMenu.

Template bitmap
Click the Browse button to open the Select Bitmap dialog box.

You can use a bitmap of any size, but it will be cropped to 60 x 40 pixels.

New Unit (Project Manager SpeedMenu)

Choose New Unit from the Project Manager SpeedMenu to create a new unit and add it to the project. When you add a new unit the Code Editor becomes the active window and the new unit is the active page.

The new unit is titled UnitXX.PAS. XX represents the unit number. For example, the first form is Unit1, the second Unit2, and so on.

You can change the unit name by saving the file with File|Save As, or by saving the entire project using Save Project from the Project Manager SpeedMenu. Changing a unit name by any other means might cause an error.

This SpeedMenu command works the same as the File|New Unit.

New Form (Project Manager SpeedMenu)

Choose New Form from the Project Manager SpeedMenu to create a blank form and a new unit and add them to the project.

The new form is titled FormXX and the new unit is UnitXX.PAS. (XX represents the form/unit number. For example the first form is Form1, the second Form2, and so on.)

You can change the name of the form by editing the Name property from the Object Inspector.

You can change the unit name by saving the file with File|Save As or by saving the entire project using Save Project on the Project Manager SpeedMenu.

This SpeedMenu command works the same as File|New Form.

Add File (Project Manager SpeedMenu)

See also

Choose Add File from the Project Manager SpeedMenu to open the Add To Project dialog box.

Add To Project
Use the Add To Project dialog box to add an existing unit and its associated form to the Delphi project. When you add a unit to a project, Delphi automatically adds that unit to the **uses** clause of the project file.

Dialog box options

File Name
Enter the name of the file you want to load, or enter wildcards to use as filters in the Files list box.

Files
Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type
Choose the type of file you want to open; the default file type is Source file (.PAS). All files in the current directory of the selected type appear in the Files list box.

Directories
Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives
Select the current active drive. The directory structure for the current drive appears in the Directories list box.

This SpeedMenu command works the same as File|Add File.

See also

Adding units and forms to a project

Remove file

Viewing units and forms in a project

Remove File (Project Manager SpeedMenu)

See also

Choose Remove File from the Project Manager SpeedMenu to remove the module selected in the Project Manager from the **uses** clause of the current project file.

If you have modified the file you are removing during this editing session, Delphi prompts you to save your changes, just in case you want to use the form or unit in another project. If you have not modified the file, Delphi removes that file from the project without prompting you.

If a file has been modified during the current editing session, it is bold in the Project Manager.

Warning: Do not delete unit files by using other file management programs, or directly from the DOS prompt. Doing so causes errors.

This SpeedMenu command works the same as File|Remove File.

See also

Add File

Removing units and forms from a project

Viewing units and forms in a project

View Unit (Project Manager SpeedMenu)

See also

Choose View Unit from the Project Manager SpeedMenu to make the selected module in the Project Manager the active page of the Code Editor. It also makes the Code Editor the active window.

If the module you want to view is not currently open, Delphi opens it.

This SpeedMenu command works the same as View|Unit.

See also
Code Editor

View Form (Project Manager SpeedMenu)

See also

Choose View Form from the Project Manager SpeedMenu to make the form associated with the module selected in the Project Manager the active window.

If the form you want to view is not currently open, Delphi opens it.

This SpeedMenu command works the same as View|View Form.

View Project Source (Project Manager SpeedMenu)

See also

Choose View Project from the Project Manager SpeedMenu to make the project file the active page in the Code Editor. If the project file is not currently open, Delphi opens it.

Options (Project Manager SpeedMenu)

See also

Choose Options from the Project Manager SpeedMenu to open the Options|Project dialog box.
You can use this dialog box to set compiler directives.
This SpeedMenu command works the same as the Options|Project.

See also

Compiler directives

Update (Project Manager SpeedMenu)

Choose Update from the Project Manager SpeedMenu to synchronize any changes you have made directly in the project (.DPR) file itself.

Note: This command remains disabled (dimmed) unless you have *manually* edited the project file. Since Delphi maintains the project file for you automatically, manually modifying the .DPR file is not recommended.

SpeedBar SpeedMenu

See also

The SpeedBar SpeedMenu contains commands that enable you to edit or rearrange the buttons on the SpeedBar. You can also use the SpeedBar SpeedMenu to hide the SpeedBar.

The commands on the SpeedBar SpeedMenu are:

Configure

Show Hints

Hide

Help

To display the SpeedBar SpeedMenu,

- Right-click anywhere on the SpeedBar.

See also
SpeedBar

Configure (SpeedBar SpeedMenu)

See also

Choose Configure from the SpeedBar SpeedMenu to open the SpeedBar Editor dialog box.

SpeedBar editor
Use this dialog box to add buttons that represent menu commands to the SpeedBar, remove buttons from the SpeedBar, or rearrange buttons on the SpeedBar.

Dialog box options

Categories
Select a menu whose commands you want to add as buttons to the SpeedBar. The commands on the selected menu appear in the Command list box.

Command
Drag and drop a command from this list box onto the SpeedBar. The Commands list box displays all the commands available on the menu selected in the Categories list box.

The icon to the left of the menu command shows how the button will appear on the SpeedBar.

Reset Defaults
Click Reset Defaults to reset the SpeedBar to the default configuration.

When the SpeedBar editor is open, you can delete or rearrange any of the buttons currently on the SpeedBar; however, none of the buttons on the SpeedBar are active.

See also

Configuring the SpeedBar

SpeedBar

Configuring the SpeedBar

See also

The Delphi SpeedBar is configurable. That is, you can do any of the following:

- Add buttons to the SpeedBar
- Remove buttons from the SpeedBar
- Rearrange the buttons on the SpeedBar

Before you can configure the SpeedBar, you must open the SpeedBar editor by choosing Configure from the SpeedBar SpeedMenu.

To add buttons to the SpeedBar,

1. Enlarge the SpeedBar area by dragging the separator line between the SpeedBar and Component palette to the right. When you are over the separator line, the cursor turns into a horizontally split cursor.
2. Select a menu from the Categories list box. The menu commands on the selected menu are displayed in the Commands list box.
3. Drag the menu command you want to add from the Commands list box and drop it on the SpeedBar in an open space.

Note: You can have overlapping buttons on the SpeedBar, but this is not advisable.

To remove a button from the SpeedBar,

- Drag the button off the SpeedBar. An open space appears where the button was.

To rearrange the buttons on the SpeedBar,

- Drag and drop the button to a new position.

See also
SpeedBar

Hide

Choose Hide from the following SpeedMenus to close that interface element.

Alignment Palette SpeedMenu
Component Palette SpeedMenu
Object Inspector SpeedMenu
SpeedBar SpeedMenu

If close an interface element, you can display it again using the View menu.

Show Hints

Choose Show Hints from the following SpeedMenus to toggle the display of Help Hints.

Alignment Palette SpeedMenu
Component Palette SpeedMenu
ObjectBrowser SpeedMenu
SpeedBar SpeedMenu

When this command is checked, Help Hints are enabled.

Help

Choose Help from any of the following SpeedMenus to get Help on using that element:

Alignment Palette SpeedMenu
Component Palette SpeedMenu
Object Inspector SpeedMenu
SpeedBar SpeedMenu

Component Palette SpeedMenu

See also

The Component Palette SpeedMenu enables you to edit or rearrange the components on the Component palette. You can also use the Component Palette SpeedMenu to hide the Component palette.

Configure

Show Hints

Hide

Help

To display the Component Palette SpeedMenu,

- Right-click anywhere on the SpeedBar.

See also

Component palette

Configure (Component Palette SpeedMenu)

See also

Choose Configure from the Component palette SpeedMenu to open the Palette page of the Options|Environment dialog box. You can use this dialog box to rearrange the Components on the Component palette.

See also

Component palette

Customizing the Component library

Customizing the Component palette

Alignment Palette SpeedMenu

See also

The Alignment Palette SpeedMenu contains the following commands:

Stay On Top

Show Hints

Hide

Help

See also
Alignment palette

Stay On Top (Alignment Palette SpeedMenu)

Choose Stay On Top from the Alignment palette SpeedMenu to keep the Alignment palette in front of all other Delphi windows and dialog boxes.

Object Inspector SpeedMenu

See also

The Object Inspector SpeedMenu provides you with commands for closing the Object Inspector, displaying Help, and for keeping the Object Inspector the topmost window.

The commands on the Object Inspector SpeedMenu are:

- Expand
- Collapse
- Stay On Top
- Hide
- Help

See also
Object Inspector

Stay On Top (Object Inspector SpeedMenu)

Choose Stay On Top from the Object Inspector SpeedMenu to keep the Object Inspector in front of all other Delphi windows and dialog boxes.

Expand (Object Inspector SpeedMenu)

Choose Expand from the Object Inspector SpeedMenu to view the nested properties of the selected property.

Properties with nested properties show a plus (+) sign on their left side in the Object Inspector. You need to view these nested properties to set them.

For more information on viewing nested properties, see Viewing nested properties.

Collapse (Object Inspector SpeedMenu)

Choose Collapse from the Object Inspector SpeedMenu to hide the nested properties of the selected property.

Properties with nested properties show a plus (+) sign on their left side in the Object Inspector. You need to view these nested properties to set them.

For more information on viewing and hiding nested properties, see Viewing nested properties.

Breakpoint List SpeedMenu

See also

Use the Breakpoint List SpeedMenu to access commands that enable you to manipulate breakpoints.

The commands on the Breakpoint List SpeedMenu are:

Edit Breakpoint	Opens the Edit Breakpoint dialog box, where you can create new breakpoints
Add Breakpoint	Opens the Edit Breakpoint dialog box, where you can create new breakpoints
Delete Breakpoint	Removes a breakpoint
Enable Breakpoint	Enables a disabled breakpoint
Disable Breakpoint	Disables an enabled breakpoint
View Source	Locates a breakpoint in your source code quickly
Edit Source	Locates a breakpoint in your source code quickly and activates the Code Editor
Disable All Breakpoints	Disables all enabled breakpoints
Enable All Breakpoints	Enables all disabled breakpoints
Delete All Breakpoints	Removes all breakpoints

To display the Breakpoint List SpeedMenu, do one of the following:

- Right-click anywhere in the Breakpoint List window.
- Press Alt+F10 when the Breakpoint List window is active.

See also
Breakpoints

Edit Breakpoint (Breakpoint List SpeedMenu)

See also Breakpoint List SpeedMenu

Choose Edit Breakpoint from the Breakpoint List SpeedMenu to open the Edit Breakpoint dialog box, where you can create new breakpoints.

Alternate ways to perform this command are:

- Choose Run|Add Breakpoint.
- Choose Add Breakpoint from the Breakpoint List SpeedMenu.

See also

Breakpoints

The Breakpoint List window

The Delphi Debugger

Add Breakpoint (Breakpoint List SpeedMenu)

See also Breakpoint List SpeedMenu

Choose Add Breakpoint from the Breakpoint List SpeedMenu to open the Edit Breakpoint dialog box, where you can create new breakpoints.

Alternate ways to perform this command are:

- Choose Run|Add Breakpoint.
- Right-click an existing breakpoint in the Breakpoint List window and choose Edit Breakpoint from the Breakpoint List SpeedMenu.

See also

Breakpoints

The Breakpoint List window

The Delphi Debugger

Delete Breakpoint (Breakpoint List SpeedMenu)

See also Breakpoint List SpeedMenu

Choose Delete Breakpoint from the Breakpoint List SpeedMenu to remove a breakpoint.

When you no longer need to examine the code at a breakpoint location, you can delete the breakpoint from the debugging session. This command is not reversible.

See also

Delete All Breakpoints (Breakpoint List SpeedMenu)
Breakpoints
The Breakpoint List window
The Delphi Debugger

Enable Breakpoint (Breakpoint List SpeedMenu)

See also Breakpoint List SpeedMenu

Choose Enable Breakpoint from the Breakpoint List SpeedMenu to enable a disabled breakpoint.

Disabling a breakpoint hides the breakpoint from the current program run. When you disable a breakpoint, its settings remain defined, but the breakpoint does not cause your program to stop. When you set a breakpoint, it is enabled by default.

Disabling is useful when you temporarily do not need a breakpoint but want to preserve its settings.

See also

Enable All Breakpoints (Breakpoint List SpeedMenu)
Disable Breakpoint (Breakpoint List SpeedMenu)
Disable All Breakpoints (Breakpoint List SpeedMenu)
Breakpoints
The Breakpoint List window
The Delphi Debugger

Disable Breakpoint (Breakpoint List SpeedMenu)

See also Breakpoint List SpeedMenu

Choose Disable Breakpoint from the Breakpoint List SpeedMenu to disable an enabled breakpoint.

Disabling a breakpoint hides the breakpoint from the current program run. When you disable a breakpoint, its settings remain defined, but the breakpoint does not cause your program to stop. When you set a breakpoint, it is enabled by default.

Disabling is useful when you temporarily do not need a breakpoint but want to preserve its settings.

See also

Disable All Breakpoints (Breakpoint List SpeedMenu)
Enable Breakpoint (Breakpoint List SpeedMenu)
Enable All Breakpoints (Breakpoint List SpeedMenu)
Breakpoints
The Breakpoint List window
The Delphi Debugger

View Source (Breakpoint List SpeedMenu)

See also Breakpoint List SpeedMenu

Choose View Source from the Breakpoint List SpeedMenu to locate a breakpoint in your source code quickly.

The View Source command scrolls the Code Editor to the location of the breakpoint that is selected in the Breakpoint List window.

See also

Edit Source (Breakpoint List SpeedMenu)
Breakpoints
The Breakpoint List window
The Delphi Debugger

Edit Source (Breakpoint List SpeedMenu)

See also Breakpoint List SpeedMenu

Choose Edit Source from the Breakpoint List SpeedMenu to locate a breakpoint in your source code quickly.

The Edit Source command scrolls the Code Editor to the location of the breakpoint that is selected in the Breakpoint List window, and makes the Code Editor active.

See also

View Source (Breakpoint List SpeedMenu)
Breakpoints
The Breakpoint List window
The Delphi Debugger

Disable All Breakpoints (Breakpoint List SpeedMenu)

See also Breakpoint List SpeedMenu

Choose Disable All Breakpoints from the Breakpoint List SpeedMenu to disable all enabled breakpoints.

Disabling a breakpoint hides the breakpoint from the current program run. When you disable a breakpoint, its settings remain defined, but the breakpoint does not cause your program to stop. When you set a breakpoint, it is enabled by default.

Disabling is useful when you temporarily do not need a breakpoint but want to preserve its settings.

See also

Disable Breakpoint (Breakpoint List SpeedMenu)
Enable Breakpoint (Breakpoint List SpeedMenu)
Enable All Breakpoints (Breakpoint List SpeedMenu)
Breakpoints
The Breakpoint List window
The Delphi Debugger

Enable All Breakpoints (Breakpoint List SpeedMenu)

See also Breakpoint List SpeedMenu

Choose Enable All Breakpoints to enable all disabled breakpoints.

Disabling a breakpoint hides the breakpoint from the current program run. When you disable a breakpoint, its settings remain defined, but the breakpoint does not cause your program to stop. When you set a breakpoint, it is enabled by default.

Disabling is useful when you temporarily do not need a breakpoint but want to preserve its settings.

See also

Enable Breakpoint (Breakpoint List SpeedMenu)
Disable Breakpoint (Breakpoint List SpeedMenu)
Disable All Breakpoints (Breakpoint List SpeedMenu)
Breakpoints
The Breakpoint List window
The Delphi Debugger

Delete All Breakpoints (Breakpoint List SpeedMenu)

See also Breakpoint List SpeedMenu

Choose Delete All Breakpoints from the Breakpoint List SpeedMenu to remove all breakpoints.

When you no longer need to examine the code at a breakpoint location, you can delete the breakpoint from the debugging session. This command is not reversible.

See also

Delete Breakpoint (Breakpoint List SpeedMenu)
Breakpoints
The Breakpoint List window
The Delphi Debugger

Call Stack SpeedMenu

See also

Use the Call Stack SpeedMenu to access commands that enable you to examine previous function calls.

The commands on the Call Stack List SpeedMenu are:

View Source	Locates a function call in your source code quickly
Edit Source	Locates a function call in your source code quickly, and activates the Code Editor

To display the Call Stack SpeedMenu, do one of the following:

- Right-click anywhere in the Call Stack window.
- Press Alt+F10 when the Call Stack window is active.

See also
Call Stack

View Source (Call Stack SpeedMenu)

See also Call Stack SpeedMenu

Choose View Source from the Call Stack SpeedMenu to locate a function call in your source code quickly.

The View Source command scrolls the Code Editor to the location of the function call that is selected in the Call Stack window.

See also

Edit Source (Call Stack SpeedMenu)

Locating function calls

The Delphi Debugger

Edit Source (Call Stack SpeedMenu)

See also Call Stack SpeedMenu

Choose Edit Source from the Call Stack SpeedMenu to locate a function call in your source code quickly.

The Edit Source command scrolls the Code Editor to the location of the function call that is selected in the Call Stack window, and makes the Code Editor active.

See also

View Source (Call Stack SpeedMenu)

Locating function calls

The Delphi Debugger

Watch List SpeedMenu

See also

Use the Watch List SpeedMenu to access commands that enable you manipulate watch points.

The commands on the Watch List SpeedMenu are:

Edit Watch	Opens the Watch Properties dialog box, where you can create and modify watches
Add Watch	Opens the Watch Properties dialog box, where you can create and modify watches
Enable Watch	Enables a disabled watch expression
Disable Watch	Disables an enabled watch expression
Delete Watch	Removes a watch expression
Disable All Watches	Enables all disabled watch expressions
Enable All Watches	Disables all enabled watch expressions
Delete All Watches	Removes all watch expressions

To display the Watch List SpeedMenu, do one of the following:

- Right-click anywhere in the Watch List window.
- Press Alt+F10 when the Watch List window is active.

See also

WatchList

Edit Watch (Watch List SpeedMenu)

See also Watch List SpeedMenu

Choose Edit Watch from the Watch List SpeedMenu to open the Watch Properties dialog box, where you can create and modify watches. After you create a watch, use the Watch List window to display and manage the current list of watches.

Alternate ways to perform this command are:

- Choose Run|Add Watch.
- Choose Add Watch At Cursor from the Code Editor SpeedMenu.
- Choose Add Watch from the Watch List SpeedMenu.

See also

Watches

The Watch List window

The Delphi Debugger

Add Watch (Watch List SpeedMenu)

See also Watch List SpeedMenu

Choose Add Watch from the Watch List SpeedMenu to open the Watch Properties dialog box, where you can create and modify watches. After you create a watch, use the Watch List window to display and manage the current list of watches.

Alternate ways to perform this command are:

- Choose Run|Add Watch.
- Choose Add Watch At Cursor from the Code Editor SpeedMenu.
- Right-click an existing watch in the Watch List window and choose Edit Watch from the Watch List SpeedMenu.

See also

Watches

The Watch List window

The Delphi Debugger

Enable Watch (Watch List SpeedMenu)

See also Watch List SpeedMenu

Choose Enable Watch from the Watch List SpeedMenu to enable a disabled watch expression.

Disabling a watch hides the watch from the current program run. When you disable a watch, its settings remain defined, but Delphi does not evaluate the watch.

Disabling watches improves performance of the debugger because it does not monitor the watch as you step through or run your program. When you set a watch, it is enabled by default.

See also

Enable All Watches (Watch List SpeedMenu)
Disable Watch (Watch List SpeedMenu)
Disable All Watches (Watch List SpeedMenu)
Watches
The Watch List window
The Delphi Debugger

Disable Watch (Watch List SpeedMenu)

See also Watch List SpeedMenu

Choose Disable Watch from the Watch List SpeedMenu to disable an enabled watch expression.

Disabling a watch hides the watch from the current program run. When you disable a watch, its settings remain defined, but Delphi does not evaluate the watch.

Disabling watches improves performance of the debugger because it does not monitor the watch as you step through or run your program. When you set a watch, it is enabled by default.

See also

Disable All Watches (Watch List SpeedMenu)
Enable Watch (Watch List SpeedMenu)
Enable All Watches (Watch List SpeedMenu)
Watches
The Watch List window
The Delphi Debugger

Delete Watch (Watch List SpeedMenu)

See also Watch List SpeedMenu

Choose Delete Watch from the Watch List SpeedMenu to remove a watch expression.

When you no longer need to examine the value of an expression, you can delete the watch from the debugging session. This command is not reversible.

See also

Delete All Watches (Watch List SpeedMenu)
Watches
The Watch List window
The Delphi Debugger

Enable All Watches (Watch List SpeedMenu)

See also Watch List SpeedMenu

Choose Enable All Watches from the Watch List SpeedMenu to enable all disabled watch expressions.

Disabling a watch hides the watch from the current program run. When you disable a watch, its settings remain defined, but Delphi does not evaluate the watch.

Disabling watches improves performance of the debugger because it does not monitor the watch as you step through or run your program. When you set a watch, it is enabled by default.

See also

Enable Watch (Watch List SpeedMenu)
Disable Watch (Watch List SpeedMenu)
Disable All Watches (Watch List SpeedMenu)
Watches
The Watch List window
The Delphi Debugger

Disable All Watches (Watch List SpeedMenu)

See also Watch List SpeedMenu

Choose Disable All Watches from the Watch List SpeedMenu to disable all enabled watch expressions.

Disabling a watch hides the watch from the current program run. When you disable a watch, its settings remain defined, but Delphi does not evaluate the watch.

Disabling watches improves performance of the debugger because it does not monitor the watch as you step through or run your program. When you set a watch, it is enabled by default.

See also

Disable Watch (Watch List SpeedMenu)
Enable Watch (Watch List SpeedMenu)
Enable All Watches (Watch List SpeedMenu)
Watches
The Watch List window
The Delphi Debugger

Delete All Watches (Watch List SpeedMenu)

See also Watch List SpeedMenu

Choose Delete All Watches from the Watch List SpeedMenu to remove all watch expressions.

When you no longer need to examine the value of an expression, you can delete the watch from the debugging session. This command is not reversible.

See also

Delete Watch (Watch List SpeedMenu)

Watches

The Watch List window

The Delphi Debugger

Project menu

See also

Use the Compile menu to compile, or build your application. You need to have a project open.
The commands on the Compile menu are:

Add to Project	Enables you to add a file to a project
Remove from Project	Enables you to remove a file from a project
Add To Repository	Enables you to easily add a project to the Object Repository
Compile	Compiles any source code that has changed since the last compile
Build All	Compiles everything in the project, regardless of whether source has changed
Syntax Check	Compiles your project but does not link it.
Information	Displays all the build information and build status for your project
Options	Opens the Project Options dialog box.

See also

Compiling, building and running projects

Project | Add to Project

See also

Choose Project|Add to Project to open the Add To Project dialog box.

Add To Project

Use the Add To Project dialog box to add an existing unit and its associated form to the Delphi project. When you add a unit to a project, Delphi automatically adds that unit to the **uses** clause of the project file.

Dialog box options

File Name

Enter the name of the file you want to load or wildcards to use as filters in the Files list box.

Files

Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type

Choose the type of file you want to open; the default file type is Source file (.PAS). All files in the current directory of the selected type appear in the Files list box.

Directories

Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives

Select the current active drive. The directory structure for the current drive appears in the Directories list box.

See also

Adding units and forms to a project

Viewing units and forms in a project

Project | Remove From Project

See also

Choose File|Remove From Project open the Remove From Project dialog box.

Remove From Project dialog box

Use this dialog box to select a module to remove from the current project. When you click OK, Delphi removes the selected module from the **uses** clause of the current project file but does not delete the file from your disk. Any references to the unit that you added must be removed manually.

If you have modified the file you are removing during this editing session, Delphi prompts you to save your changes, just in case you want to use the form or unit in another project. If you have not modified the file, Delphi removes that file from the project without prompting you.

Warning: Do not delete unit files by using other file management programs, or directly from the DOS prompt. Doing so will cause errors.

See also

Removing units and forms from a project
Viewing units and forms in a project

Add To Repository

See also

Choose Project|Add To Repository to open the Save As Template dialog box. Use this command to add a project to the Object Repository.

You can add your own projects and forms to those already available in the Object Repository. This is helpful in situations where you want to enforce a standard framework for programming projects throughout an organization.

See also
Using Project Templates

Project | Compile

See also

Use Project|Compile to compile all files in the current project that have changed since the last build into a new executable file (.EXE).

This option is identical to Project|Build All, except that it builds only those files that have changed. Build All rebuilds all files regardless of whether or not they have changed.

If you have selected Show Compiler Progress from the Preferences page of the Options|Environment dialog box, the Compiling dialog box displays information about the compilation progress and results.

When your application successfully compiles, choose OK to close the Compiling dialog box.

If the compiler encounters an error, Delphi reports that error on the status line of the Code Editor and places the cursor on the line of source code containing the error.

The compiler builds .EXE files according to the following rules:

- The project (.DPR) file is always recompiled.
- If the source code of a unit has changed since the last time the unit was compiled, the unit is compiled. When a unit is compiled, Delphi creates a file with a .DCU extension for that unit.
  If Delphi can not locate the source code for a unit, that unit is not recompiled.
- If the interface part of a unit has changed, all the other units that depend on the changed unit are recompiled.
- If a unit links in an .OBJ file (external routines), and the .OBJ file has changed, the unit is recompiled.
- If a unit contains an Include file. and the Include file has changed, the unit is recompiled.

You can choose to compile only portions of your code if you use conditional directives and predefined symbols in your code.

Project | Build All

See also

Choose Project|Build All to rebuild all the components of your application regardless of whether they have changed.

This option is identical to Project|Compile, except that it rebuilds everything. Compile rebuilds only those files that have been changed.

The Project|Build All command recompiles all files included in the project file.

Project | Syntax Check

Choose Project|Syntax Check to compile the modules of your project but not link them. This provides you with a means for checking your code for compile time errors.

If you do not have a project open when you choose this command, only the current module will compile.

Project | Information

Choose Project|Information to open the Information dialog box.

Information dialog box

Use this dialog box to view the program compilation information and compilation status for your project.

Program Information

The Program Information options provide you with information about your project.

Options	What it lists
Source Compiled	Total number of lines compiled
Code Size	Total size of the executable or DLL without debug information
Data Size	Memory needed to store the global variables
Initial Stack Size	Memory needed to store the local variables

Status Information

The Status Information line displays whether or not your last compile succeeded or failed.

Project | Options

Choose Project|Options to display the Project Options dialog box. Use the pages of this dialog box to specify form, application, compiler, and linker options for your project, and to manage project directories.

The pages of the Project Options dialog box are:

Forms
Application
Compiler
Linker
Directories/Conditionals

To change pages in the dialog box,

- Click the tab at the bottom of the dialog box that represents the page you want to use.

Default

Check Default to save the current project options so that every new project you create will use those options.

To save the current project options as defaults,

- Check Default and click OK.

Forms (Project | Options)

Use the Forms page of the Project Options dialog box to select the main form for your applications, and to choose which of the available forms are automatically created and in which order.

Dialog box options

Main Form

Displays the form users will see when they start your application. Use the drop-down list to select which form is the main form for the project.

The main form is the first form listed in the Auto-Create Forms list box.

Auto-Create Forms

Lists forms that are automatically added to the startup code of the project file and are created at run time. You can rearrange the create order of forms by dragging and dropping forms to a new location.

To select multiple forms, hold down the Shift key while selecting the form names.

Available Forms

Lists those forms that are used by your application but are not automatically created. If you want to create an instance of that form, you must call the the form's Create method.

Arrow buttons

Use the arrow buttons to move one or more files or all the files from one list box to the other.

To move all the files from one list box into the other,

- Click the double arrow buttons (>> or <<).
- Drag and drop the files from one list box into the other.

To move only the selected file or files from one list box into the other,

- Click the single arrow buttons (> or <).
- Drag and drop the file from one list box into the other.

Default

Check Default to save the current project options so that every new project you create will use those options.

Application (Project | Options)

See also

Use the Application page of the Project Options dialog box to specify a title, a Help file, and an icon for your application.

Dialog box options

Title

Specify a title that will appear under the application's icon when the application is minimized. The character limit is 255 characters.

Help File

Specify the name of the Help file (.HLP) your application will use whenever your application calls Help. The Help file name is passed to the WinHelp function call.

If you are unsure of the Help file name, you can click the Browse button to display the Application Help File dialog box.

Icon

Displays the icon file (.ICO) that will represent the application in the Program Manager and when the application is minimized.

Default

Check Default to save the current project options so that every new project you create will use those options.

To change the icon,

- Click Load Icon and Delphi displays the Application Icon dialog box, where you can select an icon.

See also

Creating Windows Help

Specifying the main form for the project

Specifying the project Help and icon files

Application Icon dialog box

Application Icon dialog box

Use the Application Icon dialog box to select an icon that will represent your application in the Program Manager and when your application is minimized.

To display this dialog box,

- Click Load Icon on the Application page of the Project Options dialog box.

Dialog box options

File Name

Enter the name of the file you want to use, or enter or wildcards to use as filters in the Files list box.

Files

Displays all files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type

Choose the type of file you want to use; the default file type is an icon (.ICO). All files in the current directory of the selected type appear in the Files list box.

Directories

Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives

Select the current active drive. The directory structure for the current drive appears in the Directories list box.

Application Help File dialog box

Use the Application Help File dialog box to select a Help file to use as the Help file for your application. The Help file you specify here is entered into the Help File edit box on the Applicaton page of the Project Options dialog box.

To display this dialog box,

- Click Browse on the Application page of the Project Options dialog box.

Dialog box options

File Name

Enter the name of the file you want to use, or enter or wildcards to use as filters in the Files list box.

Files

Displays all files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type

Choose the type of file you want to use; the default file type is a Help file (.HLP). All files in the current directory of the selected type appear in the Files list box.

Directories

Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives

Select the current active drive. The directory structure for the current drive appears in the Directories list box.

Compiler (Project | Options)

See also

Use the Compiler page of the Project Options dialog box to set options for how you want your program to compile. These options correspond to switch directives that you can also set directly in your program code.

Selecting an option is equivalent to setting the switch directive to its positive (+) state.

Code generation	Effect
Optimizations	Enables compiler optimizations. Corresponds to {$O}.
Aligned record fields	Aligns elements in structures to 32-bit boundaries. Corresponds to {$A}.
Pentium-Safe FDIV	Generates code that detects a faulty floating-point division instruction. Corresponds to {$U}.
Stack frames	Forces compiler to generate stack frames on all procedures and functions. Corresponds to {$W}.

Runtime errors	Effect
Range Checking	Checks that array and string subscripts are within bounds. Corresponds to {$R}.
Stack Checking	Checks that space is available for local variables on the stack. Corresponds to {$S}.
I/O Checking	Checks for I/O errors after every I/O call. Corresponds to {$I}.
Overflow Checking	Checks overflow for integer operations. Corresponds to {$Q}.

Syntax options	Effect
Strict Var-Strings	Sets up string parameter error checking. Corresponds to {$V}. (If the Open parameters option is selected, this option is not applicable.)
Complete Boolean Eval	Evaluates every piece of an expression in Boolean terms, regardless of whether the result of an operand evaluates as false. Corresponds to {$B}.
Extended Syntax	Enables you to define a function call as a procedure and to ignore the function result. Also enables Pchar support. Corresponds to {$X}.
Typed @ Operator	Controls the type of pointer returned by the @ operator. Corresponds to {$T}.
Open Parameters	Enables open string parameters in procedure and function declarations. Corresponds to {$P}. Open parameters are generally safer, and more efficient.
Huge Strings	Enables new garbage collected strings. The **string** keyword corresponds to the new AnsiString type with this option enabled. Otherwise the **string** keyword corresponds to the ShortString type. Corresponds to {$H}.
Assignable Typed Constants	Enable this for backward compatability with Delphi 1.0. When enabled, the compiler allows assignments to typed constants. Corresponds to {$J}.

Messages	Effect
Show Hints	Causes the compiler to generate hint messages.
Show Warnings	Causes the compiler to generate warning messages.

Debugging	Effect
Debug Information	Puts debug information into the unit (.DCU) file. Corresponds to {$D}.
Local Symbols	Generates local symbol information. Corresponds to {$L}..
Symbol Information	Generates symbol information. Corresponds to {$Y}.

Default check box

Check Default to save the current project options so that every new project you create will use those options.

See also

Compiler directives

Linker (Project | Options)

See also

Use the Linker page of the Project Options dialog box to specify how your program files are linked.

Map file options

Select the type of map file produced, if any. The map file is placed in the Output Directory specified on the Directories/Conditionals page, and it has a .MAP extension.

Option	Effect
Off	Does not produce map file.
Segments	Linker produces a map file that includes a list of segments, the program start address, and any warning or error messages produced during the link.
Publics	Linker produces a map file that includes a list of segments, the program start address, any warning or error messages produced during the link, and a list of alphabetically sorted public symbols.
Detailed	Linker produces a map file that includes a list of segments, the program start address, any warning or error messages produced during the link, a list of alphabetically sorted public symbols, and an additional detailed segment map. The detailed segment map includes the segment address, length in bytes, segment name, group, and module information.

Linker output options

Specify the output from the linker.

Option	Effect
Generate DCUs	Output standard Delphi DCU format files
Genrate Object Files	Output binary Object files (OBJ)

EXE and DLL Options

Check box	What it does
Generate Console Application	Causes linker to set a flag in the application's .EXE file indicating a console mode application.
Include TDW Debug Info	Places debug information in your program's executable file. This will make the resulting .EXE file larger, but it does not affect memory requirements or performance. If you are debugging your program using Turbo Debugger for Windows, this option must be selected.

Memory sizes

Use these edit boxes to specify the minimum and maximum stack size and heap image base for the compiled executable. Memory-size settings can also be specified in your source code with the $M compiler directive.

Option	Specifies
Min Stack Size	Initial commited size of the stack..
Max Stack Size	Total reserved size of the stack.
Image Base	Specifies the preferred load address of the compiled image. This value is typically only changed when compiling to a DLL.

Default

Check Default to save the current project options so that every new project you create will use those options.

See also

Compiler directives

Directories/Conditionals (Project | Options)

See also

Use the Directories/ Conditionals page of the Project Options dialog box to specify the location of files needed to compile, link, and distribute your program. Click the down arrow next to any edit box to choose from a list of previously entered directories or symbols.

Dialog box options

Output Directory

Where the compiler should put the compiled units and the executable file.

Search Path

Where your source files are located. Only those files on the compiler's search path or the library search path will be included in the build. If you try to build your project with a file not on the search path, you will receive a compiler error.

You must include the entire search path.

Conditional Defines

Symbols referenced in conditional compiler directives. You can separate multiple defines with semicolons.

Unit Aliases

Useful for backwards compatibility. Specify alias names for units that may have changed names or were merged into a single unit. The format is <oldunit>=<newunit>. You can separate multiple aliases with semicolons. The default value is **WinTypes=Windows;WinProcs=Windows.**Default

Check Default to save the current project options so that every new project you create will use those options.

Guidelines for search paths

Use the following guidelines when entering directory names into the Search Path edit box:

- Separate multiple directory path names with a semicolon (;).
- Whitespace before and after the semicolon is allowed but not required.
- You can use a maximum of 127 (note that this may change...) characters (including whitespace).
- Relative and absolute path names are allowed, including path names relative to the logged position in drives other than the current one.

See also

Compiling, building, and running projects
Managing projects

Run menu

The Run menu contains commands that provide a way for you to debug your program from within Delphi. The following commands form the core functionality of the integrated debugger:

Run	Compiles and executes your application
Parameters	Specifies startup parameters for your application
Step Over	Executes a program one line at a time, stepping over procedures while executing them as a single unit
Trace Into	Executes a program one line at a time, tracing into procedures and following the execution of each line
Trace To Next Source Line	Executes the program, stopping at the next executable source line in your code
Run To Cursor	Runs the loaded program up to the location of the cursor in the Code Editor window
Show Execution Point	Positions the cursor at the execution point in an edit window
Program Pause	Temporarily pauses the execution of a running program
Program Reset	Ends the current program run and releases it from memory
Add Watch	Opens the Watch Properties dialog box, where you can create and modify watches
Add Breakpoint	Opens the Edit Breakpoint dialog box, where you can create and modify breakpoints
Evaluate/Modify	opens the Evaluate/Modify dialog box, where you can evaluate or change the value of an existing expression

Note: The integrated debugger commands become accessible when you generate symbolic debug information for the project you are working on.

Run | Run

See also

Choose Run|Run to compile and execute your application, using any startup parameters you specified in the Parameters dialog box.

If you have modified the source code since the last compilation, the compiler recompiles those changed modules and relinks your application.

If the compiler encounters an error, it displays an Error dialog box. When you choose OK to dismiss the dialog box, the Code Editor places the cursor on the line of code containing the error.

The compiler builds .EXE files according to the following rules:

- The project (.DPR) file is always recompiled.
- If the source code of a unit has changed since the last time the unit was compiled, the unit is compiled. When a unit is compiled, Delphi creates a file with a .DCU extension for that unit.
  If Delphi cannot locate the source code for a unit, that unit is not recompiled.
- If the interface section of a unit has changed, all the other units that depend on the changed unit are recompiled.
- If a unit links in an .OBJ file (external routines), and the .OBJ file has changed, the unit is recompiled.
- If a unit contains an Include file. and the Include file has changed, the unit is recompiled.

See also

The Delphi Debugger
Controlling program execution
Build commands keyboard shortcuts
Compiling, building and running projects

Run | Parameters

See also

Choose Run|Parameters to open the Parameters dialog box.

Parameters dialog box

Use this dialog box to pass command-line parameters to your application when you run it, just as if you were running the application from the Program Manager File|Run menu.

You can use these parameters with the ParamCountand ParamStr() functions.

Dialog box options

Run Parameters

Enter the parameters you want to pass to your application when it starts, or use the drop-down button to choose from a history of previously specified parameters. Parameters take effect only when your application is started. Do not enter the application name in this edit box.

See also
The Delphi Debugger

Run | Step Over

See also

Choose Run|Step Over to execute a program one line at a time, stepping over procedures while executing them as a single unit.

The Step Over command executes the program statement highlighted by the execution point and advances the execution point to the next statement.

- If you issue the Step Over command when the execution point is located on a function call, the debugger runs that function at full speed, then positions the execution point on the statement that follows the function call.
- If you issue Step Over when the execution point is positioned on the **end** statement of a routine, the routine returns from its call, and the execution point is placed on the statement following the routine call.

The debugger considers multiple program statements on one line of text as a single line of code; you cannot individually debug multiple statements contained on a single line of text. The debugger also considers a single statement that spans several lines of text as a single line of code.

By default, when you initiate a debugging session with Run|Step Over, Delphi moves the execution point to the first line of code that contains debugging information (this is normally a location that contains user-written code). To step over start-up code that Delphi automatically generates, see Debugging Start-up Code.

In addition to stepping over procedures, you can trace into them, following the execution of each line. Use Run|Trace Into to execute each line of a procedure.

An alternative way to perform this command is:

- Choose the Step Over button.

See also

The Delphi Debugger
Controlling program execution
Debug commands keyboard shortcuts

Run | Trace Into

See also

Choose Run|Trace Into to execute a program one line at a time, tracing into procedures and following the execution of each line.

The Trace Into command executes the program statement highlighted by the execution point and advances the execution point to the next statement.

- If you issue the Trace Into command when the execution point is located on a function call, the debugger traces into the function, positioning the execution point on the function's first statement.
- If you issue Step Over when the execution point is positioned on the **end** statement of a routine, the routine returns from its call, and the execution point is placed on the statement following the routine call.
- If the execution point is located on a function call that does not have debugging information, such as a library function, the debugger runs that function at full speed, then positions the execution point on the statement following the function call.

By default, when you initiate a debugging session with Run|Trace Into, Delphi moves the execution point to the first line of code that contains debugging information (this is normally a location that contains user-written code). To trace into start-up code that Delphi automatically generates, see Debugging Start-up Code.

In addition to tracing into procedures, you can step over them, executing each procedure as a single unit. Use Run|Step Over to execute procedures as a single unit.

An alternative way to perform this command is:

- Choose the Trace Into button.

See also

The Delphi Debugger
Controlling program execution
Debug commands keyboard shortcuts

Run | Trace To Next Source Line

See also

Use this command to stop on the next source line in your application, regardless of the control flow. For example, if you select this command when stopped at a Windows API call that takes a callback function, control will return to the next source line, which in this case is the callback function.

See also

The Delphi Debugger
Controlling program execution
Debug commands keyboard shortcuts

Run | Run To Cursor

See also

Choose Run|Run To Cursor to run the loaded program up to the location of the cursor in the Code Editor window.

When you run to the cursor, your program is executed at full speed, then pauses and places the execution point on the line of code containing the cursor.

You can use Run To Cursor to run your program and pause before the location of a suspected problem. You can then use Run|Step Over or Run|Trace Into to control the execution of individual lines of code.

An alternative way to perform this command is:

- Choose Run To Cursor from the Code Editor SpeedMenu.

See also

The Delphi Debugger
Controlling program execution
Debug commands keyboard shortcuts

Run | Show Execution Point

See also

Choose Run|Show Execution Point to position the cursor at the execution point in an edit window. If you closed the edit window containing the execution point, Delphi opens an edit window displaying the source code at the execution point.

See also

The Delphi Debugger
Controlling program execution
Debug commands keyboard shortcuts

Run | Program Pause

See also

Choose Run|Program Pause to temporarily pause the execution of a running program.

The debugger pauses program execution and positions the execution point on the next line of code to execute. You can examine the state of your program in this location, then continue debugging by running, stepping, or tracing.

If your program assumes control and does not return to the debugger--for example, if it is running in an infinite loop--you can press Ctrl+Alt+Sys Req to stop your program. You might need to press this key combination several times before your program actually stops, because the command will not work if Windows kernel code is executing.

In addition to temporarily pausing a program running in the debugger, you can also stop a program and release it from memory. Use Run|Program Reset to stop a running program and release it from memory.

See also

The Delphi Debugger
Controlling program execution
Debug commands keyboard shortcuts

Run | Program Reset

See also

Choose Run|Program Reset to end the current program run and release it from memory.

Use Program Reset to restart a program from the beginning, such as when you step past the location of a bug, or if variables or data structures become corrupted with unwanted values.

When you reset a program, Delphi performs the following actions:

- Closes all open program files
- Releases resources allocated by calls to the VCL
- Clears all variable settings

Resetting a program does not delete any breakpoints or watches you have set, which makes it easy to resume a debugging session.

Windows resources

Resetting a program does not necessarily release all Windows resources allocated by your program. In most cases, all resources allocated by VCL routines are released. However, Windows resources allocated by code which you have written might not be properly released.

If your system becomes unstable, through either multiple hardware or language exceptions or through a loss of system resources as a resul of resetting your program, you should exit Delphi before restarting your debugging session.

See also

The Delphi Debugger
Controlling program execution
Debug commands keyboard shortcuts

Run | Add Watch

See also

The Add Watch command opens the Watch Properties dialog box, where you can create and modify watches. After you create a watch, use the Watch List window to display and manage the current list of watches.

Alternate ways to perform this command are:

- Choose Add Watch at Cursor from the Code Editor SpeedMenu.
- Choose Add Watch from the Watch List SpeedMenu.
- Right-click an existing watch in the Watch List window and choose Edit Watch from the Watch List SpeedMenu.

See also

The Delphi Debugger
Watches
Debug commands keyboard shortcuts

Run | Add Breakpoint

See also

The Add Breakpoint command opens the Edit Breakpoint dialog box, where you can create and modify breakpoints.

Alternate ways to perform this command are:

- Choose Add Breakpoint from the Breakpoint List SpeedMenu.
- Right-click an existing breakpoint in the Breakpoint List window and choose Edit Breakpoint from the Breakpoint List SpeedMenu.

See also

The Delphi Debugger
Breakpoints
Controlling program execution
Debug commands keyboard shortcuts

Run | Evaluate/Modify

See also

The Evaluate/Modify command opens the Evaluate/Modify dialog box, where you can evaluate or change the value of an existing expression.

An alternate way to perform this command is:

- Choose Evaluate/Modify from the Code Editor SpeedMenu.

See also

The Delphi Debugger
Debug commands keyboard shortcuts

Search menu

Search commands keyboard shortcuts

Use the Search menu to locate text, errors, objects, units, variables and symbols in the Code Editor.

The commands on the Search menu are:

Find	Searches for specific text
Replace	Searches for specific text and replace it with new text
Search Again	Repeats search
Incremental Search	Searches for text as you type
Go to Line Number	Moves cursor to specific line number
Show Last Compile Error	Moves cursor to the line of code that caused last compiler error
Find Error	Searches for most recent run-time error
Browse Symbol	Searches for specified symbol

Search | Find

See also

Choose Search|Find to display the Find Text dialog box.

Find Text dialog box

Use this dialog box to specify text you want to locate, and to set options that affect the search.

Dialog box options

Text to Find

Enter a search string; or, click the down arrow next to the input box to select from a list of previously entered search strings.

Options

Specifies attributes for the search string.

Case Sensitive	Differentiates uppercase from lowercase when performing a search.
Whole Words Only	Searches for words only. (With this option off, the search string might be found within longer words.)
Regular Expression	Recognizes regular expressions in the search string.

Direction

Specifies which direction you want to search, starting from the current cursor position.

Forward	From the current position to the end of the file. Forward is the default Direction setting.
Backward	From the current position, to the beginning of the file.

Scope

Determines how much of the file is searched.

Global	Searches the entire file, in the direction specified by the Direction setting. Global is the default scope.
Selected Text	Searches only the selected text, in the direction specified by the Direction setting. You can use the mouse or block commands to select a block of text.

Origin

Specifies where the search should start.

From Cursor	The search starts at the cursor's current position, and then proceeds either forward to the end of the scope, or backward to the beginning of the scope depending on the Direction setting. From Cursor is the default Origin setting.
Entire Scope	The search covers either the entire block of selected text or the entire file (no matter where the cursor is in the file), depending upon the Scope options.

See also

Block commands keyboard shortcuts (brief)
Block commands keyboard shortcuts (default and classic)
Block commands keyboard shortcuts (Epsilon)
Search commands keyboard shortcuts

Search | Replace

See also

Choose Search|Replace to display the Replace Text dialog box.

Replace Text

Use this dialog box to specify text you want to search for and then replace with other text (or with nothing).

Most components of the Replace Text dialog box are identical to those in the Find Text dialog box.

Dialog box options

Text to Find

Enter a search string; or, click the down arrow next to the input box to select from a list of previously entered search strings.

Replace With

Enter the replacement string; or, click the down arrow next to the input box to select from a list of previously entered search strings. To replace the text with nothing, leave this input box blank.

Options

Specifies attributes for the search strings.

Case Sensitive	Differentiates uppercase from lowercase when performing a search.
Whole Words Only	Searches for words only. (With this option off, the search string might be found within longer words.)
Regular Expression	Recognizes specific regular expressions in the search string.
Prompt on Replace	Prompts you before replacing each occurrence of the search string. When Prompt on Replace is off, the editor automatically replaces the search string.

Direction

Specifies which direction you want to search the file, starting from the current cursor position.

Forward	From the current position, to the end of the file. Forward is the default Direction setting.
Backward	From the current position, to the beginning of the file.

Scope

Determines how much of the file is searched.

Global	The entire file, in the direction specified by the Direction setting. Global is the default scope.
Selected Text	Only the selected text, in the direction specified by the Direction setting. To select a block of text, use the mouse or block commands.

Origin

Specifies where the search should start.

From Cursor	The search starts at the cursor's current position, and then proceeds either forward to the end of the scope, or backward to the beginning of the scope depending on the Direction setting. From Cursor is the default Origin setting.
Entire Scope	The search covers either the entire block of selected text or the entire file (no matter where the cursor is in the file), depending upon the Scope options.

Replace All

Click Replace All to replace every occurrence of the search string. If you have checked the Prompt on Replace option, the Confirm dialog box appears on each occurrence of the search string.

Search | Search Again

See also

Choose Search|Search Again to repeat the last Find, or Replace command.

The settings last made in the Find Text or Replace Text dialog box remain in effect when you choose Search Again. For instance, if you have not cleared the Replace Text settings, the Search Again command searches for the string you last specified and replaces it with the text specified in the Replace Text dialog box.

Search | Incremental Search

Choose Search|Incremental Search to bypass the Find Text dialog box by moving the cursor directly to the next occurrence of text that you type.

When you are performing an incremental search, the Code Editor status line reads "Searching For:" and displays each letter you have typed.

For example, if you type "class" the cursor moves to the next occurrence of the word, highlighting each letter as you type it. This behavior continues until the editor loses focus or you press Enter or Escape.

Here are some Incremental Search keystroke options.

Option	Effect	
Backspace	Remove the last character from the search string and move to the previous match.	
F3	Repeat search	(Default keybinding)
Ctrl+L	Repeat search	(Classic keybinding)
Ctrl+S	Repeat search	(Epsilon keybinding)
Shift+F5	Repeat search	(Brief keybinding)

Search | Go to Line Number

See also

Choose Search|Go to Line Number to display the Go To Line Number dialog box.

Go to Line Number dialog box

This dialog box prompts you for the line number you want to find. (The current line number and column number are displayed in the Line and Column Indicator on the status bar of the Code Editor.)

When this dialog box first appears, the current line number is in the input box.

Dialog box options

Enter New Line Number

Specify the line number of the code you want to go to; or click the down arrow next to the input box to select from a list of previously entered line numbers.

Search | Show Last Compile Error

See also

Choose Search|Show Last Compile Error to reopen the Error dialog box containing the most recent compile-time error. When you click OK to clear the dialog box, the cursor moves to the line of code that caused the error.

If the error is not in the active file, Delphi makes the file with the last compiler error active, opening a closed file if necessary.

See also

Search commands keyboard shortcuts

Search | Find Error

See also

Choose Search|Find Error to display the Find Error dialog box.

Find Error dialog box

Use this dialog box to specify the address of the most recent run-time error.

Dialog box options

Error Address

Enter the address of the most recent run-time error.

When you click OK, Delphi recompiles your program and stops at the address location you entered, highlighting the line that caused the run-time error.

Search | Browse Symbol

See also

Choose Search|Browse Symbol to display the Browse Symbol dialog box.

Browse Symbol dialog box

Use this dialog box to browse a specific symbol.

Symbol to Find input box

Enter the symbol name you want to inspect, or click the down arrow to choose from a list of previously entered symbols. You can also use the arrow keys to move through the list box.

When you click OK, Delphi opens the ObjectBrowser so you can view the symbol declaration.

See also
ObjectBrowser window

Regular expressions

Regular expressions are characters that customize a search string.

The regular expressions that Delphi recognizes are:

Character	Description
^	A circumflex at the start of the string matches the start of a line.
$	A dollar sign at the end of the expression matches the end of a line.
.	A period matches any character.
*	An asterisk after a string matches any number of occurrences of that string followed by any characters, including zero characters. For example, bo* matches bot, bo and boo but not b.
+	A plus sign after a string matches any number of occurrences of that string followed by any characters except zero characters. For example, bo+ matches boo, and booo, but not bo or be.
[ ]	Characters in brackets match any one character that appears in the brackets, but no others. For example [bot] matches b, o, or t.
[^]	A circumflex at the start of the string in brackets means NOT. Hence, [^bot] matches any characters except b, o, or t.
[-]	A hyphen within the brackets signifies a range of characters. For example, [b-o] matches any character from b through o.
{ }	Braces group characters or expressions. Groups can be nested, with a maximum number of 10 groups in a single pattern.
\	A backslash before a wildcard character tells the Code Editor to treat that character literally, not as a wildcard. For example, **\^** matches **^** and does not look for the start of a line.

Note: Delphi also supports Brief regular expressions if you are using Brief keystoke mappings.

Brief regular expressions

The symbols you can use to produce Brief regular expressions include:

<	A less than at the start of the string matches the start of a line.
%	A percent sign at the start of the string matches the start of a line.
$	A dollar sign at the end of the expression matches the end of a line.
>	A greater than at the end of the expression matches the end of a line.
?	A question mark matches any single character.
@	An at sign after a string matches any number of occurrences of that string followed by any characters, including zero characters. For example, bo@ matches bot, boo, and bo.
+	A plus sign after a string matches any number of occurrences of that string followed by any characters, except zero characters. For example, bo+ matches bot and boo, but not b or bo.
|	A vertical bar matches either expression on either side of the vertical bar. For example, bar|car will match either bar or car.
~	A tilde matches any single character that is **not** a member of a set.
[ ]	Characters in brackets match any one character that appears in the brackets, but no others. For example [bot] matches b, o, or t.
[^]	A circumflex at the start of the string in brackets means NOT. Hence, [^bot] matches any characters except b, o, or t.
[-]	A hyphen within the brackets signifies a range of characters. For example, [b-o] matches any character from b through o.
{ }	Braces group characters or expressions. Groups can be nested, with the maximum number of 10 groups in a single pattern.
****	A backslash before a wildcard character tells the Borland C++ IDE to treat that character literally, not as a wildcard. For example, **\^** matches ^ and does not look for the start of a line.

Tools menu

See also

Use the Tools menu to view and change environment settings, to modify the list of programs on the Tools menu, to modify templates and experts.

These are the commands on the Tools menu:

Options	Specifies Editor, Browser, and configuration preferences, and customizes the appearance of the Component palette.
Repository	Displays the Object Repository dialog box.
Tools	Brings up the Tools Options dialog box. Use this dialog box to add programs to, delete programs from, or edit programs on the Tools menu..
Database Desktop	Runs the Database Desktop applicaton, a database maintenance and data definition tool.

See also

Adding programs to the Tools menu

Tools|Tools

Tools | Options

Choose Tools|Options to display the Environment Options dialog box. Use the pages of this dialog box to specify Editor, Browser, and configuration preferences, and to customize the way components and pages are arranged on the Component palette.

The pages of the Environment Options dialog box are:

Preferences

Library

Editor Options

Display

Colors

Palette

Browser

To change pages in the dialog box,

- Click the tab at the bottom of the dialog box that represents the page you want to use.

Preferences (Tools | Options)

See also Environment options

Use the Preferences page of the Enviroment Options dialog box to specify your Delphi configuration preferences.

Desktop contents options

Select which desktop settings are saved when you exit Delphi.

Option	When selected
Desktop Only	Saves directory information, open files in the editor, and open windows
Desktop And Symbols	Saves desktop information and browser symbol information from the last successful compile

Autosave options

Specify which files and options are saved automatically by the environment or when you run your program. A check mark means it is enabled.

Check box	When checked
Editor Files	Saves all modified files in the Code Editor when you choose Run|Run, Run|Trace Into, Run|Step Over, Run|Run To Cursor, or when you exit Delphi
Desktop	Saves the arrangement of your desktop when you close a project or exit Delphi. When you later open the same project, all files opened when the project was last closed are opened again regardless of whether they are not used by the project.

Form designer options

Set grid preferences that make it easier to design forms.

Option	Effect
Display Grid	Makes the dots that show the form grid visible.
Snap To Grid	Automatically aligns components on the form with the nearest gridline. You cannot place a component "in between" gridlines.
Grid Size X	Sets grid spacing in pixels along the x-axis. Specify a higher number (between 2 and 128) to increase grid spacing.
Grid Size Y	Sets grid spacing in pixels along the y-axis. Specify a higher number (between 2 and 128) to increase grid spacing.
Show Component Captions	Select this option to display component captions.

Debugging check boxes

Use the Debugging check boxes to select the debugger you want to use and to enable stepping.

Check box	When checked
Integrated Debugging	Uses the Delphi Integrated Debugger
Step Program Block	Causes the debugger to stop at the first unit initialization that contains debug information
Break On Exception	Stops the application when it reaches an exception and displays the following information: ▪ The exception class ▪ The exception message ▪ The location of the execption When this option is unchecked, exceptions do not stop the running application. If you are stepping your application, you can step through the exception handlers as if going through the code sequentially.
Minimize On Run	Minimizes Delphi when you run your application by choosing Run|Run. When you close your application Delphi is restored.
Hide Designers On Run	Hides designer windows, such as the Object Inspector and Form window, while the application is running. The windows reappear when the application closes.

Show Compiler Progress

Check to see progress reports while your program compiles.

See also

The Delphi Debugger
Exception handlingexExceptionHandling
Form
Project options
Saving projects

Library (Tools | Options)

Use the Library page of the Environment Options dialog box to control various aspects of the component library, such as debug information and search path. The component library is used by the Component palette.

The options on this page take effect whenever you choose Component|Rebuild Library.

Map File options

Select the type of map file produced, if any, when you rebuild the library. The map file is placed in the same directory as the library, and it has a .MAP extension.

Option	Effect
Off	Does not produce map file.
Segments	Linker produces a map file that includes a list of segments, the program start address, and any warning or error messages produced during the link.
Publics	Linker produces a map file that includes a list of segments, the program start address, any warning or error messages produced during the link, and a list of alphabetically sorted public symbols.
Detailed	Linker produces a map file that includes a list of segments, the program start address, any warning or error messages produced during the link, a list of alphabetically sorted public symbols, and an additional detailed segment map. The detailed segment map includes the address, length in bytes, segment name, group, and module information.

Show Hints

Select this option to have the compiler generate hint messages.

Show Warnings

Select this option to have the compiler generate warning messages.

Compile With Debug Info

Check to compile the library file (.DCL) with debug information. This will make the resulting .DCL file larger, but it does not affect memory requirements or performance.

When you compile a library using debug information, you can use the integrated debugger or Turbo Debugger for Windows to debug the library file.

Save Library Source Code

Check to save the code for the library project file using the .DPR extension.

The **uses** clause of this library source file lists all the units (.DCU) that are used to build the Component palette.

Library Path

Specify search paths where compiler can find the source files for the component library. The compiler can find only those files listed in Library Path. If you try to build your project with a file not on the library path or project options path, you will receive a compiler error.

Aliases

Useful for backwards compatibility. Specify alias names for units that may have changed names or were merged into a single unit. The default value is WinTypes=Windows;WinProcs=Windows.

See also

Component palette

Customizing the component library

Palette (Options|Environment)

See also Environment options

Use the Palette page of the Enviroment Options dialog box to customize the way the component palette appears. You can rename, add or remove, or reorder pages and components.

Pages

Lists the pages in the Component palette. Pages are listed in the order they currently appear. You can rearrange these pages or view their content so that you can rearrange their components.

The Library page contains a listing of all the components installed in the library. You can move any component onto this page, however, that will not create a second instance of the component within the library.

Components

Lists the components in their current order for each page of the Component palette. The components displayed correspond to the currently selected page in the Pages list box. You can rearrange the order in which components appear on a page, or move them to a different page.

Up arrow and down arrow

Click the up arrow or the down arrow to change the position of the selected page or component. You can also drag pages or components to a new position.

Add

Click Add to display the Add Page dialog box, where you can create new pages on the Component palette.

Once you have created a new Component palette page, you can then move components from other pages onto it, or you can add new components.

Delete

Click Delete to remove the selected page or component from the palette. Before you can delete a page, it must be empty of components.

If you accidently delete a component, you restore that component by using the Library page.

Rename

Click Rename to display the Rename Page dialog box, where you can rename the pages on the Component palette.

Reset Defaults

Click Reset Defaults to reset the Component palette to the layout that is specified in the component library.

See also

Component palette
Customizing the component library
Customizing the Component palette

Rename Page dialog box

See also Options|Environment - Palette

Use this dialog box to specify a new name for a page on the Component Palette.

(You first need to select the page you want to rename from Pages list box on the Palette page of the Options|Environment dialog box.)

To open this dialog box,

- Click the Rename button on the Palette page of the Options|Environment dialog box.

Page Name

Enter the new name for the page in the Page Name edit box.

When you click OK, the new name is reflected in the Pages list box, but the new name is not reflected in the Component palette until you click OK in the Environment Options dialog box.

To exit this dialog box without changing the page name, choose Cancel.

See also
Customizing the Component palette

Add Page dialog box

See also Options|Environment - Palette

Use this dialog box to add a new page to the Component palette.

The new page is added to the end of the Pages list. You can change the position of the page using the Palette page of the Options|Environment Options dialog box.

To open this dialog box,

- Click the Add button on the Palette page of the Options|Environment dialog box.

Page Name

Enter the new name for the page in the Page Name edit box.

When you click OK, the new name is added in the Pages list box, but the new page is not added to the Component palette until you click OK in the Environment Options dialog box.

To exit this dialog box without changing the page name, choose Cancel.

Browser (Tools | Options)

See also Environment options

Use the Browser page of the Enviroment Options dialog box to specify how the ObjectBrowser functions and what symbol information is displayed.

Symbols Type Filters check boxes

Specify what symbol information is displayed in the ObjectBrowser. By default, every check box is selected; therefore all symbol information is displayed.

Choose from these symbols:

Constants	Inherited
Types	Virtual only
Functions and Procedures	Private
Variables	Protected
Properties	Public
	Published

Initial View options

Specify what you want to see when the ObjectBrowser first opens. There are three choices:

- Objects
- Units
- Globals

At any time during a browser session, you can switch views using the ObjectBrowser SpeedMenu.

Display check boxes

Use these check boxes to select how much detail you want for a symbol and whether or not you want the display sorted.

Option	What it does
Qualified Symbols	Displays the qualified identifier for a symbol. When this option is off, only the symbol name is displayed.
Sort Always	Displays the symbols in alphabetical order by symbol name. When this option is off, the symbols sort by declaration order.

Note: If both Qualified Symbols and Sort Always are checked, symbols sort by their symbol name, not their object name.

Collapse Nodes

Specify which branches of the object tree hierarchy you want collapsed or expanded when you start the ObjectBrowser.

For this setting to take effect, you must be browsing objects.

See also
ObjectBrowser window

Editor (Tools | Options)

See also Environment options

Use the Editor page of the Enviroment Options dialog box to customize the behavior of the Delphi editor.

Editor SpeedSetting

Use the Editor SpeedSettings to configure the editor.

Option	Automatically sets
Default Keymapping	Auto Indent Mode, Insert Mode, Smart Tab, Backspace Unindents, Group Undo, Overwrite Blocks, Use Syntax Highlight
IDE Classic	Auto Indent Mode, Insert Mode, Smart Tab, Backspace Unindents, Cursor Through Tabs, Group Undo, Persistent Blocks, Use Syntax Highlight
BRIEF Emulation	Auto Indent Mode, Insert Mode, Smart Tab, Backspace Unindents, Cursor Through Tabs, Cursor Beyond EOF, Keep Trailing Blanks, BRIEF Regular Expressions, Force Cut And Copy Enabled, Use Syntax Highlight
Epsilon Emulation	Auto Indent Mode, Insert Mode, Smart Tab, Backspace Unindents, Cursor Through Tabs, Group Undo, Overwrite Blocks, Use Syntax Highlight

Editor Options check boxes

Use the following editor options to control text handling in the Code Editor. Check the option to enable it.

Check box	When selected
Auto Indent Mode	Positions the cursor under the first nonblank character of the preceding nonblank line when you press Enter.
Insert Mode	Inserts text at the cursor without overwriting existing text. If Insert Mode is disabled, text at the cursor is overwritten. (Use the Ins key to toggle Insert Mode in the Code Editor without changing this default setting.)
Use Tab Character	Inserts tab character. If disabled, inserts space characters. If Smart Tab is enabled, this option is off.
Smart Tab	Tabs to the first non-whitespace character in the preceding line. If Use Tab Character is enabled, this option is off.
Optimal Fill	Begins every autoindented line with the minimum number of characters possible, using tabs and spaces as necessary.
Backspace Unindents	Aligns the insertion point to the previous indentation level (outdents it) when you press Backspace, if the cursor is on the first nonblank character of a line.
Cursor Through Tabs	Enables the arrow keys to move the cursor to the beginning of each tab.
Group Undo	Undoes your last editing command as well as any subsequent editing commands of the same type, if you press Alt+Backspace or choose Edit|Undo.
Cursor Beyond EOF	Positions the cursor beyond the end-of-file character.
Undo After Save	Allows you to retrieve changes after a save.
Keep Trailing Blanks	Keeps any blanks you might have at the end of a line.
BRIEF Regular Expressions	Uses BRIEF regular expressions.
Persistent Blocks	Keeps marked blocks selected even when the cursor is moved, until a new block is selected.
Overwrite Blocks	Replaces a marked block of text with whatever is typed next. If Persistent Blocks is also selected, text you enter is added to the currently selected block.
Double Click Line	Highlights the line when you double-click any character in the line. If disabled, only the selected word is highlighted.
Find Text At Cursor	Places the text at the cursor into the Text To Find list box in the Find Text dialog box when you choose Search|Find. When this option is disabled you must type in the search text, unless the Text To Find list box is blank, in which case the editor still inserts the text at the cursor.
Force Cut And Copy Enabled	Enables Edit|Cut and Edit|Copy, even when there is no text selected.
Use Syntax Highlighting	Enables syntax highlighting. To set syntax highlighting preferences, use the options from the Editor Display page.

Block Indent

Specify the number of spaces to indent a marked block. The default is 1; the upper limit is 16. If you enter a value greater than 16, you will receive an error.

Undo Limit

Specify the number of keystrokes that can be undone. The default value is 32,767 (32K).

Note: The undo buffer is cleared each time Delphi generates code.

Tab Stops

Set the character columns that the cursor will move to each time you press Tab. If each successive tab stop is not larger than its predecessor, you will receive an error. The default tab stops are 9 and 17.

Syntax Extensions

Specify, by extension, which files will display syntax highlighting information. The default extensions are .PAS, .DPR, .DFM, .INC, and .INT.

See also
Using the Code Editor

Display (Tools | Options)

See also Environment options

Use the Display page of the Enviroment Options dialog box to select display and font options for the Code Editor. The sample window displays the selected font.

The new settings take effect when you click OK.

Display and File check boxes

Configure the editor's display and choose how it saves files.

Check box	Effect
BRIEF Cursor Shapes	Uses BRIEF cursor shapes.
Create Backup File	Creates a backup file that replaces the first letter of the extension with a tilde (~) when you choose File|Save.
Preserve Line Ends	Preserves end-of-line position.
Zoom To Full Screen	Maximizes the Code Editor to fill the entire screen. When this option is off, the Code Editor does not cover the Delphi main window when maximized.

Keystroke Mapping

Enables you to quickly switch key bindings.

Mapping	Effect
Default	Uses key bindings that match CUA mappings (default)
Classic	Uses key bindings that match Borland Classic editor keystrokes
Brief	Uses key bindings that emulate most of the standard BRIEF keystrokes
Epsilon	Uses key bindings that emulate a large part of the Epsilon editor

Visible Right Margin

Check to display a line at the right margin of the Code Editor.

Right Margin

Set the right margin of the Code Editor. The default is 80 characters. The valid range is 0 to 1024. If you enter a value larger than 1024, you will receive an error.

Editor Font

Select a font type from the available screen fonts installed on your system (shown in the list). The Code Editor displays and uses only monospaced screen fonts, such as `Courier`. Sample text is displayed below the combo box.

Size

Select a font size from the predefined font sizes associated with the font you selected in the Font list box. Sample text is displayed below the combo box.

Sample text

DIsplays a sample of the select editor font and size.

See also

Code Editor

Using the Code Editor

Colors (Tools | Options)

See also Environment options

Use the Colors page of the Enviroment Options dialog box to specify how the different elements of your code appear in the Code Editor.

You can specify foreground and background colors for anything listed in the Element list box. The sample Code Editor shows how your settings will appear in the Delphi Code Editor.

Color SpeedSettings

Enables you to quickly configure the Code Editor display using predefined color combinations. The sample Code Editor shows how your settings will appear in the Delphi Code Editor.

Option	Effect
Defaults	Displays reserved words in bold. Background is white.
Classic	Displays reserved words in light blue and code in yellow. Background is dark blue.
Twilight	Displays reserved words and code in light blue. Background is black.
Ocean	Displays reserved words in black and code in dark blue. Background is light blue.

Element

Specifies syntax highlighting for a particular code element. You can choose from the Element list or click the element in the sample Code Editor.

The element options are:

Whitespace	Marked Block
Comment	Search Match
Reserved Word	Execution Point (for debugging)
Identifier	Enabled Break (for debugging)
Symbol	Disabled Break (for debugging)
String	Invalid Break (for debugging)
Number	Error Line
Assembler	Right Margin
Plain Text	

Color Grid

Sets the foreground (FG) and background (BG) colors for the selected code element.

To select a color using the mouse, choose one of the following methods:

- Click a color to select it as the foreground color.
- Right-click a color to select it as the background color.

If you choose the same color for the foreground and the background, it is marked as FB. (This is not recommended.)

To select the color using the keyboard,

1. Use the arrow keys to highlight a color.
2. Press F to select it as the foreground color, or B to select it as the background color.

Text Attributes check boxes

Specify format attributes for the code element. The attribute options Delphi supports are:

- Bold
- Italic
- Underline

Use defaults for check boxes

Display the code element using default Windows system colors (foreground, background, or both).

Unchecking either option restores the previously selected color or, if no color has been previously selected, sets the code element to the Windows system color.

Note: To change the Windows system colors, use Control Panel in the Windows Program Manager.

See also

Code Editor

Using syntax highlighting

Using syntax highlighting

See also

Syntax highlighting changes the colors and attributes of your code in the editor, making it easier to quickly identify parts of your code.

To enable syntax highlighting,

- On the Editor Options page of the Options|Environment dialog box, check the Use Syntax Highlight option.

To change the syntax highlighting colors for elements of your code,

- Use the Editor Colors page of the Options|Environment dialog box.

See also
Code Editor

Tools|Tools

Choose Tools|Tools to display the Tools Options dialog box.
You use the Tools Options dialog box to add, delete, or edit programs on the Tools menu.

Tools Options dialog box

Choose Tools|Tools to open the Tools Options dialog box.

Use this dialog box to add programs to, delete programs from, or edit programs on the Tools menu.

Tools Options dialog box

Tools

Lists the programs currently installed on the Tools menu.

Add

Click Add to display the Tool Properties dialog box, where you can specify a menu name, a path, and startup parameters for the program.

Delete

Clic-k Delete to remove the currently selected program from the Tools menu.

Edit

Click Edit to display the Tool Properties dialog box, where you can edit the menu name, the path, or the startup parameters for the currently selected program.

Arrow

Use the arrow buttons to rearrange the programs in the list. The programs appear on the Tools menu in the same order they are listed in the Tool Options dialog box.

To add a program to the tools menu,

- Choose Add. Delphi displays the Tool Properties dialog box, where you specify information about the application you are adding.

To delete a program from the tools menu,

- Select the program to delete, and choose Delete. Delphi prompts you to confirm the deletion.

To change a program on the tools menu,

- Select the program to change, and choose Edit. Delphi displays the Tool Properties dialog box with information for the selected program.

See also

Adding programs to the Tools menu

Tool Properties dialog box

Use the Tool Properties dialog box to enter or edit the properties for a program listed on the Tools menu.

To display the Tool Properties dialog box,

- Click Add or Edit in the Tool Options dialog box.

Dialog box options

Title

Enter a name for the program you are adding. This name will appear on the Tools menu.

You can add an accelerator to the menu command by preceding that letter with an ampersand (&). If you specify a duplicate accelerator, Delphi displays a red asterisk (*) next to the program names in the Tool Options dialog box.

Program

Enter the location of the program you are adding. You can include the full path to the program. Click the Browse button to search your drives and directories to locate the path and file name for the program.

Working Dir

Specify the working directory for the program. Delphi specifies a default working directory when you select the program name in the Program Edit Box. You can change the directory path if needed.

Parameters

Enter parameters to pass to the program at startup. For example, you might want to pass a file name when the program launches. You can type the parameters or use the Macros button to supply startup parameters. You can specify multiple parameters and macros.

Browse

Click Browse to select the program name for the Program edit box. When you click Browse, the Select Transfer Item dialog box opens.

Macros

Click Macros to expand the Tool Properties dialog box to display a list of available macros. You can use these macros to supply startup parameters for your application.

To add a macro to the list of parameters,

- Select a macro from the list and click Insert.

Transfer macros

Use transfer macros to supply startup parameters to a program on the Tools menu.

To display the macros,

- Click the Macros button on the Tool Properties dialog box.

The transfer macros are:

Macro	Description
$COL	Expands to the column number of the cursor in the active Code Editor window. For example, if the cursor is in column 50, at startup Delphi passes "50" to the program.
$ROW	Expands to the row number of the cursor in the active Code Editor window. For example, if the cursor is in row 8, at startup Delphi passes "8" to the program.
$CURTOKEN	Expands to the word at the cursor in the active Code Editor window. For example, if the cursor is on the word Token, at startup Delphi passes "Token" to the program.
$PATH	Expands to the directory portion of a parameter you specify. When you insert the $PATH macro, Delphi inserts $PATH() and you specify a parameter within the parentheses. For example, if you specify $PATH($EDNAME), at startup Delphi passes the path for the file in the active Code Editor window to the program.
$NAME	Expands to the file name portion of a parameter you specify. When you insert the $NAME macro, Delphi inserts $NAME() and you specify a parameter within the parentheses. For example, if you specify $NAME($EDNAME), at startup Delphi passes the file name for the file in the active Code Editor window to the program.
$EXT	Expands to the file extension portion of a parameter you specify. When you insert the $EXT macro, Delphi inserts $EXT() and you specify a parameter within the parentheses. For example, if you specify $EXT($EDNAME), at startup Delphi passes the file extension for the file in the active Code Editor window to the program.
$EDNAME	Expands to the full file name of the active Code Editor window. For example, if you are editing the file C:\PROJ1\UNIT1.PAS, at startup Delphi passes "C:\PROJ1\UNIT1.PAS" to the program.
$EXENAME	Expands to the full file name of the current project executable. For example, if you are working on the project PROJECT1 in C:\PROJ1, at startup Delphi passes "C:\PROJ1\PROJECT1.EXE" to the program.
$PARAMS	Expands to the command-line parameters specified in the Run Parameters dialog box.
$PROMPT	Prompts you for parameters at startup. When you insert the $PROMPT macro, Delphi inserts $PROMPT() and you specify a default parameter within the parentheses.
$SAVE	Saves the active file in the Code Editor.
$SAVEALL	Saves the current project.
$TDW	Sets up your environment for running Turbo Debugger. For example, this macro saves your project, ensures that your project is compiled with debug info turned on, and recompiles your project if it is not compiled with debug info turned on. Be sure to use this macro if you add Turbo Debugger to the Tools menu.

Select Transfer Item dialog box

Use the Select Transfer Item dialog box to search drives and directories for a program to add to the Tools menu.

To locate a transfer item,

- Click the Browse button on the Tool Properties dialog box.

File Name

Enter the name of the file you want to load, or enter wildcards to use as filters in the Files list box.

Files

Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type

Choose the type of file you want to open. The default file types are .EXE, .COM, and .PIF files. All files in the current directory of the selected type appear in the Files list box.

Directories

Select the directory whose contents you want to view. Files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Tools | Repository

Choose Tools|Repository to display the Object Repository dialog box. You use the Object Repository dialog box to add, delete, and rename pages in the Object Repository. In addition you can edit and delete Object Repository items. You can also specify template and expert options for forms and projects

See also

Object Repository dialog box

About the Object Repository

Object Repository dialog box

Choose Tools|Repository to display the Object Repository dialog box.

The settings in the Object Repository Options dialog box affect the behavior of Delphi when you begin a new project or create a new form in an open project. This is where you specify

- Default project
- Default new form
- Default main form

You always have to option to override these defaults by choosing File|New and selecting from the New Items dialog box.

By default, opening a new project displays a blank form. You can change this default behavior by changing Object Repository options.

Object Repository dialog box

Pages

This list box displays the pages in the Object Repository. When you select a page, the items on that page appear in the Objects list box. Select [Object Repository] to view all items in the Object Repository.

Objects

The Objects list box displays the items on the currently selected page of the Object Repository.

Add Page button

Use the Add Page button to add a new blank page to the ObClick here to add apage.

To add a page,

1 Click the Add Page button.

The Add Page dialog box appears.

2 Type the name of the page you want to add.

3 Click OK.

Delete Page button

Use the Delete Page to remove an item from the Object Repository.

To delete a page,

1 In the Pages list box, select the name of the page you want to delete.

2 Click the Delete Page button.

The selected page is removed from the Object Repository.

Rename Page button

Use the Rename Page button to rename an item in the Object Repository.

To rename a page,

1 In the Pages list box, select the name of the page you want to rename.

1 Click the Rename Page button.

The Rename Page dialog box appears.

2 Type the name of the page you want to rename.

3 Click OK.

The renamed page appears in the Pages list box.

Edit Object

Use the Edit Object button to edit the properties of items in the Object Repository.

To edit an object,

1 From the Objects list box, select the item you want to edit.

2 Click the Edit Object button.

The Edit Object Info dialog box appears.

3 Edit the information as desired.

4 Click OK.

Delete Object

Use the Delete Object button to remove items from the Object Repository.

Object Repository Options

The settings in the Object Repository Options dialog box affect the behavior of Delphi when you begin a new project or create a new form in an open project. When you select an item in the Objects list box, the appropriate options become available at the bottom of the Objects list box. Depending on the item you select, one or more of the options listed below become available.

- Default project
- Default new form
- Default main form

You always have the option to override these defaults by choosing File|New and selecting from the New Items dialog box.

By default, opening a new project displays a blank form. You can change this default behavior by changing Object Repository options. For more information see Customizing the Object Repository.

Up arrow and down arrow

Click the up arrow or the down arrow to change the position of the selected page. You can also move pages using a drag-and-drop operation.

Database Desktop

The Database Desktop (DBD) is a database maintenance and data definition tool. It enables you to query, restructure, index, modify, and copy database tables, including Paradox, dBASE, and SQL tables. It also enables you to create new Paradox and dBASE tables. You do not have to own Paradox or dBASE to use the DBD with desktop files in these formats.

The DBD can copy data and data dictionary information from one format to another. For example, you can copy a Paradox table to an existing database on a remote SQL server. For a complete description of the DBD, see Database Desktop Contents.

To open the Database Desktop,

- Choose Tools|Database Desktop.

Add Page dialog box

You access the Add Page dialog box from the Object Repository dialog box. Use the Add Page dialog box to add a page to the Object Repository.

Add Page dialog box

Page name

Type the name of the new page into the Page name text box.

Rename Page dialog box

You access the Rename Page dialog box from the Object Repository dialog box. Use the Rename Page dialog box to rename a page in the Object Repository.

Rename Page dialog box

Page name

Type the new name of the page into the Page name text box.

Edit Object Info dialog box

Use the Edit Object Info dialog box to edit information of Object Repository items.

Edit Object Info dialog box

Title

This text box displays the title of the selected item.

Description

This text box displays the description of the selected item.

To view the item description,

1 From the File menu select New.

2 Select an item in the New Items dialog box.

3 Right-click the mouse.

4 Select View Details from the SpeedMenu.

The item description appears in the Description column.

Page

Displays the current page containing the selected item.

To change the page on which the item appears, select a different page from the Page drop-down list.

Author

Displays the name of the Author of the selected item.

Browse button

The icon of the selected item is displayed to the left of the Browse button. Use the Browse button to select a different icon.

Adding programs to the Tools menu

See also

The commands on the Tools menu transfer execution to an external application. You can add programs to, delete programs from, or edit programs on the Tools menu.

To add a program to the tools menu,

1. Choose Tools|Tools.

   Delphi displays the Tool Options dialog box, which lists the programs currently on the Tools menu.
2. Choose Add.

   Delphi displays the Tool Properties dialog box.
3. Specify a title for the program. The title you specify will be listed on the Tools menu.
4. Specify the program file or choose Browse to select it from a list.
5. Specify the working directory for the program, if necessary.
6. Specify startup parameters for the program, if necessary. You can type the parameters or use the Macros button to supply startup parameters. You can specify multiple parameters and macros.
7. Choose OK.

   Delphi closes the Tool Properties dialog box. The new program is on the Tools list in the Tool Options dialog box.
8. Choose Close.

   Delphi closes the Tool Options dialog box. The new program is on the Tools menu.

See also

Tool Properties dialog box
Transfer macros

View menu

See also

Use the commands on the View menu to display or hide different elements of the Delphi environment and open windows that belong to the integrated debugger.

The commands on the View menu are:

Project Manager	Displays the Project Manager
Project Source	Displays the project file in the Code Editor
Object Inspector	Displays the Object Inspector
Alignment Palette	Displays the Alignment Palette
Browser	Displays the Object Browser
Breakpoints	Displays the Breakpoints List dialog box
Call Stack	Displays the Call Stack dialog box
Watches	Displays the Watch List dialog box
Threads	Displays the threads status box
Component List	Displays the Components dialog box
Window List	Displays a list of open windows
Toggle Form/Unit	Toggles the inactive form or unit window active
Units	Displays the View Unit dialog box
Forms	Dislpays the View Form dialog box
New Edit Window	Opens a new Code Editor
SpeedBar	Hides or shows the SpeedBar
Component Palette	Hides or shows the Component Palette

See also
Aligning components

View | Project Manager

Choose View|Project Manager to open the Project Manager. If the Project Manager is already open, it becomes the active window.

Use the Project Manager window to add, delete, save, or copy a file to the current project. The Project Manager also contains a listing of all the units and their associated form used in the current project.

Note: Units are listed in the Project Manager even if they are not currently open.

You can position the Project Manager anywhere on your desktop.

View | Project Source

See also

Choose View|Project Source to make the project file for the open project, the active page in the Code Editor. If the project file is not currently open when you choose this command, Delphi will open it for you.

An alternative way to perform this command is

- Choose View Project from the Project Manager SpeedMenu.

See also

Project file

Project Manager

View | Object Inspector

Choose View|Object Inspector to toggle between the Object Inspector and the last active form or Code Editor file. If you have closed the Object Inspector, choose this command to reopen the it.

Use the Object Inspector to edit property values and event-handler links.

View | Alignment Palette

See also

Choose View|Alignment Palette to display the Alignment Palette, which you can use to align components to the form, or to each other.

Note: You can also align components by using the Alignment dialog box.

Alignment palette

See also

Use the Alignment palette to align components to the form, or to each other.

To open the Alignment palette,

- Choose View|Alignment Palette.

The Alignment palette has Tool Help for each button.

The icons on the Alignment palette are:

Icon	Effect
	Aligns the selected components to the left edge of the component first selected. (Not applicable for single components.)
	Moves the selected components horizontally until their centers are aligned with the component first selected. (Not applicable for single components.)
	Aligns the selected component(s) to the center of the form along a horizontal line.
	Aligns the selected components to the right edge of the component first selected. (Not applicable for single components.)
	Aligns the selected components to the top edge of the component first selected. (Not applicable for single components.)
	Moves the selected components vertically until their centers are aligned with component first selected. (Not applicable for single components.)
	Aligns the selected component(s) to the center of the form along a vertical line.
	Aligns the selected components to the bottom edge of the component first selected. (Not applicable for single components.)

If you are unsure of how a particular button on the Alignment palette will act, you can click the button, and the icon on the button will change to show you how it will align the selected components.

Note: You can also use the Alignment dialog box to align components.

See also

Aligning components
Selecting components
Using the grid to align components

View | Watches

Choose View|Watches to open the Watch List window.

The Watch List window displays all the currently set watch expressions.

If you keep this window open during your debugging sessions, you can monitor how your program updates the values of important variables as the program runs.

View | Threads

Choose View|Threads to view the Threads status box.

Use this status box to view the status of all the threads currently executing in the application being debugged.

Threads status box

Choose View|Threads to view the Threads status box.

Use this status box to view the status of all the threads currently executing in the application being debugged.

When a debug event occurs (breakpoint, exception, paused), the thread status view will indicate the status of each thread as it executes. Using the local menu, the user can make a different thread current. When a thread is marked as current, the next step or run operation is relative to that thread.

Threads status box

State

The thread state is either Running or Stoped.

Status

The thread status displays one of the following:

Breakpoint	The thread stopped due to a breakpoint.
Faulted	The thread stopped due to a processor exeception.
Unknown	The thread is not the current thread so its status is unknown.
Stepped	The last step command was successfully completed.

Location

Displays the source position. Displays the address if there is no source location available.

Thread ID

Displays the OS assigned thread ID.

View | Breakpoints

Choose View|Breakpoints to open the Breakpoint List window.

The Breakpoint List window lists all currently set breakpoints.

Each breakpoint listing shows the following:

- The file in which the breakpoint is set
- The line number of the breakpoint
- Any condition or pass count associated with the breakpoint

View | Call Stack

Choose View|Call Stack to open the Call Stack window.

The Call Stack window lists the current sequence of routines called by your program. In this listing, the most recently called routine is at the top of the window, with each preceding routine call listed beneath.

Each entry in the Call Stack window displays the procedure name and the values of any parameters passed to it.

View | Browser

See also

Choose View|Browser to open the ObjectBrowser window.

If the Browser command is dimmed, compile your program before opening the ObjectBrowser.

The ObjectBrowser provides enables you to visually explore the objects, units, and hierarchies of classes and methods used in your application.

See also
ObjectBrowser window

View | Component List

See also

Choose View|Component List to display the Components window.

Components window

Use this window to add components to your forms using the keyboard.

Dialog box options

Search By Name

Enter the name of the component you want to add. This list box performs an incremental search so that the cursor moves to the first component containing the letters you type.

Component

Select the component you want to add. Components are listed in alphabetical order, and their button representation is on the left.

When you select a component, its name appears in the Search edit box.

Add To Form

Click Add To Form to place an instance of the selected component in the center of the form. You can select Add To Form by pressing Enter.

To add the component you selected in the Component list box, do one of the following,

- Press Enter.
- Double-click the component name.
- Click the Add to Form button.

Note: When you add a component to a form using the keyboad, Delphi uses the default component size and adds the component to the center of the form unless a container component (such as a group box or panel) is selected.

If a container component is selected, Delphi places the component you are adding in the center of that container. To add a component into a container you must select the container before selecting Add To Form.

See also

Adding components using the keyboard

View | Toggle Form/Unit

See also

Choose View|Toggle Form/Unit to display either the inactive form or the unit associated with the active form or unit.

Alternative ways to perform this command are:

- Choose the Toggle Form/Unit button from the SpeedBar.
- Choose View Unit or View Form from the Project Manager SpeedMenu.

See also

Viewing units and forms in a project

View|Forms

View|Units

View | Units

See also

Choose View|Units to display the View Unit dialog box.

View Unit dialog box

Use this dialog box to open the project file or any unit in the current project. When you choose a unit, it becomes the active page in the Code Editor.

If the unit you want to open is not currently open, Delphi will open it.

Alternative ways to perform this command are:

- Choose the Select Unit From List button from the SpeedBar.
- Choose View Unit from the Project Manager SpeedMenu.

See also

Project file
Unit file
Viewing units and forms in a project

View | Forms

See also

Choose View|Forms to display the View Form dialog box.

View Form dialog box

Use this dialog box to quickly open any form in the current project. When you select a form, it becomes the active form, and its associated unit becomes the active module in the Code Editor.

If the associated unit is not open when you select a form, Delphi opens it.

Alternative ways to perform this command are:

- Choose the Select Form From List button from the SpeedBar.
- Choose View Form from the Project Manager SpeedMenu.

See also

Form

Viewing units and forms in a project

View | Window List

Choose View|Window List to display the Window List dialog box.

Window List dialog box

Use this dialog box to make an inactive Delphi window active. If you have a lot of windows open, this is the easiest way to locate a specific window. The Windows List dialog box displays all the open Delphi windows.

To select a window, do one of the following:

- Double-click the window name.
- Select the window name and click OK.

View | New Edit Window

Choose View|New Edit Window to open a new Code Editor that contains a copy of the active page from the original Code Editor.

Any changes you make to either the original or the copy are reflected in both files.

So that you can distinguish between the windows, the caption in the original window is postfixed with a 1, the first copy with a 2, the second copy with a 3, and so on.

View | SpeedBar

Choose View|SpeedBar to show or hide the SpeedBar.
When this command is checked, the SpeedBar is visible.
An alternitive way to hide the SpeedBar is,

- Choose Hide from the SpeedBar SpeedMenu.

View | Component Palette

Choose View|Component Palette to show or hide the Component palette.

When this command is checked, the Component palette is visible.

An alternitive way to hide the Component palette is,

- Choose Hide from the Component palette SpeedMenu.

Using DDE

See also

Dynamic Data Exchange (DDE) sends data to and receives data from other applications. With Delphi, you can use this data to exchange text with other applications. You can also send commands and macros to other applications, so your application can control other applications.

Here is a typical way to use DDE: a link between two applications is established, either by your application or the other application. Once this link (called a conversation) is established, the two applications can continuously and automatically send text data back and forth. When the text changes in one application, DDE automatically updates the text in the other.

To understand DDE applications, you need to become familiar with the concept of DDE conversations.

When to use DDE

You want to use DDE when exchanging distinct text strings. If all you want to know is the bottom line of a profits spreadsheet, it makes sense to link the cell that contains the bottom line to a Delphi DDE client application.

You could then output the data in an edit box or label. DDE protects the data in the spreadsheet by not allowing the user to activate and edit the spreadsheet from your client application.

Note: Not all applications support DDE. To determine whether an application supports DDE, refer to its documentation.

See also

Creating DDE client applications

Creating DDE server applications

DDE conversations

See also

DDE conversations consist of a DDE client application and a DDE server application. With Delphi, you can create both DDE clients and DDE servers. In fact, a single Delphi application can be both a DDE client and a DDE server at the same time.

A DDE conversation is defined by the following three characteristics:

- DDE services
- DDE topics
- DDE items

Note: See the documentation for the DDE server for specific information about specifying the services, topics, or items of a conversation.

DDE services

The service of a conversation is usually the name of the DDE server application's main executable file without the .EXE extension.

Sometimes the service name can differ from the main executable file name.

When the server is a Delphi application, the service is the project name without the .DPR or .EXE extension.

Note: Sometimes DDE services are called application names. The terminology is interchangeable.

DDE topics

The topic of a DDE conversation is a unit of data, identifiable to the server, containing the linked text. Typically, the topic is a file.

When the server is a Delphi application, the topic is either the Caption of the form containing the data you want to link (if a TDDEServerConv component has not been used) or the Name of the DDE server conversation component (if a TDDEServerConv component has been used).

DDE items

The item of a DDE conversation identifies the actual piece of data to link, for example, spreadsheet cells or database fields.

The syntax used for specifying the DDE item depends on the DDE server application.

When the server is a Delphi application, the item is the Name of the linked TDDEServerItem component.

See also

Creating DDE client applications
Creating DDE server applications
Using DDE

Creating DDE client applications

See also Example

You can create a DDE client by adding a DDE client conversation (TDDEClientConv) component and a DDE client item (TDDEClientItem) component to a form.

Client applications can poke data (send data to the server) with the PokeData or PokeDataLines method.

Clients can control the server by running it or sending macros with the ExecuteMacro or ExecuteMacroLines method.

To create a DDE client,

1. Add a DDE client conversation component (TDDEClientConv) and a DDE client item component (TDDEClientItem) to a form.
2. Assign the name of the conversation component to the DDEConv property of the client item component.

- To establish a link at design time, choose this value from a list of possible conversations for DDEConv in the Object Inspector.
- To establish a link at run time, your application must execute code that assigns the value to the DDEConv property.

Example

The following example links an item component named DDEClientItem1 to a conversation component named DDEClientConv1:

```
DDEClientItem1.DDEConv := 'DDEClientConv1';
```

See also

Controlling other applications with DDE
Creating DDE server applications
Establishing a link with a DDE server
Poking data
Using DDE

Establishing a link with a DDE server

If you have access to the DDE server application and data, you can establish a DDE link by pasting it from the Clipboard at design time.

To establish a DDE link at design time,

1. Activate the server application and select the data to link to your client application.
2. Copy the data and DDE link information to the Clipboard from the server application by choosing Copy from the Edit menu of the server.
3. Activate Delphi and select the DDE client conversation component.
4. Click the ellipsis button for either the DDEService or the DDETopic property in the Object Inspector.

   The DDE Info dialog box appears.
5. Choose Paste Link.

   The service and topic fill in with the correct values automatically. If the Paste Link button is disabled, then the application you intended to be the server does not support DDE, or the DDE information was not successfully copied to the Clipboard.
6. Choose OK.

   The DDEService and DDETopic properties now contain the appropriate values to establish a DDE link.
7. Select the DDE client item component and choose the name of the linked DDE client conversation component for the DDEConv property from the list in the Object Inspector.
8. If the Clipboard still contains the DDE link information, choose the appropriate value for the DDEItem property from the list in Object Inspector. Otherwise, type the correct value for the DDEItem property.

To establish a DDE link at run time,

1. Specify the DDE service and topic with the SetLink method of the DDE client conversation component.

   The following example establishes a link to a Borland Paradox 5.0 table named GADGETS.DB in the working directory:

   ```
   DDEClientConv1.SetLink('PDOXWIN', ':WORK:GADGETS.DB');
   ```
2. Assign the item to the DDEItem property of the DDE client item component.

   The following example establishes a link to the Price field of the Paradox table:

   ```
   DDEClientItem1.DDEItem := 'PRICE';
   ```

Processing DDE linked data

Example

Before you can process data from a DDE server, you first need to establish a DDE link.

After establishing a DDE link, the linked data appears in the Text property of the DDE client item component. (If the data is too long to be stored in a string, it is stored in the Lines property. The data is continuously updated by the DDE server, and an OnChange event of the client item component occurs whenever the data changes.

To process linked text data,

1. Add an edit box (TEdit component) to your form.
2. Write a statement that assigns the value of the Text property to the Text property of the edit box.

   Attach the assignment statement to the OnChange event handler of the DDE client item.

Example

The following event handler assigns the text of the DDE client item to an edit box.

```
procedure TForm1.DDEClientItem1.Change(sender: TObject);
begin
  Edit1.Text := DDEClientItem1.Text;
end;
```

Poking data

Example

Poking data means sending data from your DDE client application to the DDE server application, which is opposite the usual data flow direction for DDE.

- To poke data, call the PokeData method of a DDE client conversation component. To poke text data that is too long to be contained in a string, use PokeDataLines.

PokeData has two parameters:

- The first parameter specifies the item of the DDE conversation (specified in the DDEItem property of the associated DDE client item component).
- The second parameter is a string containing the text to send.

Example

The following example sends the text 'Hello' from a DDE client conversation component named DDEClientConv1 to a linked DDE server. The string is inserted into the DDE item specified in the DDEItem property of the DDE client item component named DDEClientItem1:

```
DDEClientConv1.PokeData(DDEClientItem1.DDEItem, 'Hello');
```

Controlling other applications using DDE

Example

All DDE client applications can control DDE server applications. When your DDE client tries to establish a link with a DDE server that is not running, the client activates the server and loads the conversation topic (specified in the DDETopic property).

The ConnectMode property of a DDE client conversation component has two possible values:

Value	When active
ddeAutomatic	Your client will run the server upon run-time creation of the form containing the DDE client conversation component.
ddeManual	Your application must execute the OpenLink method of the DDE client conversation component.

Using macros

Another way to control other applications is to execute macro commands. Use the ExecuteMacro method of the DDE client conversation component to send a string containing one or more macro commands to the server. The server then processes the macro. To send a list of macro strings to the DDE server, use ExecuteMacroLines

Note: Not all DDE servers can process macros. See the documentation for the server application to determine whether it supports macros and for its macro syntax.

Example

The following example uses macros to tell Microsoft Excel 4.0 to close its active worksheet by executing the following code in your client application, assuming your DDE client conversation component is named DDEClientConv1:

```
DDEClientConv1.ExecuteMacro('[FILE.CLOSE()]', False);
```

Creating DDE server applications

Example

DDE server applications respond to DDE client. Typically, they contain data that the client application needs to access. Servers simply update clients.

If you want to handle macros sent by the DDE client, use both a TDDEServerItem and a TDDEServerConv to create the DDE server. Then, you can use the OnExecuteMacro event of the DDE server conversation component to process the macro. Also, use both components if you want the Name of the DDE server conversation component to be the topic of the DDE conversation. With only a DDE server item component, the topic of the conversation is the Caption of the form containing the DDE server item.

To create a DDE server using only a DDE server item component,

- Add a DDE server item (TDDEServerItem) component to a form.

To create a DDE server using a DDE server conversation component,

1. Add a DDE server conversation (TDDEServerConv) component and a DDE server item component to a form.
2. Assign the name of the conversation component to the ServerConv property of the item component.

- To establish a link at design time, choose this value from a list of possible conversations for ServerConv in the Object Inspector.
- To establish a link at run time, your application must execute code that assigns the value to the ServerConv property.

Example

The following example links an item component named DDEServerItem1 to a conversation component named DDEServerConv1:

```
DDEServerItem1.ServerConv := 'DDEServerConv1';
```

Establishing a link with a DDE client

Example

Linking a DDE server to a DDE client enables your client application to share data with the client application.

To establish a DDE link,

1. Use the CopyToClipboard method of the DDE server item component to copy the value of the Text property (or Lines property), along with DDE link information, to the Clipboard.
2. Insert the linked data into the DDE client application. Typically, do this by choosing the appropriate command (such as Edit| Paste Special or Edit|Paste Link) of the client application.

Note: The method for establishing a DDE link depends on the DDE client application. See the documentation for the client for specific information about establishing DDE links. If the DDE client is another Delphi application, see Establishing a link with a DDE server.

Example

The following example creates a link from a DDE server item component named DDEServerItem1 to a WordPerfect 6.0 document. If you do not have WordPerfect, this example is worth examining because the steps required are probably similar for any other DDE client application that can paste links.

1. At run time, your DDE server application should execute the following code:

   ```
   DDEServerItem1.CopyToClipboard;
   ```

2. Activate WordPerfect and choose Edit | Paste Special.

   The WordPerfect Paste Special dialog box appears.

3. Choose Paste Link.

   The Paste Special dialog box closes, and the linked text from the Value property of DDEServerItem1 will appear at the insertion point in the WordPerfect document. When the Value property changes, the text in the WordPerfect document will be updated automatically.

Using OLE

See also

To use OLE, one application must be an OLE server, and another application must be an OLE container. With Delphi, you can create OLE container applications to hold OLE objects.

Here is a typical way to use OLE: the OLE container application displays a picture representing the OLE object. The user activates the OLE object, typically by double-clicking the picture. When the OLE object is activated, the OLE server application opens, and the user can edit the OLE object using the OLE server. After editing, the user updates the OLE object within the OLE container and closes the OLE server.

When to use OLE

You want to use OLE when you are exchanging complex information, such as sounds and documents. Control transfers to the OLE server application when you activate an object in your OLE container, so the user can access all the functionality of the server application from within your container application.

The server does all the processing, and you do not have to program your container to edit the OLE object.

Note: Not all applications support OLE. To determine whether an application supports OLE, you should refer to its documentation.

See also

Linking versus embedding
OLE 1.0 versus OLE 2.0
OLE data in files
OLE classes, documents and items

OLE 1.0 versus OLE 2.0

See also

There are currently two versions of OLE:

- OLE 1.0
- OLE 2.0

OLE 1.0

When the user activates an OLE object that was created with OLE 1.0 server application, the server application opens in its own window in the foreground and receives focus. The OLE container application exists in a separate window in the background.

OLE 2.0

When the user activates an OLE object that was created with an OLE 2.0 server application, the object might be activated in place. In-place activation means the server's menu and tool bar replace the OLE container's. The object could be edited from within the OLE container window, but all the processing is handled by the OLE server.

See also
OLE classes, documents and items

Linking versus embedding

See also

Data in a linked OLE object is maintained by the OLE server that created it.

Data in an embedded OLE object is maintained by the OLE container application.

Linked objects

Linked objects must be stored in files. You cannot create a new OLE link unless the OLE object has previously been saved to a file from the OLE server application.

The data in a linked OLE object file can be modified by your OLE container application and by other applications. The OLE server application and other OLE container applications can all access and modify the OLE object. Data can exist in one location but be accessible from multiple applications.

You can program your OLE container application in Delphi to continually obtain the latest data from the OLE object file. When the OLE object data is modified, even by other applications, the changes appear in all OLE container applications that contain a link to the file, including your own.

Embedded objects

Embedded objects are stored in your OLE container application, and other OLE container applications cannot access that OLE object. The only time another application can edit the embedded OLE object is when the user activates the object from your OLE container application. Only then can the OLE server edit the OLE object data.

Embedded OLE objects do not need to exist in files. All the data can be stored in your container application. This ensures that the OLE data does not accidentally get deleted, modified, or corrupted by being stored in an external file.

The drawback to embedding objects is that the size of your OLE container application increases by the size of the included OLE data.

If you want the changes you make to the embedded OLE object to appear the next time you run your application, you need to save the OLE data to a file.

The following table lists guidelines for when to link or embed OLE objects:

When to link	When to embed
You want to be able to make changes to the original object and have those changes reflected in all applications or documents linked to that original.	You want to be able to make changes to the object and have those changes reflected only in one particular application or document.
The original object is likely to be frequently modified, or modified from multiple OLE container applications.	The original object is unlikely to be frequently modified, or modified from only one instance of an OLE container application.
The original object file is unlikely to be frequently moved, and will not be deleted.	The original object file is likely to be frequently moved, or possibly deleted.
The object is large, and you will be distributing it via a network or electronic mail.	The object is small, or the object is large but you will not be distributing it via a network or electronic mail.
You need to conserve disk space by having only one copy of the object.	You do not need to conserve disk space by having only one copy of the object.

See also

OLE data in files

Using OLE

OLE classes, documents and items

OLE classes, documents and items

OLE classes, documents and items define the OLE object.

OLE classes

The class of an OLE object determines the OLE server application that created the OLE object.

The OLE class is the name of the OLE server application. For example, if the OLE object is a Novell Quattro Pro 6.0 notebook, the class determines that the server application is Quattro Pro.

Sometimes, you can link or embed more than one type of object from an OLE server application. For example, you could link or embed an entire spreadsheet, a range of cells, or a graph from a spreadsheet application.

The OLE class also determines which type of data the OLE object contains.

For an an OLE container component, the OLE class is determined by the OBJClass property.

OLE documents

The document of an OLE object determines the source file that contains the data for the OLE object. The object document must be used for a linked object, because linked objects must exist in files. The object document is used for an embedded object only if you create the object from an existing source file. If you create a new embedded object that does not yet exist in a file, you would not specify an OLE document.

For example, if the OLE object is linked to the Novell Quattro Pro 6.0 notebook SALES.WB1, the OLE document determines that the data is stored in the file named SALES.WB1.

For an an OLE container component, the OLE class is determined by the ObjDoc property.

OLE items

The item of an OLE object determines what portion of an OLE document contains the data to link or embed. Items are used when you want the OLE object to contain a smaller piece of data than an entire document file.

For example, if the OLE object is linked to the cell range E2 to F5 of a Novell Quattro Pro 6.0 notebook, the OLE document determines that the data is stored only in cells E2 to F5 rather than the entire notebook.

For an an OLE container component, the OLE item is determined by the ObjItem property.

Note: You do not need to use the OLE item if the OLE object will not contain a piece of data more specific than the file specified in the OLE document.

See also

OBJClass property
OBJDoc property
OBJItem property

OLE data in files

Example

To save changes made to an OLE object from within your OLE container application, you need to save the object data to a file.

If the object is linked, the data is automatically stored in the original source file, and the object is automatically updated every time the data changes.

If the object is embedded, your application must save the data in a special OLE format.

OLE data file format

The OLE data file is not in the same format used by the application that created the original OLE object. For example, if your container application saves an embedded spreadsheet to a file, the OLE server spreadsheet application would not be able to open your file as a spreadsheet. Only your OLE container application can use the data saved in this file.

- To save an OLE object, call the SaveToFile method of the OLE container component.
- To obtain OLE data from a file saved with the SaveToFile method, use the LoadFromFile method.

If you attach a LoadFromFile call to the OnCreate event of the form containing the OLE container component, your application restores the latest version of the OLE object every time you run your OLE container application.

Example

Example for saving OLE data to a file

Example for loading OLE data from a file

Example

The following example stores the OLE data in a file called SALES.OLE in the C:\TIPPER directory.

```
OleContainer1.SaveToFile('C:\TIPPER\SALES.OLE');
```

Example

The following example loads the OLE data from the file called SALES.OLE in the C:\TIPPER directory.

```
OleContainer1.LoadFromFile('C:\TIPPER\SALES.OLE');
```

Using the Printer object

See also Example TPrinter reference

The Printer object provides several methods and properties that enable you to control the printing of documents from your application. These methods and properties interact with the Print and Printer Setup common dialog boxes.

Canvas

The canvas represents the surface of the currently printing document. You assign the contents of your text file to the Canvas property of the printer object by using AssignPrn. The printer object then directs the contents of the Canvas property (your text file) to the printer.

Fonts

Represents the list of fonts supported by the current printer. These fonts appear in the Font list of the Font dialog box.

Any font selected from this dialog box is reflected back into the Font property for the memo component that contains the text you want to print. However, the printer object has no such relationship to the Font dialog box or to the Font property for the memo. Unless your program specifies otherwise, the printer uses the default (System) font that is returned by the Windows device driver to print your text file.

To change the printer's font,

Assign the Font property for the memo component (or other component whose text you want to print) to the Font property for the printer object's Canvas. This downloads the selected font to the printer.

See also

Printing the contents of a memo

Printing the contents of a memo

See also Example

To print the contents of a memo component,

1. Assign a text-file variable to the printer by calling AssignPrn.
2. Create and open the output file by calling Rewrite.

   Any Write or Writeln statements sent to the file variable are then written on the Canvas of the printer object.

The AssignPrn procedure is declared in the Delphi Printers unit, so you must add Printers to the **uses** clause of the unit that calls AssignPrn.

When the printer is ready for input, you can write the contents (Lines property) of the memo to the printer.

See also
Using the printer object

Example

The following example prints the contents of a Memo field when the user chooses File|Print.

```
procedure TForm1.Print1Click(Sender: TObject);
var
  Line: Integer;
  PrintText: TextFile;   {declares a file variable}
begin
  if PrintDialog1.Execute then
  begin
    AssignPrn(PrintText);   {assigns PrintText to the printer}
    Rewrite(PrintText);     {creates and opens the output file}
    Printer.Canvas.Font := Memo1.Font;  {assigns Font settings to the canvas}
    for Line := 0 to Memo1.Lines.Count - 1 do
      Writeln(PrintText, Memo1.Lines[Line]);{writes the contents of the Memo1 to the printer
object}
    CloseFile(PrintText); {Closes the printer variable}
  end;
end;
```

Accessing and editing menus at run time

See also

While you use the Delphi Menu Designer to visually design your application menus, the underlying code is what makes the menus ultimately useful. Each menu command needs to be able to respond to an OnClick event, and there are many times when you want to change menus dynamically in response to program conditions.

You can design your own application menus, or use the predesigned menu templates included with Delphi. For information about how to design menu bars and pop-up (local) menus, see Designing Menus.

The following topics describe how to associate code with menu events at design time, and how to access and modify menus in your running application.

Associating menu events with code

Adding menu items dynamically

Merging menus

Disabling (dimming) menu items

Opening a dialog box with a menu command

See also

TPopupMenu component

Displaying a list of open documents in a menu

Associating menu events with code

See also

There is only one event for menu items: OnClick. Code that you associate with a menu item's OnClick event is executed whenever the user chooses the menu item in the running application, either by clicking the menu command or by using its accelerator or shortcut keys.

To generate an event handler for any menu item,

1. From the Menu Designer window, double-click the menu item.
2. Inside the **begin**...**end** block, type the code you want to execute when the user clicks this menu command.

To generate an event handler for a menu item displayed in the form,

- Simply click the menu item (not the menu component).

  For example, if your form contains a File menu with an Open menu item below it, you can click the Open menu item to generate or open the associated event handler.

To associate code with a menu item's OnClick event,

2. Double-click the OnClick event's Value column.
3. Inside the **begin...end** block, enter the code you want to execute when the user clicks this menu command.

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

1. In the Menu Designer window, select the menu item.
2. Display the Properties page of the Object Inspector, and ensure that a value is assigned to the menu item's Name property.
3. Display the Events page of the Object Inspector.
4. Click the down arrow next to OnClick to open a list of previously written event handlers.

   Only those event handlers written for OnClick events in this form appear in the list.
5. Select from the list by clicking an event handler name.

   The code written for this event handler is now associated with the selected menu item.

   Note: Delphi does not create a new event handler in the Code Editor for the new item; it simply calls the original code segment when the new item receives the OnClick event.

See also

Accessing and editing menus at run time
Adding menu items dynamically
Arranging and accessing open child windows
Disabling (dimming) menu items
Merging menus
Opening a dialog box with a menu command
Code Editor

Adding menu items dynamically

See also Example

You can add menu items to an existing menu structure while the application is running to provide more information or options to the user.

- Insert a menu item by using the menu item's Add or Insert method.
- Alternately hide and show the items in a menu by changing their Visible property. The Visible property determines whether the menu item is displayed in the menu.
- Dim a menu item without hiding it by using the Enabled property.

In Multiple Document Interface (MDI) and Object Linking and Embedding (OLE) applications, you can also merge menu items into an existing menu bar. See Merging Menus.

To insert a menu item into an existing menu,

1. Create an event handler for the event you want to respond to.
2. Declare the menu item as a variable of type TMenuItem.
3. Write a procedure that adds the menu item to an existing menu.

See also

Accessing and editing menus at run time
Associating menu events with code
Arranging and accessing open child windows
Disabling (dimming) menu items
Merging menus
Opening a dialog box with a menu command

Example

The following example adds a new menu item to the Window menu (WindowMenu) when Button1 is clicked.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NewItem: TMenuItem;
begin
  NewItem := TMenuItem.Create(WindowMenu); {Creates the new menu item}
  NewItem.Caption := 'My Menu Command';  {Caption for the new menu item}
  WindowMenu.Insert(1, NewItem);   {Inserts the new menu item}
end;
```

Merging menus

See also

For Multiple Document Interface (MDI) applications, the application's main menu needs to be able to receive menu items either from another form or from the OLE server object. This is often called merging menus.

You prepare menus for merging by specifying values for two properties:

- Menu, a property of the form
- GroupIndex, a property of menu items in the menu

Specifying the active menu: Menu property

The Menu property specifies the active menu for the form. Menu-merging operations apply only to the active menu. If the form contains more than one menu component, you can change the active menu at run time by setting the Menu property in code. For example,

```
Form1.Menu := SecondMenu;
```

Determining the order of merged menu items: GroupIndex property

The GroupIndex property determines the order in which the merging menu items appear in the shared menu bar. Merging menu items can replace those on the main menu bar or can be inserted.

The default value for GroupIndex is 0. Several rules apply when specifying a value for GroupIndex:

- Lower numbers appear first (farther left) in the menu.

  For instance, set the GroupIndex property to 0 (zero) for a menu that you always want to appear leftmost, such as a File menu. Similarly, specify a high number (it needn't be in sequence) for a menu that you always want to appear rightmost, such as a Help menu.
- To replace items in the main menu, give items on the child menu the same GroupIndex value.

  This can apply to groupings or to single items. For example, if your main form has an Edit menu item with a GroupIndex value of 1, you can replace it with one or more items from the child form's menu by giving them a GroupIndex value of 1 as well.

  Giving multiple items in the child menu the same GroupIndex value keeps their order intact when they merge into the main menu.
- To insert items without replacing items in the main menu, leave room in the numeric range of the main menu's items and "plug in" numbers from the child form.

  For example, number the items in the main menu 0 and 5, and insert items from the child menu by numbering them 1, 2, 3, and 4.

Additional rules apply for OLE client applications.

See also

Accessing and editing menus at run time

Using OLE

Disabling (dimming) menu items

See also Example

You can prevent users from accessing certain menu commands based on current program conditions, without removing the command from the menu. For example, if no text is currently selected in a document, the Cut, Copy, and Delete items on the Edit menu appear dimmed.

To disable a menu item,

- Set the Enabled property. for the menu item to False. (In contrast to the Visible property, the Enabled property leaves the item visible: a value of False simply dims the menu item.)

As with most properties, you can specify an initial value for Enabled using the Object Inspector. The default value for this property is True.

See also

Accessing and editing menus at run time

Example

The UpdateMenus procedure shown here sets the Enabled property for Cut, Copy, and Delete menu items based on whether any text is selected in the memo field. By contrast, it sets the Enabled property for a Paste command based on whether any text exists on the Clipboard.

You could call this procedure, for example, from the OnClick event handler for an Edit menu item, so that the state of the menu items dynamically reflects these conditions.

```
procedure TEditForm.SetEditItems(Sender: TObject);
begin
  UpdateMenus;
end;
```

UpdateMenus uses a Boolean variable, HasSelection, to reflect the value of the memo's SelLength property. If SelLength is not equal to zero, (in other words, if the memo contains selected text), HasSelection stores True; otherwise, it stores False. The rest of the procedure assigns a value to the menu items' Enabled property based on the value of HasSelection.

```
procedure TEditForm.UpdateMenus;
var
  HasSelection: Boolean;
begin
  Paste1.Enabled := Clipboard.HasFormat(CF_TEXT);
  HasSelection := Memo1.SelLength <> 0;
  Cut1.Enabled := HasSelection;
  Copy1.Enabled := HasSelection;
  Delete1.Enabled := HasSelection;
end;
```

The HasFormat method of the Clipboard returns a Boolean value based on whether the Clipboard contains objects, text, or images of a particular format. By checking the value of the Boolean expression for HasFormat of type text (CF_TEXT), the UpdateMenus procedure enables the Paste menu item if the Clipboard contains any text, and disables the Paste item if the Clipboard does not contain text.

Opening a dialog box with a menu command

See also

To open a dialog box with a menu command, you can call the Execute method of the dialog box in response to the menu item's OnClick event. For example, the following event-handler code calls the Open File common dialog box when the user selects the application's File|Open command, (assuming the command's Name property has been set to FileOpen).

```
procedure TForm1.FileOpenClick(Sender: TObject);
begin
  OpenFileDialog1.Execute;
end;
```

Of course, you still need to specify how you want your application to interact with the dialog box once it is open.

See also

Accessing and editing menus at run time
Associating menu events with code
Adding menu items dynamically
Arranging and accessing open child windows

String List editor

See also

Use the String List editor at design time to add, edit, load, and save strings into any property that has been declared as TStrings.

To open the String List editor,

1. Place a component that uses a string list on the form.
2. With that component selected, do one of the following:

- Click the ellipsis button in the Value column for any of the properties listed below.
- Double-click the word (TStrings) in the Value column for any of the properties listed below.

Load

Click Load to display the Load String List dialog box, where you can select an existing file to read into the String List editor.

Save

Click Save to write the current string list to a file. Delphi opens the Save String List dialog box, where you can specify a directory and file name.

See also

Working with string lists

Load String List Dialog Box

Use the Load String List dialog box to select a text file to load into a property.

To open the Load String List dialog box,

- In the String List editor, click Load.

Dialog box options

File Name

Enter the name of the file you want to load or wildcards to use as filters in the Files list box.

Files

Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type

Choose a filter to display the different types of files. By default the text files (*.TXT) for the current directory are displayed in the Files list box.

Directories

Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives

Select the current drive. The directory structure for the current drive appears in the Directories list box.

Save String List dialog box

Use the Save String List dialog box to store the string list from the String List editor into a text file.

To open the Save String List dialog box,

- In the String List editor, click Save.

Dialog box options

File Name

Enter the name of the file you want to save or wildcards to use as filters in the Files list box.

Files

Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type

Choose a filter to display the different types of files. By default the text files (*.TXT) for the current directory are displayed in the Files list box.

Directories

Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives

Select the current drive. The directory structure for the current drive appears in the Directories list box.

Picture editor

Use the Picture editor to select an image to add to any of the graphic-compatible components and to specify an icon for your form.

To open the Picture editor,

1. Place a graphic-compatible component on the form.
2. With that component selected, do one of the following:

- Click the ellipsis button in the Value column for any of the properties listed below.
- Double-click the Value column for any of the properties listed below.

Note: To open the Picture editor from an Image component, you can also double-click the component in the form.

Load

Click Load to display the Load Picture dialog box, where you can select an existing file to read into the Picture editor. For more information about loading images into the Picture editor, see Loading an image at design time.

Save

Click Save to display the Save Picture As dialog box, where you can specify a directory and file name in which to store the image.

Clear

Click Clear to remove the association between the current image and the selected component.

Loading an image at design time

See also

Use the Delphi Picture editor to load images onto any of several graphic-compatible components and to specify an icon to represent a form when it is minimized at run time.

Each graphic-compatible component has a property that uses the Delphi Picture editor. For a list of graphic-compatible components and their properties, see the Picture editor.

To load an image at design time,

1. Add a graphic-compatible component to your form.
2. To automatically resize the component so that it fits the graphic, set the component's AutoSize property to True before you load the graphic.
3. In the Object Inspector, select the property that uses the Picture editor.
4. Either double-click in the Value column, or choose the ellipsis button to open the Picture editor.

   (To open the Picture editor from an Image component, you can also double-click the component in the form.)
5. Choose the Load button to open the Load Picture dialog box.
6. Use the Load Picture dialog box to select the image you want to display, then choose OK.

   The image you choose is displayed in the Picture editor.
7. Choose OK to accept the image you have selected and exit the Picture Editor dialog box.

   The image appears in the component on the form.

Note: When loading an graphic into an Image component, you can automatically resize the graphic so that it fits the component by setting the Image component's Stretch property to True. (Stretch has no effect on the size of icon (.ICO) files.)

See also

AutoSize property
Specifying different glyphs for the different button states
TBitBtn component
TImage component
TOutline component
TForm component
TSpeedButton component

Load Picture dialog box

See also

Use the Load Picture dialog box to select an image to add to any of the graphic-compatible components and to specify an icon for your form.

To open the Load Picture dialog box,

- In the Picture editor, click Load.

Dialog box options

File Name

Enter the name of the file you want to load or wildcards to use as filters in the Files list box.

Files

Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type

Choose a filter to display the different types of image files. By default the icon files (*.ICO) for the current directory are displayed in the Files list box.

Directories

Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives

Select the current drive. The directory structure for the current drive appears in the Directories list box.

See also
Loading an image at design time

Save Picture As dialog box

Use the Save Picture As dialog box to store the image loaded in the Picture editor into a new file or directory.

To open the Save Picture As dialog box,

- In the Picture editor, click Save As.

Dialog box options

File Name

Enter the name of the file you want to load or wildcards to use as filters in the Files list box.

Files

Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type

Choose filter to display the different types of image files. By default the icon files (*.ICO) for the current directory are displayed in the Files list box.

Directories

Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives

Select the current drive. The directory structure for the current drive appears in the Directories list box.

Notebook editor

Use the Notebook editor to add, edit, remove, or rearrange pages in either a TabbedNotebook component or Notebook component. You can also use the Notebook editor to add or edit Help context numbers for each notebook page.

The Notebook editor displays the current pages of the notebook in their current order, and it also displays the Help context associated with that page.

To open the Notebook editor,

1. Place a Notebook component or TabbedNotebook component on the form.
2. With that component selected, do one of the following:

- Click the ellipsis button in the Value column for the Pages property.
- Double-click the Value column for the Pages property.

Edit

Click Edit to open the Edit Page dialog box, where you can modify the page name and Help context number for the selected notebook page.

Add

Click Add to open the Add Page dialog box, where you can create a new notebook page. For more information about adding pages to a notebook, see Manipulating Notebook Pages.

Delete

Click Delete to remove the selected page from the notebook.

Move Up/Move Down

Click Move Up or Move Down to rearrange the order of the selected page or pages.

Edit Page dialog box

See also

Use the Edit Page dialog box to edit existing notebook pages from either the Notebook component or the TabbedNotebook component.

To open this dialog box,

- In the Notebook editor, click Edit.

Dialog box options

Page Name

Enter the name of the notebook page. There is a 255-character limit on page names.

Help Context

Enter the context ID number for the notebook page. This number is significant if you want to have context-sensitive help for the individual pages of the notebook. The Help context is optional.

See also

Manipulating notebook pages.
Notebook editor
TNotebook component
TTabbedNotebook component

Add Page dialog box

See also

Use the Add Page dialog box to add notebook pages to either the Notebook component or the TabbedNotebook component.

To open this dialog box,

- In the Notebook editor, click Add.

Dialog box options

Page Name

Enter the name of the notebook page. There is a 255-character limit on page names.

Help Context

Enter the context ID number for the notebook page. This number is significant if you want to have context-sensitive Help for the individual pages of the notebook. The Help context is optional.

See also

Manipulating notebook pages.
Notebook editor
TNotebook component
TTabbedNotebook component

Input Mask editor

See also

Use the Input Mask editor to define an edit box that limits the user to a specific format and accepts only valid characters. For example, in a data entry field for telephone numbers you might define an edit box that accepts only numeric input. If a user then tries to enter a letter in this edit box, your application will not accept it.

Use the Input Mask editor to edit the EditMask property of the MaskEdit component.

To open the Input Mask editor,

1. Place a MaskEdit component on the form.
2. With that component selected, do one of the following:

- Click the ellipsis button in the Value column for the EditMask property.
- Double-click the Value column for the EditMask property.

Input Mask

Define your own masks for the edit box. You can use special character to specify the mask; for a listing of those characters, see the EditMask property.

The mask consists of three fields separated by semicolons. The three fields are

- The mask itself; you can use predefined masks or create your own.
- The character that determines whether or not the literal characters of the mask are saved as part of the data.
- The character used to represent a blank in the mask.

Save Literal Characters

Check to store the literal characters from the edit mask as part of the data. This option affects only the Text property of the MaskEdit component. If you save data using the EditText property, literal characters are always saved.

This check box toggles the second field in your edit mask.

Character For Blanks

Specify a character to use as a blank in the mask. Blanks in a mask are areas that require user input.

This edit box changes the third field of your edit mask.

Test Input

Use Test Input to verify your mask. This edit box displays the edit mask as it will appear on the form.

Sample Masks

Select a predefined mask to use in the MaskEdit component. When you select a mask from this list, Delphi places the predefined mask in the Input Mask edit box and displays a sample in the Test Input edit box. To display masks appropriate to your country, choose the Masks button.

Masks

Choose Masks to display the Open Mask File dialog box, where you choose a file containing the sample masks shown in the Sample Masks list box.

See also
TMaskEdit component

Masked Text editor

See also

Use the Mask Test editor to enter Values into the edit mask.

Use the Masked Text editor to edit the Text property of the MaskEdit component.

To open the Masked Text editor,

1. Place an MaskEdit component on the form.
2. With that component selected, do one of the following:

- Click the ellipsis button in the Value column for the Text property.
- Double-click the Value column for the Text property.

Input Text edit box

Enter initial values for the MaskEdit component. You can overwrite these values at run time.

Edit Mask label

Displays the mask definition for the current component.

See also

EditText property

TMaskEdit component

Font editor

See also

Use the Font editor to specify, at design time, a font and other font attributes for the selected component or form. Changes you make using the Font editor are reflected in the Font property for a component.

To open the Font editor,

1. Select any component or the form.
2. Do one of the following:

- Click the ellipsis button in the Value column for the Font property or one of the other properties listed below that use the Font editor.
- Double-click the Value column for the Font property or one of the other properties listed below that use the Font editor.

Font

Select a font from the list of all the available fonts you can use in your application.

Font Style

Select a style for the font. This combo box displays only those styles that are available for the selected font. For most of the available fonts there are four possible styles:

- Regular
- Italic
- Bold
- Bold Italic

Size

Select a size for the font. This combo box displays only those font sizes that are available for the selected font.

Effects

Check these options to make the text strikethrough or underlined.

Color

Select a color for the font. This combo box lists all the available colors for the selected font.

Sample area

Displays a sample of the selected font before you apply it to the components. The font within this area is updated with every change you make to the font settings.

See also

TFontDialog component

Color editor

Use the Color editor to specify or define a color for the selected component. Changes you make using the Color editor are reflected in the Color property for a component.

To open the Color editor,

1. Select any component or the form.
2. Double-click the Value column for the Color property or one of the other properties listed below that use the Color editor.

Basic Colors grid

Displays selection of standard colors. Click a color to apply it to the selected component.

Custom Colors grid

Displays the colors that you have created. You can create custom colors by clicking Define Custom Colors.

Define Custom Colors

Click Define Custom Colors to expand the Color editor to show options that enable you to create your own colors.

Color field

Displays the spectrum of available colors. The crosshairs indicate the current color. For example, the crosshairs look like this when the color is a shade of yellow:

Click anywhere or drag in the color field to select a color. When you select a color here and then click Add To Custom Color, the selected color is added to one of the Custom Color boxes so you can use it again.

Color|Solid

Displays the currently selected color and its closest solid color. Double-click the solid color to make it the current color.

Hue

Enter a value for the hue. Hue is the "actual" color, for example, red, yellow-green, or purple. Hue refers to the color without regard to saturation or brightness (luminosity).

Sat(uration)

Enter a value for the saturation. Saturation refers to how much gray is in the color. The Sat(uration) field shows the saturation from 0 (medium gray) to 240 (pure color).

Note: Saturation affects how clear the color is. Luminosity affects how bright the color is.

Lum(inosity) and the Luminosity Slider Control

Enter a value for the luminosity, or drag the pointer on the slide to set the luminosity. Luminosity is the brightness of a color. The Lum(inosity) field shows the luminosity from 0 (black) to 240 (white). The column to the right of the color field shows the range of luminosity for the current color. Slide the arrow to the right of the column up or down to adjust the luminosity. When you change the luminosity, the Red/Green/Blue color values also change.

Red/Green/Blue

Enter values for the proportion of red, green, and blue you want in your color. The values in these fields show the balance of red, green, and blue in the current color. This is sometimes called the RGB color. The range of available values for an RGB color is 0 to 255.

Add To Custom Colors

Click to add the color you have defined to the Custom Color grid on the Color editor.

Following section followed the first para in the *Insert Object dialog box* topic, but have jumps to ObjClass and ObjDoc property which do not appear to be in the VCL.

To open the Insert Object dialog box,

1. Place an OLEContainer component on the form.
2. With that component selected, do one of the following:

- Click the ellipsis button in the Value column for the ObjClassvclObjClassProperty@vcl.hlp or ObjDocvclObjDOCProperty@vcl.hlp property.
- Double-click the Value column for the ObjClass or ObjDoc property.

Insert Object dialog box

Use the Insert Object dialog box at design time to insert an OLE object into an OLEContainer component. The OLEContainer component enables you to create applications that can share data with an OLE server application. After you insert an OLE server object in your application, you can double-click the OLEContainer component to start the server application.

Create New/Create From File Radio

Select whether or not you want to create a new file for the chosen OLE server or use an existing file. If you use an existing file, it must be associated with an application that can be used as an OLE server.

Object Type

Select an application that you want to use as the OLE server. This list box displays all available applications that can be used as an OLE server. After you select an application, Delphi launches that application.

Note: This list box is available only when you have selected the Create New radio button.

File

Enter the fully qualified path for the file you want to insert into your application. The file you choose must be associated with an application that can be used as an OLE server.

Note: This option is available only when you have selected the Create From File radio button.

Browse

Click Browse to display the Browse dialog box, where you can select a file to use as the OLE server.

Note: This option is available only when you have selected the Create From File radio button.

Link

Check Link to link the object on the form to a file. When an object is linked, it is automatically updated whenever the source file is modified. When Link is unchecked, you are embedding the object, and changes made to the original are not reflected in your container.

Display As Icon

Check to display the inserted object as an icon on the form. When this option is checked, the Change Icon button is displayed.

Change Icon

Click Change Icon to open the Change Icon dialog box, where you can specify an icon and label for the object you inserted onto the form.

Note: This option is available only when you have selected the Create From File radio button.

Browse dialog box

The Browse dialog box has multiple uses depending on where you open it. You can use the Browse dialog box for either of the following:

- To load an existing file into the OLE container. The file you select must be associated with an application that can be used as an OLE server.
- To select an icon to represent an OLE object on the form.

To open the Browse dialog box, do one of the following:

- Click Browse in the Insert Object dialog box when you have Create From File selected.
- Click Browse in the Change Icon dialog box.

Dialog box options

Source
Enter the name of the file you want to load or wildcards to use as filters in the Files list box.

Files
Displays the files in the current directory that match the wildcards in the Source edit box or the file type in the List Files Of Type combo box.

List Files Of Type
Choose the type of file you want to use as the OLE server. By default all files in the current directory are displayed in the Files list box.

Directories
Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the Source edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives
Select the current drive. The directory structure for the current drive appears in the Directories list box.

Change Icon dialog box

Use the Change Icon dialog box to specify an icon and a label for the object you are placing on the form.

To open the Change Icon dialog box,

1. On the Insert Object dialog box, check Display As Icon.
2. Click Change Icon.

Icon Radio

Select which icon you want to use. There are three options:

Option	When selected
Current	Uses the current icon.
Default	Uses the default icon.
From File	Enables you to specify an icon using a fully qualified path name. If you do not know the icon name or the path, click Browse to open the Browse dialog box. The display box below the edit box shows all the available icons in the specified file. To choose an icon, select it.

Label

Enter a label that will appear below the icon on the form.

Browse

Click Browse to open the Browse dialog box, where you can select an icon to represent the inserted object on the form.

Sample Icon display

Displays how the icon and label will appear on the form.

Following section followed the first para in the *Paste Special dialog box* topic, but has jump to ObjItem property which does not appear to be in the VCL.

To open the Paste Special dialog box,

1. Place an OLEContainer component on the form.
2. With that component selected, do one of the following:

- Click the ellipsis button in the Value column for the ObjItemvclObjItemProperty@vcl.hlp property.
- Double-click the Value column for the ObjItem property.

Paste Special dialog box

Use the Paste Special dialog box to insert an object from the Windows Clipboard into your OLE container.

Source label

Displays the path of the file you are going to paste.

Paste/Paste Link Radio

Select Paste to embed the object on the form. When you embed an object on a form, your container application stores all the information for the object. It is not necessary to have an external file.

Select Paste Link to link the object to the form. When you link an object to a form, the main source is stored in a file so that when you update the object, the source file is also updated.

As

Lists the type of application object you are pasting. The application listed is the source application from which you received the object that you are pasting.

DDE Info dialog box

See also

Use the DDE Info dialog box to specify, at design time, a DDE server application and a topic for a DDE conversation.

To open the DDE Info dialog box,

1. Place a DDEClientConv component on the form.
2. With the component selected, do one of the following:

- Click the ellipsis button in the Value column for the DdeService property or DdeTopic property.
- Double-click the Value column for the DdeService property or DdeTopic property.

Dialog box options

Dde Service

Specify the server application for the DDE conversation. The application you specify is entered into the Value column for the DdeService property.

You do not need to specify an extension for the server application.

If the directory containing the application is not on your path, you need to specify a fully qualified path.

Dde Topic

Enter the topic for a DDE conversation. The topic is a unit of data, identifiable to the server, containing the linked text. For example, the topic could be the file name of a Quattro Pro spreadsheet.

When the server is a Delphi application, the topic is the name of the form containing the data you want to link.

If the directory containing the topic is not on your path, you need to specify a fully qualified path.

Paste Link

Click Paste Link to paste the application name and file name from the contents of the Clipboard into the App and File edit boxes.

This button is active only when the Clipboard contains data from an application that can be a DDE server.

See also

Creating DDE client applications
DDE conversations
Establishing a link with a DDE server

Filter editor

See also

Use the Filter editor to define filters for the OpenDialog component and the SaveDialog component. These common dialog boxes use the value of Filters in the List Files Of Type combo box to display certain files in the Files list box.

Use the Filter editor to edit the Filters property.

To open the Filter editor,

1. Place an OpenDialog component or SaveDialog component on the form.
2. With that component selected, do one of the following:

- Click the ellipsis button in the Value column for the Filters property.
- Double-click the Value column for the Filters property.

Filter Name column

Enter the name of the filter you want to appear in the Files Of Type combo box.

Filter column

Enter wildcards and extensions that will define your filter. For example, *.TXT would display only files with the .TXT extension.

To apply multiple file extensions to your filter, separate them using a semicolon (;).

See also

TOpenDialog component

TSaveDialog component

Select Directory dialog box (ReportSmith)

See also

Use the Select Directory dialog box to choose a directory from which you can load or store a ReportSmith report.

Use the Select Directory dialog box to edit the ReportDir property for the Report component.

To open the Select Directory dialog box,

1. Place a Report component on the form.
2. With that component selected, do one of the following:

- Click the ellipsis button in the Value column for the ReportDir property.
- Double-click the Value column for the ReportDir property.

Dialog box options

Directory Name label

Displays the fully qualified path name of the current directory.

Directories

Select a directory where you want to store or load a report. In the current directory, all the files are displayed in the Files list box.

Files

Displays all the files in the current directory selected in the Directories list box. These files are dimmed.

Drives

Select an active drive. The directory structure for the selected drive appears in the Directories list box.

Network

If you are running under Windows for Workgroups or Windows NT, you can click the Network button to open the Connect Network Drive dialog box.

See also
TReport component

Open dialog box

See also

Use the Open dialog box at design time to load a report into the Report component or a multimedia file into the MediaPlayer component.

To open the Open dialog box,

1. Place a Report component or a MediaPlayer component on the form.
2. With that component selected, do one of the following:

- Click the ellipsis button in the Value column for any of the properties listed below.
- Double-click the Value column for either of the properties listed below.

Dialog box options

File Name

Enter the name of the file you want to load or wildcards to use as filters in the Files list box.

Files

Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type

Choose the type of file you want to load. When you are loading a report, the default file extenstion is .RPT. When you are loading a multimedia file, all files in the current directory are displayed. However, you can limit the display to wave files, midi files, or Windows video files.

Directories

Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives

Select the current drive. The directory structure for the current drive appears in the Directories list box.

See also

TReport component

TMediaPlayer component

Working with text

See also

Almost all applications manipulate text in some manner, ranging from providing word-processing capabilities to the user to simply displaying text in a label or menu item that the user can't modify.

The Delphi Memo and Edit components enable the user to read and write text at run time.

Choose a topic for more information.

Providing an area for text manipulation

Setting text alignment and word wrap

Using the Clipboard with text

See also

Using the memo component
Using the edit component
Setting properties.
TDBMemo component
TDBEdit component

Providing an area for text manipulation

See also

The Memo component can be used in an application to read and display multiple lines of text, whether that text is loaded in from an existing file, pasted from the Clipboard, or entered by the user. You can set properties of the Memo component to enable word wrap, provide scroll bars, and control the border style and alignment of the memo within the form.

To provide an area where the user can manipulate text,

1. Add a Memo component to the form.
2. Set the properties shown below to ensure that the memo component remains indistinguishable from the form at run time.

Property	Value	Purpose
Align	alClient	Ensures that the memo always fills the client area, or working space, of the form.
BorderStyle	bsNone	Removes any border from within the memo component.
ScrollBars	ssBoth	Provides scroll bars when text extends beyond the borders of the form.
Text	<Blank>	Ensures that initially no text displays in the form.
WordWrap	True	Causes the text to wrap at the right margin so that it fits in the form.

See also

Setting text alignment and word wrap

Using the Clipboard with text

Setting text alignment and word wrap

See also

Alignment and WordWrap are properties of the Memo component. As with all properties, you can set their values during run time with a simple assignment statement.

The following topics discuss ways to set text alignment and word wrap at run time.

Setting text alignment

Setting word wrap

See also

Providing an area for text manipulation

Setting text alignment

See also

You set the initial text alignment at design time by setting the component's Alignment property. You can also let your users specify the type of text alignment they prefer at run time.

The following code refers to a menu item called Text that contains commands for Left, Right, and Center alignment. The code specifies that when the Left menu item receives the Click event, the text in the memo field gets aligned to the left, and the Left command gets checked in the menu.

```
procedure TForm1.AlignLeft;
begin
   MemoLeft.Checked := True;
   MemoRight.Checked := False;
   EditCenter.Checked := False;
   Memo1.Alignment := taLeft;
end;
```

Only one menu command should be checked at any time, so the previous code ensures that the other commands are unchecked.

See also

Setting word wrap

Providing an area for text manipulation

Setting word wrap

See also Example

The WordWrap property is True by default for the Memo component. The Memo can contain both vertical and horizontal scroll bars, which is the setting you might choose if word wrap were False. Word Wrap is often set as a toggle at run time.

You set the initial value of the ScrollBars property for a Memo component at design time. You might change the ScrollBars property at run time depending on the Memo.WordWrap setting. The example code illustrates one way to accomplish this.

See also

Setting text alignment

Providing an area for text manipulation

Example

The following example uses a Character menu with a Word Wrap item that the user can change dynamically to turn word wrap on and off. A check mark next to the menu item indicates that word wrap is on.

The OnClick event handler sets the value of the Memo component's WordWrap property as a toggle. Whenever the user selects the Word Wrap command, the value of the WordWrap property changes to its inverse. If WordWrap was True, it becomes False; if False, it becomes True.

The event handler adds either vertical scroll bars or both vertical and horizontal scroll bars to the Memo component based on the value of the WordWrap property.

Finally, it sets the Checked property to the value of the WordWrap property: if WordWrap is True, then the Checked property is also set to True. Checked is a Boolean property for menu items: If True, a check mark appears next to that menu item.

```
procedure TEditForm.SetWordWrap(Sender: TObject);
begin
  with Memo1 do
  begin
    WordWrap := not WordWrap;
    if WordWrap then
      ScrollBars := ssVertical else
      ScrollBars := ssBoth;
    WordWrap1.Checked := WordWrap;
  end;
end;
```

Using the Clipboard with text

See also

Most text-handling applications provide users with a way to move selected text between documents, including documents in different applications. The Clipboard object in Delphi encapsulates the Windows Clipboard and includes methods that provide the basis for operations such as cutting, copying, and pasting text (and other formats).

The Clipboard object is declared in the Delphi Clipbrd unit. Before you can access methods declared in the Clipboard object, you need to add Clipbrd to the **uses** clause of any units that will use those methods.

Choose from the following topics for more information:

Selecting text

Cutting, copying, and pasting text

Deleting text

See also

Using the Clipboard with graphics

Providing an area for text manipulation

Selecting text

See also Example

Before you can send any text to the Clipboard, the text must be selected. The function of reading and displaying selected text is native to the Memo and Edit components. In other words, you don't need to write code so that the Memo component can display selected text; it comes with this behavior.

The Stdctrls unit in Delphi provides several methods to work with selected text. (Recall that Delphi automatically adds the Stdctrls unit to the **uses** clause of any unit whose form contains a component declared within Stdctrls.) SelText, a run-time only property, contains a string based on any text selected in the component. The SelectAll method selects all the text in the memo or other component. The SelLength and SelStart properties return values for a selected string's length and starting position, respectively.

See also

Cutting, copying, and pasting text
Deleting text
Using the Clipboard with text
Providing an area for text manipulation

Example

The following code selects all text in a memo component. This could be an event handler, for example, for a Select All menu item.

```
procedure TEditForm.SelectAll(Sender: TObject);
begin
  Memo1.SelectAll;
end;
```

Cutting, copying, and pasting text

See also Example

The following methods cut, copy, and paste text:

- CutToClipboard cuts selected text from a memo or edit field and also places it on the Clipboard.
- CopyToClipboard copies all selected text in a memo or edit field to the Clipboard.
- PasteFromClipboard copies all text currently on the Clipboard back to the location of the insertion point.

See also

Selecting text
Deleting text
Using the Clipboard with text
Providing an area for text manipulation

Example

The following OnClick event handlers cut, copy, and paste selected text from a memo component to the Clipboard. These event handlers could be used on an Edit menu for the Cut, Copy, and Paste commands.

```
procedure TEditForm.CutToClipboard(Sender: TObject);
begin
  Memo1.CutToClipboard;
end;

procedure TEditForm.CopyToClipboard(Sender: TObject);
begin
  Memo1.CopyToClipboard;
end;

procedure TEditForm.PasteFromClipboard(Sender: TObject);
begin
  Memo1.PasteFromClipboard;
end;
```

Deleting text

See also Example

The ClearSelection method provides you with a way to remove selected text from a memo component without copying the selected text to the Clipboard.

Contrast this with the CutToClipboard method, which deletes selected text and also copies it to the Clipboard.

See also

Selecting text
Cutting, copying, and pasting text
Using the Clipboard with text
Providing an area for text manipulation

Example

The following event handler deletes selected text from a memo component without copying the text selection onto the Clipboard.

```
procedure TEditForm.Delete(Sender: TObject);
begin
  Memo1.ClearSelection;
end;
```

Workgroups menu: Delphi version control interface

After you have created more than one application or source code file with Delphi, you might want a way to organize and keep track of the programs. You might want to archive old versions of an application before you start to work on the new version. If you work with a team of programmers, you will need a way to coordinate which programmers have access to certain files, so that one programmer does not accidentally lose, change, or overwrite another programmer's code.

A version control system is useful when managing multiple or complex programming projects. Version control is used to archive files, control access to files by locking, and manage multiple versions of your projects. Delphi includes an interface to Intersolv PVCS Version Manager 5.1 or later.

Note: Team development support is included in Delphi Client/Server.

Enabling PVCS support

Delphi's version control system support should be enabled when you run the setup program, however, if you install the PVCS Developer Toolkit or Version Manager programs after installing Delphi, you will need to do the following:

2. After installing the PVCS product, add the following two lines to the file DELPHI.INI in your Windows directory:

   ```
   [Version Control]
   VCSManager=C:\Program Files\Borland\Delphi\Bin\STDVCS32.DLL
   ```

   Note that the path given above should specify the directory in which you installed Delphi. If necessary, replace C:\Program Files\Borland\Delphi\Bin (the default) with the path required by your system.

   Note that the PVCS runtime DLL's will need to be on your DOS path in order for Delphi to find them.

3. Run Delphi.

   The Workgroups menu group appears in the Delphi menu bar. Choose commands from the Workgroups menu to use the PVCS Version Manager. Here is a list of the commands on the Workgroups menu:

   Workgroups|Browse PVCS Project

   Workgroups|Manage Archive Directories

   Workgroups|Add Project to Version Control

Workgroups | Manage Archive Directories

See also

Choose Workgroups|Manage Archive Directores to specify the locations of archive directories.

To use the Manage Archive Directories dialog box,

1. Specify an archive directory in the Directory and Drives lists.
2. To designate the specified directory an archive location, choose Add.
3. To remove an archive location from the Archive directories list, choose Remove.
4. Choose OK to save the archive directory specification.

Workgroups | Add Project to Version Control

See also

Choose Workgroups|Add Project to Version Control to create a new PVCS project for the current Delphi Project.

Workgroups | Browse PVCS Project

See also

Choose Workgroups| Browse PVCS project to display an explorer type view of the current proejct. The project must fist be created using the Add Project to Version Control menu item.

Custom Expert menu item

You have pressed F1 on a menu item for an Expert that was installed externally. For more information regarding this item, select it to open the Expert. Help is available for externally installed Experts if provided by the developer of the Expert.

Custom Version Control menu item

You have pressed F1 on a menu item for a version control system that was installed externally. For more information regarding this item, select it to gain access to the version control system. Help may be available for your version control system depending on the particular installation.

See also

Workgroups menu
Workgroups|Browse PVCS Project
Workgroups|Add Project To Version Control

See also

Workgroups menu
Workgroups|Browse PVCS Project
Workgroups|Manage Archive Directories

See also

Workgroups menu
Workgroups|Manage Archive Directories
Workgroups|Add Project To Version Control

SQL Monitor

The SQL Monitor enables you to see the actual statement calls made through SQL Links to a remote server or through the ODBC socket to an ODBC data source.

To open the SQL Monitor,

- Choose Database|SQL Monitor.

You can elect to monitor different types of activity. Choose Options|Trace Categories to select different categories of activities to monitor. You can monitor any number of the following categories:

Category	Displays
Prepared Query Statements	Prepared statements to be sent to the server.
Executed Query Statements	Statements to be executed by the server. Note that a single statement may be prepared once and executed several times with different parameter bindings.
Statement Operations	Each operation performed such as ALLOCATE, PREPARE, EXECUTE, and FETCH
Connect / Disconnect	Operations associated with connecting and disconnecting to databases, including allocation of connection handles, freeing connection handles, if required by server.
Transactions	Transaction operations such as BEGIN, COMMIT, and ROLLBACK (ABORT).
Blob I/O	Operations on Blob datatypes, including GET BLOB HANDLE, STORE BLOB, and so on.
Miscellaneous	Operations not covered by other categories.
Vendor Errors	Error messages returned by the server. The error message may include an error code, depending on the server.
Vendor Calls	Actual API function calls to the server. For example, ORLON for Oracle, ISC_ATTACH for InterBase.

Working with string lists

See also

There are numerous occasions when a Delphi application needs to deal with lists of character strings. Among these lists are the items in a list box or combo box, the lines of text in a memo field, the list of fonts supported by the screen, the names of the tabs on a notepad, the items in an outline, and a row or column of entries in a string grid.

Although applications use these lists in various ways, Delphi provides a common interface to all of them through an object called a string list, and goes even farther by making them interchangeable, meaning you can, for example, edit a string list in a memo field and then use it as the list of items in a list box.

You have probably already used string lists through the Object Inspector. A string-list property appears in the Object Inspector with [TStrings] in the Value column. When you double-click [TStrings], you get the String List editor, where you can edit, add, or delete lines.

You can also work with string lists at run time. These are the most common types of string-list tasks:

Manipulating the strings in a list

Loading and saving string lists

Creating a new string list

Adding objects to a string list

Operating on objects in a string list

See also
Creating an owner-draw control

Manipulating strings in a list

See also

Quite often in a Delphi application, you need to write code to work with strings in an existing string list. The most common case is that some component in the application has a string-list property, and you need to change it or get strings from it.

These are the common operations you might need to perform:

Counting the strings in a list

Accessing a particular string

Finding the position of a string

Adding a string

Moving a string

Deleting a string

Copying a complete string list

Iterating the strings in a list

See also

Loading and saving string lists

Creating a new string list

Adding objects to a string list

Operating on objects in a string list

Counting the strings in a list

See also Example

To find out how many strings are in a string list, use the Count property.

Count is a read-only property that indicates the number of strings in the list. Since the indexes used in string lists are zero-based, Count is one more than the index of the last string in the list.

See also

Accessing a particular string
Finding the position of a string
Adding a string
Moving a string
Deleting a string
Copying a complete string list
Iterating the strings in a list

Example

The following example returns the number of different fonts the current screen supports.

```
FontCount := Screen.Fonts.Count;
```

Accessing a particular string

See also Example

Each string list has an indexed Strings, property which you can treat like an array of strings. For example, the first string in the list is Strings[0].

Since the Strings property is the most common part of a string list to access, Strings is the default property of the list, meaning that you can omit the Strings identifier and just treat the string list itself as an indexed array of strings.

To access a particular string in a string list,

- Refer to it by its ordinal position, or index, in the list.

The string numbers are zero-based, so if a list has three strings in it, the indexes cover the range 0..2.

To determine the maximum index,

- Check the Count property.

If you try to access a string outside the range of valid indexes, the string list raises an exception.

See also

Counting the strings in a list
Finding the position of a string
Adding a string
Moving a string
Deleting a string
Copying a complete string list
Iterating the strings in a list

Example

The following examples both set the first line of text in a memo field to be "This is the first line."

```
Memo1.Lines.Strings[0] := 'This is the first line.';

Memo1.Lines[0] := 'This is the first line.';
```

Finding the position of a string within a list

See also Example

If you have a list of strings, you can easily determine its position in a string list, or whether it is even in the list.

To locate a string in a string list,

- Use the string list's IndexOf method.

IndexOf takes a string as its parameter, and returns the index of the matching string in the list, or -1 if the string is not in the list.

Note: IndexOf works only with complete strings. That is, it must find an exact match for the whole string passed to it, and it must match a complete string in the list. If you want to match partial strings (for instance, to see if any of the strings in the list contains a given series of characters), you need to iterate the list yourself and compare the strings.

See also

Counting the strings in a list
Accessing a particular string
Adding a string
Moving a string
Deleting a string
Copying a complete string list
Iterating the strings in a list

Example

The following example uses IndexOf to determine whether a given file name is in the list of files in a file list box:

```
if FileListBox1.IndexOf('AUTOEXEC.BAT') > -1 then { you're in the root directory };
```

Adding a string to an existing list

See also Example

There are two ways to add a string to a string list:

- Add it to the end of the list
- Insert it in the middle of the list

To add a string to the end of the list,

- Call the Add method, passing the new string as the parameter. The added string becomes the last string in the list.

To insert a string into the list,

- Call the Insert method, passing two parameters: The index where you want the inserted string to appear, and the string.

If the list does not already have at least two strings, you will get an index-out-of-range exception.

See also

Counting the strings in a list
Accessing a particular string
Finding the position of a string
Moving a string
Deleting a string
Copying a complete string list
Iterating the strings in a list

Example
The following example inserts the string 'Three' as the third string in a list:

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

Note: If the list doesn't already have at least two strings, Delphi raises an index-out-of-range exception.

Moving a string within a list

See also Example

You can move a string to a different position in a string list, such as when you want to sort the list. If the string has an associated object, the object moves with the string.

To move a string in the list,

- Call the Move method, passing two parameters: the current index of the item and the index where you want to move the item to.

See also

Counting the strings in a list
Accessing a particular string
Finding the position of a string
Adding a string
Deleting a string
Copying a complete string list
Iterating the strings in a list

Example

The following example moves the third string in a list to the fifth position.

```
Move(3, 5);
```

Deleting a string from a list

See also Example

When you have an existing list of strings, you might often want to remove a string from that list.

To delete a string from a string list,

- Call the string list's Delete method, passing the index of the string you want to delete.

  If you do not know the index of the string you want to delete, use the IndexOf method to locate it.

To delete all the strings in a string list,

- Use the Clear method.

See also

Counting the strings in a list
Accessing a particular string
Finding the position of a string
Adding a string
Moving a string
Copying a complete string list
Iterating the strings in a list

Example

The following example uses IndexOf to determine the location of a string in a list, and deletes that string if present:

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

Copying a complete string list

See also Example

Copying the strings from one list to another overwrites the strings that were originally in the destination list.

To copy a list of strings from one string list to another,

- Assign the source list to the destination list.

Even if the lists are associated with different kinds of components (or no components at all), Delphi handles the copying of the list for you.

However, sometimes you want to append a new string list to an existing list.

To add a list of strings to the end of another list,

- Call the AddStrings method, passing as a parameter the list of strings you want to add.

See also

Counting the strings in a list
Accessing a particular string
Finding the position of a string
Adding a string
Moving a string
Deleting a string
Iterating the strings in a list

Examples

The following example copies the items from a combo box into an outline:

```
Outline1.Lines := ComboBox1.Items;
```

The following example adds all the items from the combo box to the end of the outline:

```
Outline1.AddStrings(ComboBox1.Items);
```

Iterating the strings in a list

See also Example

Many times you need to perform an operation on each string in a list, such as searching for a particular substring or changing the case of each string.

To iterate through each string in a list,

- Use a **for** loop with an Integer-type index. Inside the loop you can access each string and perform the desired operation.

  The loop should run from zero up to one less than the number of strings in the list (Count - 1).

See also

Counting the strings in a list
Accessing a particular string
Finding the position of a string
Adding a string
Moving a string
Deleting a string
Copying a complete string list

Example
The following example iterates the strings in a list box and converts each one to all uppercase characters in response to a button click:

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

Loading and saving string lists

See also Example

You can easily store any Delphi string list in a text file and load it back again (or load it into a different list).

You can also use the same mechanism to save lists of items for list boxes or complete outlines.

To load a string list from a file,

- Call the LoadFromFile method and pass the name of the text file to load from.

LoadFromFile reads each line from the text file into a string in the list.

To store a string list in a text file,

- Call the SaveToFile method and pass it the name of the text file to save to.

If the file does not already exist, SaveToFile creates it. Otherwise, it overwrites the current contents of the file with the strings from the string list.

See also

Manipulating the strings in a list

Creating a new string list

Adding objects to a string list

Operating on objects in a string list

Example
The following example loads a copy of the AUTOEXEC.BAT file from the root directory of the C drive into a memo field and makes a backup copy called AUTOEXEC.BAK:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  FileName: string;       { storage for file name }
begin
  FileName := 'C:\AUTOEXEC.BAT';  { set the file name }
  with Memo1 do
  begin
    LoadFromFile(FileName);       { load from file }
    SaveToFile(ChangeFileExt(FileName, 'BAK'));  { save into backup file }
  end;
end;
```

Creating a new string list

See also

Most of the time when you use a string list, you use one that is part of a component, so you do not have to construct the list yourself. However, you can also create standalone string lists that have no associated component. For instance, your application might need to keep a list of strings for a lookup table.

When you create your own string list you must remember to free the list when you finish with it. There are two distinct cases you might need to handle:

- A list that the application creates, uses, and destroys all in a single routine
- A list that the application creates, uses throughout run time, and destroys before it shuts down

The way you create and manage a string list depends on the string list type. It can be either of the following:

- Long-term string lists
- Short-term string lists

See also

Manipulating the strings in a list

Loading and saving string lists

Adding objects to a string list

Operating on objects in a string list

Short-term string lists

See also Example

Short-term string lists are useful if you need to use a string list only for the duration of a single routine. You can create it, use it, and destroy it all in one place. This is the safest way to use string list objects.

Because the string list object allocates memory for itself and its strings, it is important that you protect the allocation by using a **try..finally** block to ensure that the object frees its memory even if an exception occurs.

The basic outline of the use of a short term string list, then, is to

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

See also
Long-term string lists

Example

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

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

Long-term string lists

See also Example

Long-term string lists are useful when you need a string list that is available at any time while your application runs. You need to construct the list when the application is first executed, then destroy it before the application terminates.

To create a string list that is available throughout run time,

1. Add a field of type TStrings to the application's main form object, giving it the name you want to use.
2. Create a handler for the main form's OnCreate event. The create-event handler is executed before the form appears onscreen at run time.
3. In the create-event handler, construct the string-list object.
4. Create a handler for the main form's OnDestroy event. The destroy-event handler is executed just after the main form disappears from the screen before the application stops running.

Any other event handlers can then access the string list by the name you declared in the first step.

See also
Short-term string lists

Example
The following example adds a string to the list named ClickList. The click-event handler for the main form adds a string to the list every time the user presses a mouse button, and the application writes the list out to a file before destroying the list.

```
unit Unit1;
interface
uses WinTYpes, WinProcs, Classes, Graphics, Forms, Controls, Apps;

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

var
  Form1: TForm1;

implementation

{$R *.DFM}

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

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

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

end.
```

Adding objects to a string list

See also Example

In addition to its list of strings, stored in the Strings property, a string list can also have a list of objects, which it stores in its Objects property. Like Strings, Objects is an indexed property, but instead of an indexed list of strings, it is an indexed list of objects.

If you are just using the strings in the list, it does not matter whether you have objects: The list does nothing with the objects unless you specifically access them. Also, it does not matter what kind of object you assign to the Objects property. Delphi just holds the information; you manipulate it as you need to.

Note: Some string lists do not allow objects, because it does not make sense to have them. For example, the string lists representing the lines in a memo or the pages in a notebook cannot have associated objects.

To associate an object with an existing string,

- Assign the object to the Objects property at the same index.

Although you can assign any type of objects you want to Objects, the most common use is to associate bitmaps with strings for owner-draw controls. The important thing to remember is that strings and objects in a string list work in pairs. For every string, there is an associated object, although by default, that object is **nil**.

It is also important to understand that the string list does not own the objects associated with it. That is, destroying the string-list object does not destroy the objects associated with the strings.

See also

Creating an owner-draw control

Operating on objects in a string list

Operating on objects in a string list

See also

For every operation you can perform on a string list with a string, you can perform a corresponding operation with a string and its associated object. For example, you can access a particular object by indexing into the Objects property, just as you would the Strings property. The biggest difference is that you cannot omit the name Objects, since Strings is the default property of the string list.

The following table summarizes the properties and methods you use to perform corresponding operations on strings and objects in a string list.

Operation	For strings	For objects
Access an item	Strings property	Objects property
Add an item	Add method	AddObject method
Insert an item	Insert method	InsertObject method
Locate an item	IndexOf method	IndexOfObject method

Methods such as Delete, Clear and Move operate on items as a whole. That is, deleting an item deletes both the string and the corresponding object. Also note that the LoadFromFile and SaveToFile methods operate on only the strings, since they work with text files.

Accessing associated objects

You access objects associated with a string list just as you access the strings in the list. For example, to get the first string in a string list, you access the string list's Strings property at index 0: Strings[0]. The object corresponding to that string is Objects[0].

See also

Creating an owner-draw control

Adding objects to a string list

Example

The following example associates a bitmap called AppleBitmap with the string 'apple' in the string list named Fruits.

```
with Fruits do Objects[IndexOf('apple')] := AppleBitmap;
```

You can also add objects at the same time you add the strings, by calling the string list's AddObject method instead of Add, which just adds a string. AddObject takes two parameters, the string and the object. For example,

```
Fruits.AddObject('apple', AppleBitmap);
```

Working with code

See also

Part of designing your user interface includes deciding how you want components to respond to the events that can occur in your application. The code that specifies how a component should respond to an event is called an event handler.

You use the Delphi Object Inspector to generate the initial event handlers for components. Then you complete the event handler by writing code in the Delphi Code Editor.

The following topics explain how to use Delphi when you write application code.

Handling events

Using the Code Editor

See also

- Manipulating components
- Setting properties

Working with projects

Handling events

See also

Events represent user actions (or internal system occurrences) that your program can recognize, such as a mouse click. In Delphi, almost all the code you write will be executed in response to such an event. The code that specifies how a component should respond to an event is called an event handler.

Event handlers are really specialized procedures.Procedures and functions, also known as subroutines or sub-programs, generally constitute the bulk of your program code. Procedures and functions associated with a component are called methods.

You can use the Object Inspector to easily generate, locate or modify event handlers from within the Code Editor. You can also view events common to multiple components, and associate new events with existing event handlers.

For more information about handling events, choose a topic from the following list.

Generating a new event handler

Generating a handler for the default event

Reusing event handlers

Displaying and coding shared component events

Locating existing event handlers

Modifying a shared event handler

Deleting event handlers

See also

Using the Code Editor
Manipulating components
Setting properties

Generating a new event handler

See also Example

You can use the Object Inspector to generate an event handler for the form, or any component you have on the form. When you do so, Delphi generates and maintains parts of the code for you. The first line of the event handler names the procedure and specifies the parameters it uses. The code you write between the **begin..end** block will execute whenever the event occurs.

To use the Object Inspector to generate an event handler,

1. Select a component (or the form).
2. Click the Events tab at the bottom of the Object Inspector.

   The Events page of the Object Inspector displays all events that can be associated with the component, with the default event highlighted.
3. Select the event you want, then double-click the Value column, or press Ctrl+Enter.

   Delphi generates the event handler in the Code Editor and places the cursor inside the **begin..end** block.
4. Inside the **begin..end** block, type the code that you want to execute when the event occurs.

See also

Generating a handler for the default event
Reusing event handlers
Displaying and coding shared component events
Locating existing event handlers
Modifying a shared event handler
Deleting event handlers

Example

The following example is the initial event-handler code Delphi generates for the (default) OnClick event for Button1 on Form1:

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

end;
```

Generating a handler for the default event

See also Example

You can generate an event handler for most components simply by double-clicking the component in the form. This generates a handler for the default event of that component, if one exists. The default event is the one the component will most commonly need to handle at run time. For example, a button's default event is the OnClick event.

To generate a default event handler,

- Double-click the component in the form.

Note: Certain components, such as the Timer or the Notebook, don't need to handle user events. For these components, double-clicking them in the form does nothing. Double-clicking certain other components, like the Image component or the MainMenu component, opens a dialog box where you perform design-time property edits.

See also

Generating a new event handler
Reusing event handlers
Displaying and coding shared component events
Locating existing event handlers
Modifying a shared event handler
Deleting event handlers

Viewing pages in the Code Editor

See also

When a page of the Code Editor is displayed, you can scroll through all the data it contains, not just particular sections of your code.

To view a page in the Code Editor, choose one of the following methods:

- Click the tab for the page you want to view.
- Press Ctrl+Tab to go forward through the Editor pages, and Shift+Ctrl+Tab to go backward.
- Select a unit from the View Unit dialog box. To open the View Unit dialog box, choose View|Unit.

When the Code Editor is displayed, you can return to a form at any time using either of these methods:

To return to the form,

- Click any part of the form that is visible under the Code Editor.
- Choose View|Form to open the View Form dialog box, and choose the form you want to view.
- Use SpeedBar buttons to display the current form, or to open the View Form dialog box.

See also

Code Editor
Locating existing event handlers
Deleting event handlers

Locating existing event handlers

See also

If you want to view the code for an event handler that you have previously written, you can quickly locate it in the Code Editor using the Object Inspector.

To locating existing event handlers in the Code Editor,

1. In the form, select the component whose event handler(s) you want to locate.
2. In the Object Inspector, display the Events page.
3. In the Events column, double-click the event whose code you want to view.

Delphi displays the Code Editor, placing the cursor inside the **begin..end** block of the event handler. You can now modify this event handler.

You can also use the Search menu to locate specific text in the Code Editor.

Note: If you generated a default event handler for a component by double-clicking it in the form, you can locate that event handler in like fashion.

To locate an existing default handler,

- Double-click the component in the form.

See also

Generating a new event handler
Generating a handler for the default event
Reusing event handlers
Displaying and coding shared component events

Modifying a shared event handler

See also

Modifying an event handler that is shared by more than one component is virtually the same as modifying any existing event handler. Just remember that whenever you modify a shared event handler, you are modifying it for all the component events that call it.

To modify the shared event handler,

1. Select any of the components whose event calls the handler you want to modify.
2. In the Events page of the Object Inspector, double-click the event handler name.
3. In the Code Editor, modify the handler.

   Now when any of the components that share this handler receive the shared event, the modified code gets called.

See also

Generating a new event handler
Generating a handler for the default event
Locating existing event handlers
Displaying and coding shared component events
Deleting event handlers

Deleting event handlers

See also

When you delete a component, Delphi removes its reference in the form's type declaration. However, deleting a component does not delete any associated methods from the unit file, because those methods might be called by other components in the form. You can still run your application so long as the method declaration and the method itself both remain in the unit file. If you delete the method without deleting its declaration, Delphi generates an "Undefined forward" error message, indicating that you need to either replace the method itself (if you intend to use it), or delete its declaration as well as the method code (if you do not intend to use it).

You can still explicitly delete an event handler, if you choose, or you can have Delphi do it for you.

To manually delete an event handler,

- Remove all event-handler code and the handler's declaration.

To have Delphi delete an event handler,

- Remove all the code (including comments) inside the event handler. The handler is removed when you compile or save the project.

See also

Generating a new event handler
Generating a handler for the default event
Locating existing event handlers
Displaying and coding shared component events
Modifying a shared event handler

Reusing event handlers

See also

As you build your applications, you might find you often want the same result from different events. For example, if your application contains a SpeedBar with buttons for the Cut, Copy, and Paste commands, you might want to program each of these buttons to execute the same event handler as its corresponding Edit menu command.

Once you write one such event handler, you can easily associate other events with the original event handler.

You can also have several different components share a handler that does different things depending on which component called it. To determine which component received the event, check the Sender parameter. For more information, see Using the Sender parameter.

To associate a component event with an existing event handler,

1. In the form, select the component whose event you wish to code.
2. Display the Events page of the Object Inspector, and select the event to which you wish to attach handler code.
3. Click the down arrow next to the event to open a list of previously written event handlers.

   The list shows only the event handlers in this form that can be assigned to the selected event.
4. Select from the list by clicking an event-handler name.

   The code written for this event handler is now associated with the selected component event.

Note: Delphi does not duplicate the event handler code for every component event associated with a shared handler. You see the code for shared handlers in the Code Editor only once, even though the same code is called whenever the associated component events occurs.

See also

Generating a new event handler
Generating a handler for the default event
Locating existing event handlers
Displaying and coding shared component events
Modifying a shared event handler
Deleting event handlers

Using the Sender parameter

See also Example

The Sender parameter in an event handler informs Delphi which component received the event, and therefore called the handler. You can write a single event handler that responds to multiple component events by using the Sender parameter in an **if..then..else** statement.

See also

Generating a new event handler
Reusing event handlers
Displaying and coding shared component events

Example

The following code either displays the title of the application in the caption of a modal dialog box, or else displays nothing, depending on whether the OnClick event was received by Button1 or by another button using the same event handler.

```
procedure TMain1.Button1Click(Sender: TObject);
begin
if Sender = Button1 then
  AboutBox.Caption := 'About ' + Application.Title
else AboutBox.Caption := '';
AboutBox.ShowModal;
end;
```

Displaying and coding shared events

See also

There are many events, such as the OnClick event, that are available to more than one component. When components have events in common, you can associate the common event with an event handler (existing or new) without having to do this separately for each component.

You do this by first displaying the shared event, and then creating an event handler for it.

To display shared events,

1. In the form, select all the components whose common events you want to view.
2. Display the Events page of the Object Inspector.

   The Object Inspector displays only those events that pertain to all the selected components. (Note also that only events in the current form are displayed.)

When you create a shared event handler (or when you reuse an existing one), Delphi does not duplicate the event handler code for every component event associated with it. You will see the code in the Code Editor only once, but the same code gets called whenever any of the component events occurs.

To associate a shared component event with an existing event handler,

1. Select the components for which you want to associate a shared event handler.
2. Display the Events page of the Object Inspector, and select an event.

   The Object Inspector displays only those events that the selected components have in common.
3. From the drop-down list next to the event, select an existing event handler, and press Enter.

   Whenever any of the components you selected receives the specified event, the event handler you selected in called.

To create an event handler for a shared event,

1. Select the components for which you want to create a shared event handler.
2. Display the Events page of the Object Inspector, and select an event.
3. Type a name for the new handler, and press Enter, or double-click the Handler column if you want Delphi to generate a name.

   Delphi creates an event handler in the Code Editor, positioning the cursor in the **begin..end** block.

   If you choose not to name the event handler, Delphi names it for you based on the order in which you selected the components.
4. Type the code you want executed when the selected event occurs for any of the components.

See also

Generating a new event handler
Generating a handler for the default event
Reusing event handlers
Locating existing event handlers
Modifying a shared event handler
Deleting event handlers

Modifying the form's type declaration

See also

When you add a component to a form, Delphi generates an instance variable, or field, for the component and adds it to the form's type declaration. Here is how adding a button changes the form's type declaration:

```
type
 TForm1 = class(TForm)
     Button1: TButton;     {this is the code Delphi adds}
   end;
```

Similarly, when you delete a component, Delphi removes the corresponding type declaration. You can view similar code being added or removed from the Code Editor.

To view code being added in the Code Editor,

1. Click the form's Title bar and hold down the mouse button while you drag the form to a new location so you can see the entire Code Editor.
2. Scroll in the Code Editor until the type declaration section is visible.
3. Add a component to the form while watching what happens in the Code Editor.
4. Delete the component, again while viewing the Code Editor.

Note: Delphi does not remove any event handlers (or methods) associated with components you delete, because those event handlers might be called by other components in the form. You can still run your program so long as the method declaration and the method itself both remain in the unit file. If you delete the method without deleting its declaration, Delphi generates an “Undefined forward” error message.

See also

The Name property

Adding components to a form

Manipulating components

See also

You use components to create your application's user interface and to associate the code you write with application events. The topics below describe basic techniques for manipulating components in your form while in design mode.

Adding components to a form

Selecting components

Resizing, moving, and deleting components

Cutting, copying, and pasting components

Aligning components

Grouping components

See also

Component palette overview
Designing a user interface
Setting properties
Working with code
Modifying the form's type declaration

Adding components to the form

See also

Use the Component palette to add components to your form. Once components are on the form, you can move, resize, and edit them to suit the particular needs of your application.

To add a component to the form,

1. Select a component on the Component palette.
2. Do one of the following:

- To add a component to the center of the form, double-click the component in the Component palette. If there is already a component in that position, additional components are placed on top of the existing one.
- To add a component to a specific location on the form, move the cursor to where you want the upper left corner of the component to appear in the form, then click the form. The component appears in its default size, in the position you clicked on the form.
- To resize a component as you add it, place the cursor where you want the component to appear in the form, then drag the mouse pointer before releasing the mouse button. As you drag, an outline appears to indicate the size and position of the control. Release the mouse button when the outline appears as you want it.

To add multiple components of the same type to a form,

1. Press and hold the Shift key.
2. Click the component on the palette, then click on the form once for each copy of the component you want. (You don't need to hold down the Shift key after the component is selected.)
3. Click the pointer icon to clear the selected component.

Note: Before you associate code with a component, you should change the component's Name property for easy identification when reviewing your code.

See also

Form keyboard shortcuts
Modifying the form's type declaration
Manipulating components
The Name property

The Name property

See also

Every component in a given form, and every form in a project, must have a unique name. The Name property identifies the component to the underlying program and is the name reflected in, and used by, your code.

By contrast, the Caption property just displays information to the user -- for example, a title on the form's title bar, or the label on a pushbutton -- and has no other meaning to the underlying program. Changing the Caption property does not change the internal name of the form, but can still be valuable for adding user-friendliness to your application.

Delphi assigns default names to the forms and components in your project and numbers them sequentially as you add them. For example, if you add three pushbuttons to the form, Delphi names them Button1, Button2, and Button3, respectively. While these names are fine for identifying the component to the compiler, they don't make your code particularly easy to read.

It is good practice to change the Name property so that it is descriptive of the component's function.

To change the name of a component,

1. Select the component.
2. In the Object Inspector, select the Name property and enter a new name.
   Component names must follow the standard rules for naming Pascal identifiers.

Note: When you use the Object Inspector to change the Name property, Delphi renames the component in your source code only on those lines that Delphi generated. Note that this is not the case if you edit the component name directly in the Code Editor.

Changing the Name property also changes the Caption property, unless you have already changed the Caption property. The value of the Caption property has precedence over the Name property.

See also

Caption property

Setting properties

Selecting components

See also

There are several ways to select components in the form; most, but not all, of these methods also apply to components contained by other components, such as the Panel or GroupBox component.

To select a single component, do one of the following,

- Click the component in the form.
- Choose from the Object selector, located at the top of the Object Inspector.
- With focus in the form, tab to the component you want to select.

To select multiple components, do one of the following:

- First hold down the Shift key, and then click the components, one at a time.
- Click the form outside one of the components and drag over the other components you want to select. (If the components are inside a Panel or GroupBox component, press Ctrl, then drag.)

  As you drag, you surround the components with a dotted rectangle. When this rectangle encloses all the components, release the mouse button. In some cases, you might need to rearrange the components before they can be easily enclosed within a rectangle.

When a component is selected on a form, small squares called sizing handles appear on the perimeter of the component. You can use these to resize the component or components.

Note: Multiple selection applies to all components in the form, including container components and any components they contain, but when the form itself is selected, no other components can be selected along with it.

To select all components in a form,

- Choose Edit|Select All.

See also

Manipulating components
Adding components to a form
Selecting components
Resizing, moving, and deleting components
Cutting, copying, and pasting components
Aligning components
Grouping components

Resizing, moving, and deleting components

See also

When a component is selected on a form, small squares called sizing handles appear on the perimeter of the component. You can use these to resize the component or components.

During design time, you can move, resize, or delete any component on a form, to fine tune your application's user interface.

To resize a single component,

1. Select the component. Sizing handles appear on the perimeter of the component.
2. Drag a sizing handle until you are satisfied with the component's size.
3. Release the mouse button. The component is redrawn in the new size.

To resize multiple components,

1. Select the components you want to resize.
2. Choose Edit|Size to display the Size dialog box.
3. Select appropriate sizing options, then choose OK.

To move a component,

1. Select the component.
2. Drag the component to its new location. Be careful not to drag the sizing handles or you will resize the component instead of moving it.

To delete a component,

1. Select the component.
2. Press the Delete key, or choose Edit|Delete.

To restore a component that was just deleted,

- Choose Edit|Undelete.

  This restores the component just as it was, including any customizations you might have made to it such as setting properties or writing code.

Note: You must choose Undelete before performing any other operation, or the component is not restored.

See also

Adding components to a form
Aligning components
Grouping components
Manipulating components
Cutting, copying, and pasting components
Selecting components

Cutting, copying, and pasting components

See also

You can cut, copy, and paste components between forms, or between a form and a container component such as a panel or group box. The container component can be on the original form or in another form.

Note: When you copy components, you copy their property settings and event-handler associations as well. For example, copying a button with a Caption property of Button1 and an event handler called TForm1.Button1Click creates another button with the same Caption property and an OnClick event handler that calls the code in TForm1.Button1Click. However, Delphi renames the copied button so that each button maintains a unique identifier.

Four general rules apply to all cut, copy, and paste operations:

- You can select multiple components to be cut or copied, but the selection can include only those components within a single parent. That parent can be either a form or a container component within the form, but not both.
- If you individually select components that have different parents and then attempt to cut or copy them, only the components in the parent of the component you first selected are cut or copied. For example, if you first select a button in the form, then select a check box in a panel on the form, and then attempt to cut or copy the two selected components, only the button is cut or copied.
- When you paste one or more components, they appear in whichever eligible receiving component has focus.
- When you cut, copy, or paste a component, any associated event handler code is not duplicated, but the copied component retains associations to the original event handlers.

To cut a component or components,

1. Select the component(s).
2. Choose Edit|Cut.

To copy a component or components,

1. Select the component(s).
2. Choose Edit|Copy.

To paste a component or components,

1. Select the form or another container component where you want the pasted components to appear.
2. Choose Edit|Paste.

(This procedure assumes you have first placed components on the Clipboard by cutting or copying.)

See also

Adding components to a form
Aligning components
Grouping components
Manipulating components
Resizing, moving and deleting components
Selecting components

Grouping components

See also

Besides the form itself, Delphi provides several components--GroupBox, Panel, Notebook, TabbedNotebook, and Scrollbox --that can contain other components. These are often referred to as container components. You can use these container components to group other components so that they behave as a unit at design time. For instance, you might group components, such as speed buttons and check boxes, that provide related options to the user.

When you place components within container components, you create a new parent-child relationship between the container and the components it contains. Design-time operations you perform on container (or parent) components, such as moving, copying, or deleting, also affect any components grouped within them.

Note: The form remains the owner for all components, regardless of whether they are parented within another component. For more information about the difference between a component's parent and owner properties, see Owner property and Parent property.

The Panel component includes alignment, bevel, and hint properties that make it easy to create customized tool bars, backdrops, status lines, and so on.

You'll generally want to add container components to the form before you add the components you intend to group, as it is easiest to add components that you want grouped directly from the Component palette into the container component.

Once a component is in the form, you can add it to a container component by cutting and then pasting it.

To group components,

1. Add a GroupBox or Panel component to the form.
2. Add components as you normally would, making sure that the container component is selected.

   When you double-click a component in the palette, it appears in the middle of whichever eligible receiving component has focus -- either the form, or a container component in the form.

   When you select and then place components, the container component need not be selected, so long as you click within it when you place the component.

To add multiple copies of a component to a container,

1. In the form, select the container component.
2. Press Shift and then select a component from the palette.
3. Click anywhere in the container component.

   Each subsequent click continues to place the component in whatever eligible receiving component (including the form) is clicked.
4. Select the pointer icon when you have finished adding components.

See also

Adding components to a form
Aligning components
Manipulating components
Resizing, moving and deleting components
Cutting, copying, and pasting components
Selecting components
TGroupBox
TPanel

Using the grid to align components

See also

The form grid makes it easier to place components exactly where you want them by giving you a visual guide.

By default, the grid is visible while you are in design mode, but it is possible to hide it. When the grid is enabled, you can choose to have components snap to the grid, which causes the left and top sides of the component to align with the nearest grid markings. You can also modify the granularity of the grid (how far apart the grid dots appear).

To view the grid,

1. Choose Options|Environment to display the Environment Options dialog box.
2. Click the Preferences tab to display the Preference page if it is not already visible.
3. In the Form Designer options, check the Display Grid to view the grid, or uncheck it to hide it.

To enable the Snap To Grid option,

1 Choose Options|Environment to display the Environment Options dialog box.

2. In the Form Designer options, check the Snap To Grid to enable the option, or uncheck it to disable it.

To modify the granularity of the grid,

1. Choose Options|Environment to display the Environment Options dialog box.
2. In the Form Designer options, enter new values in the Grid Size boxes.

   Grid Size X controls the horizontal spacing, and Grid Size Y controls the vertical spacing. The default values are 8 and 8. To space the dots further apart, choose larger numbers. To create a finer grid, choose smaller numbers.

See also

Alignment palette
Aligning components
Adding components to a form
Resizing, moving, and deleting components
Cutting, copying, and pasting Components

Aligning components

See also

You can use the Alignment palette or the Alignment dialog box to align components in relation to each other or to the form. When aligning a group of components, the first component you select is used as a guide to which the other components are aligned.

You can also use the form grid when aligning components.

To align components using the Alignment palette,

1. Select the components you want to align.
2. Choose View|Alignment Palette to open the Alignment palette.
3. Select an alignment icon from the palette.

To align components using the Alignment dialog box,

1. Select the components you want to align.
2. Choose Edit|Align to open the Alignment dialog box.
3. Select an alignment option from the dialog box.
4. Choose OK to accept the alignment options.

You can continue to choose or modify alignment options as long as the components remain selected.

See also

Adding components to a form
Grouping components
Manipulating components
Resizing, moving, and deleting components
Cutting, copying, and pasting components
Selecting components
Using the grid to align components

Setting properties

See also

Properties are attributes that define how a component appears and responds. In Delphi, you set a component's initial properties during design time, and your code can change those properties during run time.

You use the Object Inspector to set component properties at design time. Run-time only properties don't appear in the Object Inspector.

Choose from the following topics for more information.

Setting properties at design time

Setting properties at run time

The Name property

See also

Manipulating components
Working with code
The Name property
About property editors
How the Object Inspector displays properties
Setting shared properties
Viewing nested properties

Setting properties at design time

See also

When you use the Object Inspector to set component properties at design time, the results are almost always visible without having to run your program first.

To set a component property at design time,

1. Select the component whose property you want to change.

   The Object selector displays the name of the selected component.
2. Click the Properties tab on the Object Inspector.
3. Select the property you want to change.
4. Modify the property by using the available property editors, or by typing in a new value for the property.

The following topics describe techniques for editing properties at design time in more detail.

How the Object Inspector displays properties

About property editors

Viewing nested properties

Setting shared properties

See also

Manipulating components
Setting properties at run time
Working with code

Setting properties at run time

See also

Any property you can set at design-time can also be set during run time by using code. When you use the Object Inspector to set a component property at design time, you specify the name of the component (by selecting it in the form or with the Object selector), the name of the property (by selecting it from the Properties page), and a new value for that property.

You do essentially the same thing to set properties at run time: In your source code, specify the component, the property, and the new value, in that order. For example:

```
Form1.Color := clAqua
```

Form1 is the component; Color is the property; and clAqua is the new value. (':=' is a Pascal assignment statement. For more information about using the Object Pascal language, see the Language Reference or Language Definition.)

This code assigns a new property value, but you also need to tell your application when to execute the code. You do this by associating the code with an event. For more information on events and event handlers, see Handling events.

See also

Manipulating components
Setting properties at design time
Working with code

How the Object Inspector displays properties

See also

The Object Inspector dynamically changes the set of properties it displays, based on the component selected. The Object Inspector has several other behaviors that make it easier to set component properties at design time.

- When you use the Object Inspector to select a property, the property remains selected in the Object Inspector while you add or switch focus to other components in the form, provided that those components also share the same property. This enables you to type a new value for the property without always having to reselect the property.

  If a component does not share the selected property, Delphi selects its Caption property. If the component does not have a Caption property, Delphi selects its Name property.
- When more than one component is selected in the form, the Object Inspector displays all properties that are shared among the selected components. This is true even when the value for the shared property differs among the selected components. In this case, the property value displayed will be either the default or the value of the first component selected. When you change any of the shared properties in the Object Inspector, the property value changes to the new value in all the selected components.

Note: When you select multiple components in a form, their Name property no longer appears in the Object Inspector, even though they all have a Name property. This is because you cannot assign the same value for the Name property to more than one component in a form.

See also

Setting properties
Manipulating components
Working with code
The Name property
About property editors
How the Object Inspector displays properties
Setting shared properties
Viewing nested properties

Setting shared properties

See also

Most components share common properties, such as their height (the Height property), or visibility (the Visible property). You can set the common properties of several components on a form without having to individually select and modify each component.

To set the properties of multiple components,

1. Select the components whose shared properties you want to set.

   The Properties page of the Object Inspector displays only those properties that the selected components have in common. (Similarly, values for these common properties appear only when those values are the same for all the selected components.)

2. With the components still selected, use the Object Inspector to set the property.

   Changes that are visible at design time will be reflected in each selected component, however, some changes are not reflected in your form until you run the program.

Note: When you select a component property in the Object Inspector, it remains selected while you add other components to the form, provided that those components also share the property. This enables you to type a new value for the property without always first needing to select the property. (If the component you add does not share the selected property, Delphi selects its default property.)

See also

Manipulating components
Selecting components
Setting properties
How the Object Inspector displays properties
The Name property
About property editors
Viewing nested properties
Working with code

Viewing nested properties

See also

Properties can have properties of their own, called nested properties. For example, the Font property of the Label component has nested properties, one of which is Style; the Style property in turn has its own nested properties.

Properties with nested properties show a plus (+) sign on their left side in the Object Inspector. You need to view these nested properties to set them.

To view nested properties,

Choose one of the following methods:

- Double-click any property with a plus sign next to it.

The plus sign next to the top-level property changes to a minus (-) sign, and the nested properties are displayed.

To hide nested properties,

- Double-click a property with a minus sign next to it.

See also

Manipulating components
Object Inspector overview
Setting properties
Setting shared properties
How the Object Inspector displays properties
The Name property
About property editors
Working with code

Creating dialog boxes

See also

Dialog boxes are one of the most common ways for users to interact with a Windows-based application. Use dialog boxes to obtain detailed input from and display information to the user. A dialog box can be either modal or modeless.

In addition to routines that display simple, predesigned dialog boxes and Form Template dialog boxes such as the Password Dialog template, Delphi provides many fully designed dialog boxes as components. These appear on the Dialogs page of the Component palette, and include components for many of the Windows common dialog boxes, as well as standard Find and Replace dialog boxes. You can specify different options for these dialog boxes, such as whether a Help button appears in the dialog box, but for the most part you cannot change their overall design.

It is useful, therefore, to know how to build a dialog box "from scratch" if you want to design your own.

Choose from the following list to learn about the most common considerations when designing any dialog box, including:

- Making a dialog box modal or modeless
- Setting form properties for a dialog box
- Providing command buttons
- Setting the tab order
- Enabling and disabling components
- Setting the focus in a dialog box

See also

Component basics

Dialogs page components

Making a dialog box modal or modeless

See also

Because dialog boxes are simply customized forms, they, like forms, can be either modal or modeless. Most dialog boxes are modal. When a form is modal, the user must explicitly close it before working in another form. When a form is modeless, it can remain onscreen while the user works in another form.

Any form you create can be used in your application modally, or modelessly.

To display a form in a modeless state,

- Call its Show method.

Note If you want a modeless dialog box to remain on top of other windows at run time, set its FormStyle property to fsStayOnTop.

To display a form modally,

- Call its ShowModal method.

See also

Setting properties
Building dialog boxes
Executing button code on Esc
Executing button code on Enter

Setting the tab order

See also

Tab order is the order in which focus moves from component to component in a running application when the Tab key is pressed.

To enable the Tab key to shift focus to a component on a form,

- Set the TabStop property of the component to True.

The tab order is initially set by Delphi, corresponding to the order in which you add components to the form. You can change this by changing the TabOrder property of each component, or by using the Edit Tab Order dialog box.

To use the Edit Tab Order dialog box,

1. Select the form, or a container component in the form, that contains the components whose tab order you want to set.
2. Choose Edit|Tab Order.

   The Edit Tab Order dialog box appears, displaying a list of components ordered (first to last) in their current Tab order.
3. In the Controls list, select a component and press the up or down arrow, or drag the component to its new location in the tab order list.
4. When the components are ordered to your satisfaction, choose OK.

Using the Edit Tab Order dialog box changes the value of the components' TabOrder property. You can also do this manually, if you want.

To remove a component from the tab order,

- Set the component's TabStop property to False.

When the user presses the Tab key in the running application, the focus will skip over this component and go to the next one in the tab order. This is true even if the component has a valid TabOrder value.

Note Removing a component from the tab order does not disable the component.

To manually change a component's TabOrder property,

1. Select the component whose position in the tab order you want to change.
2. In the Object Inspector, select the TabOrder property.
3. Change the TabOrder property's value to reflect the position you want the component to have in the tab order.

Note: The first component in the tab order should have the TabOrder value of 0.

Keep in mind the following points when manually setting your tab order (if you are using the Edit Tab Order dialog box, you do not need to worry about them):

- Each TabOrder property value must be unique. If you give a component a TabOrder value that has already been assigned to another component on this form, Delphi renumbers the TabOrder value for all other components accordingly.
- If you attempt to give a component a TabOrder value equal to or greater than the number of components on the form (because numbering starts with 0), Delphi does not accept the new value, instead entering a value that ensures the component will be last in the tab order.
- Components that are invisible or disabled are not recognized in the tab order, even if they have a valid TabOrder value.

When the user presses Tab, the focus skips over such components and goes to the next one in the tab order. For more information, see Enabling and disabling components.

Testing the Tab Order

You can test the tab order by running the application. At design time, focus always moves from component to component in the order that the components were placed on the form. Changes you make to the tab order at design time are reflected only at run time.

See also

Setting the focus in a dialog box
Enabling and disabling components
Building dialog boxes

Enabling and disabling components

See also Example

You often want to prevent a user from accessing certain components in a dialog box or form, either initially when the dialog box opens, or in response to changing conditions with the dialog box at run time.

To disable a component at design time,

- Use the Object Inspector to set the value of the Enabled property to False.

When a component is disabled, it appears dimmed, and the user cannot tab to it, even if its TabStop property is set to True.

Note: Certain components also contain a ReadOnly property to restrict the kind of access a user has to the contents of the component at run time.

By disabling a component at design time, you specify that the component is initially unavailable to the user when the dialog box first opens. You can also dynamically change whether a component is enabled at run time.

To disable a component at run time,

- Type the following code in an event handler for the component:

```
<componentn>.Enabled := False;
```

where <componentn> is the name of the component, for example, Button1.

Example

The following event handler specifies that when the user clicks Button1, Button2 is disabled.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Button2.Enabled := False
end;
```

See also

Setting the tab order

Creating dialog boxes

Setting the component focus in a form

See also

Only one component per form can be active, or have the focus, in a running application at any given time. The button with focus in a form takes the OnClick event when the Enter key is pressed.

The component having initial focus in the form at run time corresponds to the ActiveControl property of the form.

If no component is specified as the active control, Delphi gives initial focus to the component whose Default property is set to True.

If no component is specified as the default for the form, Delphi gives initial focus to the component that is first in the Tab Order, excluding:

- Disabled components
- Components that are invisible at run time
- Components whose TabStop property is set to False

To specify the active component at design time,

- Select the form's ActiveControl property and use the drop-down list to select the the component you want to have focus when the form first opens.

To change the active component during run time,

- Call the SetFocus method from an event handler, for example,

```
<componentn>.SetFocus;
```

where <componentn> is the name of the component, for example, Button1.

Note: If you set a button as the active component for the form at design time, that setting overrides, at run time, any default button you might have specified.

See also

Setting the tab order

Creating dialog boxes

Providing command buttons

See also

Depending on whether you intend to use your dialog box in a modal or modeless state, you might want to provide certain command buttons in the dialog box. For modal dialog boxes, you need to provide the user with a way to exit the dialog box. It's fairly standard design to provide one or more command buttons for this purpose. For simple modal dialog boxes, such as a message box, one button is often sufficient. Such a button might be labeled, for instance, "OK" or "Close." (If you have another button in the dialog box labeled "No," as described in the next paragraph, this button might be labeled "Yes.")

In cases where the dialog box accepts input from the user, you want to provide users with a choice of whether or not to process their input on exiting the dialog box. You can do this by means of an additional button labeled, for example, "Cancel" or "No." (If your dialog box explicitly asks a question of the user, you might want to label this button "No"; otherwise, "Cancel" is usually more appropriate.)

Your code controls what happens when a user chooses a command button, for example, whether changes are processed or not.

By setting properties of the Button component, you can call a button's event-handler code when the user presses Enter or Esc; and you can specify that the dialog box close when the user chooses a command button, without writing any additional code. See the following topics for more information:

Executing button code on Esc

Executing button code on Enter

Note: You can quickly create many standard command buttons by adding a BitBtn component to the form and setting its Kind property.

See also

Setting the tab order
Setting the focus in a dialog box
Building dialog boxes

Executing button code on Esc

See also

Delphi provides a Cancel property for Button components. When your form contains a button whose Cancel property is set to True, pressing the Esc key at run time executes any code contained in the button's OnClick event handler.

To designate a button as the Cancel button,

- Set its Cancel property to True.
- To specify that the modal dialog box close when the user chooses a Cancel button, set the button's ModalResult property to mrCancel.

Setting a button's ModalResult property to a nonzero value means the modal dialog box closes automatically when the user chooses the button.

You can also use the BitBtn component to create a Cancel button.

To use the bitmap button to create a Cancel button,

- Add a BitBtn component to your form, and set its Kind property to bkCancel. This sets the button's Cancel property to True, and the ModalResult property to mrCancel.

See also
Executing button code on Enter

Executing button code on Enter

See also

When your form contains a button whose Default property is set to True, pressing Enter at run time executes any code contained in the button's OnClick event handler--unless another button has focus when the Enter key is pressed.

Even if your form contains a default button, another button can take focus away at run time. Pressing the Enter key calls the OnClick event handler code of the button with focus, overriding any other button's Default property setting. (The button with focus is indicated by a darker, thicker border than that of other buttons in the dialog box.)

Note: Although other components in a form can have focus, only button components respond when the user presses Enter. The default button takes the OnClick event when another non-button component in the form has focus.

To specify a button as the default button,

- Set its Default property to True.
- To specify that the modal dialog box close when the user chooses a default button, set the button's ModalResult property to mrOK.

Setting a button's ModalResult property to a nonzero value means the modal dialog box closes automatically when the user chooses the button.

You can also use the BitBtn component to create a Default button.

To use the bitmap button to create a default button,

- Add a BitBtn component to your form, and set its Kind property to bkOK. This automatically sets the button's Default property to True and the ModalResult property to mrOK.

To change focus at run time,

- Call the button's SetFocus method.

See also
Executing button code on Esc

Setting form properties for a dialog box

See also

By default, Delphi forms have Maximize and Minimize buttons, a resizable border, and a Control menu that provides additional commands to resize the form. While these features are useful at run time for modeless forms, modal dialog boxes seldom need them.

Delphi provides a BorderStyle property for the form that includes several useful values. Setting the form's BorderStyle to bsDialog implements the most common settings for a dialog box, such as:

- Removing the Minimize and Maximize buttons
- Providing a Control menu with only the Move and Close options
- Making the form border non-resizeable, and giving it a "beveled" appearance

The following table shows other form property settings that can be used, individually or in concert, to create different form styles.

Property	Setting	Effect
BorderIcons		
biSystemMenu	False	Removes Control (System) menu
biMinimize	False	Removes Minimize button
biMazimize	False	Removes Maximize button
BorderStyle	bsSizeable	Enables the user to resize the form border
	bsSingle	Provides a single outline, non-resizeable border
	bsNone	No distinguishable border; not resizeable

Note: Changing these settings does not change the design-time appearance of the form; these property settings become visible at run time.

When you remove the form's Control (or system) menu, you need to provide the user with a way to exit the dialog box. You can do this by including buttons in the form.

Specifying a caption for a dialog box

In most Windows-based applications, each dialog box has a caption on its Title bar that describes the primary function of the dialog box.

By default, Delphi displays the Name property value for each form in the form's Title bar. If you change the Name property of the form prior to changing the Caption property, the Title bar caption changes to the new name. Once you change the Caption property, the form's title bar always reflects the current value of Caption.

See also
Creating dialog boxes

Testing the user interface

See also

After you have spent some time designing forms and coding event handlers, you might want to test your user interface to see whether it responds as you want. For instance, you can check that the proper component has focus as the application begins running, that the tab order is correct, and so on. You do not need to have a fully functional application to test your work.

To run your application, choose one of the following methods:

- Choose Run|Run.
- Click the Run button on the SpeedBar.

This compiles and then executes your program.

To terminate your application, choose one of the following methods:

- Double-click the form's Control-menu box.
- Choose Run|Program Reset.

See also

Component basics

Compiling, building, and running projects

Multiple Document Interface (MDI) applications

See also

MDI is a means for one application to simultaneously open and display two or more files. For example, you might use MDI to allow a word-processing application to open and display several documents simultaneously. Or, you might use MDI to create a spreadsheet application that lets users have several spreadsheets open at once. The text editor sample application is an MDI application.

MDI applications provide a convenient way to view and exchange data among multiple documents that share a common workspace. The common workspace is often called the parent, or frame. In a Delphi MDI application, the frame is always the application's main form.

Within the frame window, other windows, often called child windows, can be opened. Each child window appears and behaves the same way. The application documents, or other data, are opened inside the child windows. These documents can all be of the same format, or they can be a mix of file formats supported by the MDI application.

At design time, you create the MDI frame form as the main application form, and you create the child form as a template for the application's child windows. You can create more than one type of child form, but MDI applications support only one frame form.

The following topics describe some of the steps involved in creating an MDI application:

- Creating an MDI Frame Form
- Creating an MDI child form
- Merging MDI application menus
- Arranging and accessing open child windows

See also

MDI and SDI Forms
Working with text
Database applications
DDE client applications
DDE server applications

Creating an MDI frame form

See also

In an MDI application, the frame window provides a workspace for the application documents, which open inside one or more child windows.

The FormStyle property determines whether the form is a frame or child form (or a non-MDI application form). You set the FormStyle property only at design time.

To create an MDI frame form,

1. Select the main form.
2. Use the Object Inspector to set the main form's FormStyle property to fsMDIForm.

There can be only one MDI frame form per application.

Caution: You can set any form's FormStyle property to fsMDIForm, but your application will not compile correctly if you do so for a form that is not specified as the application's main form. To verify or change the application's main form, choose Options|Project and select the Forms page.

Rules

Here are some rules about MDI parent and child forms:

- The MDI parent form must always be the application's main form.

  If the MDI parent form isn't specified as the application's main form, the application won't be correctly compiled.
- There can be only one form of style fsMDIForm per application.
- The application's main form can never be of type MDIChild. This would also prevent the application from being properly compiled.
- In general, you should remove the MDI child form from the Auto-create forms list on the Forms page of the Project Options dialog box.

  This is because for true MDI applications, you create multiple instances of a single child form at run time. If, on the other hand, your application uses only one instance of each of several different types of child forms, as might be the case in a database application, you probably don't need to create each child form at run time. In this case, you needn't remove the forms from the Auto-create list.

See also

Creating an MDI child form

Multiple Document Interface (MDI) applications

Creating an MDI child form

See also

You create child forms at design time as templates for the appearance of each instance of the child window that the user opens at run time. Since child windows are visible by default, your code needs to explicitly create them at run time, rather than allowing Delphi to automatically create them when the application first runs.

To create an MDI child form,

1. Choose File|New|Form.
2. Set the form's FormStyle property to fsMDIChild.
3. Choose Options|Project to access the Forms Auto-create list.
4. In the Forms Auto-create list, select the MDI child form.
5. Move the MDI child form into the Available Forms list, and click OK.

Most MDI applications use just one type of child form, but you can have any number of different types of child windows in an MDI application. If your application uses multiple types of child windows, you'll generally want to remove them from the Forms Auto-create list, as just described, and write code that specifically creates them at run time. If, however, your application will use only one instance of each type of child form, it's probably better to let Delphi create them for you.

Caution: You can set any form's FormStyle property to fsMDIChild; however, your program won't compile correctly if you do so for the application's main form. To verify or change the application's main form, choose Options|Project and select the Forms page.

Note: When you close a form, what happens depens on the setting of the Action property of the form's OnClose event. If a form is an MDI child form, and its BorderIcons property is biMinimize, then the default Action is caMinimize. If a MDI child form does not have these settings, the default Action is caNone, meaning that nothing happens when the user attempts to close the form.

If a form is an SDI child form, Action defaults to *caHide*.

To close and form and free it in an OnClose event, set Action to caFree.

See also

Creating an MDI Frame Form
Multiple Document Interface (MDI) Applications
Instantiating Forms at Run Time

Arranging and accessing open child windows

See also

MDI applications provide the ability to open several child windows within one common workspace (the frame window). Because of this, MDI applications should always include a Window (or other) menu item that contains:

- Tile
- Cascade
- Arrange Icons
- Open document windows

See also

Multiple Document Interface (MDI) applications

Providing an area for text manipulation

Coding the Window menu commands

See also

MDI applications should always include a Window (or other) menu item that contains Tile, Cascade, and Arrange Icons commands to offer users an easy way to arrange their open documents in the client area of the frame window.

To handle the clicks for the Tile, Cascade, and Arrange Icons menu commands, generate OnClick event handlers for each menu item, and call the Tile, Cascade, or ArrangeIcons method as appropriate. For each event handler, you need write only one line of code--a method call--and Delphi does the rest for you.

For example:

```
procedure TFrameForm.Tile1Click(Sender: TObject);
begin
  Tile;
end;

procedure TFrameForm.Cascade1Click(Sender: TObject);
begin
  Cascade;
end;

procedure TFrameForm.ArrangeIcons1Click(Sender: TObject);
begin
  ArrangeIcons;
end;
```

See also

Including a list of open documents in a menu
Arranging and accessing open child windows
Multiple Document Interface (MDI) applications

Including a list of open documents in a menu

See also

MDI applications should always include a menu item that contains a list of the open document windows, which lets users quickly switch among them. (The window that currently has focus appears in the list with a check mark next to it.)

You can add a list of open documents to any menu item that appears on a menu bar in the MDI form. This list can, but need not, be included on the Window menu -- for example, it could be on a File or View menu. However, there can be only one such list per menu bar. The list of open documents appears below the last item in the menu.

To include a list of open documents as part of a menu, set the frame form's WindowMenu property to the name (not the caption) of the menu under which you want the list to appear.

To include an open document list in a menu,

1. Set the form style to fsMDIForm.

   This makes the form an MDI frame.
2. Create a menu for the MDI form that contains the menu item where you want the open document list to appear.
3. Select the frame form, and then select the Properties page of the Object Inspector.
4. From the drop-down list next to the WindowMenu property, select the name of the menu item under which you want the open document list to appear (for example, a Window or View menu.

   This name must represent an item that appears on the menu bar, not a submenu item, because document lists cannot be used in nested menus.

See also

Coding the window menu commands
Arranging and accessing open child windows
Multiple Document Interface (MDI) applications
Designing menus

Working with projects

See also

A Delphi project is a collection of the files that make up a Delphi application or distributable library. When you first start Delphi, the default new project opens. You can work with this default project, open another existing project,or start a different project by using one of Delphi's predefined Project templates.

Delphi's initial default new project uses the Blank Project template.You can change which project is specified as the default new project using the New Items dialog box. Future new projects then have the specified project as their starting point.

Choose a topic for more information.

About application projects

Basic project tasks

Managing projects and directories

See also

Working with units and forms
Project Manager basics
Saving projects and files
The project file

About application projects

See also

A Delphi application project consists of:

- Project file

  The project file is saved with a .DPR extension. There can be only one .DPR file per project.

- Form file

  Saved with a .DFM extension, these are binary files that contain the graphical image of the form. Every .DFM file has a parallel unit file with a .PAS extension.

- Unit (source code file)

  Unit files are saved with a .PAS extension. Every unit file is an Object Pascal source code file, but the unit file associated with the form is distinct from other types of unit files, because it is inseparable from the graphical form file (.DFM).

You add forms and their associated unit files as you develop your application. You can also add:

- Files written in earlier versions of Borland Pascal, or other programming languages. You include these files in a Delphi project by using compiler directives that tell Delphi the type of file you are including.
- Files that you did not create with Delphi, but are using from a library or other outside source (Non-Delphi resource files).

The project file ties all these files together, so that Delphi knows what to compile and link to create the target executable (.EXE). file or dynamic-link library (.DLL).

See also

Project files generated at design time
Compiler-generated project files
Non-Delphi resource files in projects

Files generated at design time

See also

Some project files are generated as you develop a project. These are mainly source code, configuration, and backup files. File- or project-saving operations within Delphi trigger their creation on disk. The following table summarizes these files and their DOS file extensions, and explains when they are written to disk.

Extension	Description	Purpose
.DPR	Project file	Object Pascal source code for the project's main program file. Lists all form and unit files in the project, and contains application initialization code. Created when project is first saved.
.PAS	Unit source	Object Pascal unit source code. One .PAS file is generated for each form the project contains when the project is first saved. (Your project might also contain one or more .PAS files not associated with any form.) Contains all declarations and procedures, including event handlers, for the form.
.DFM	Graphical Form file	Binary file containing the design properties of one form contained in the project. One .DFM file is generated along with the corresponding .PAS file for each form the project contains when the project is first saved.
.OPT	Project options file	Text file containing the current settings for project options. Generated with first save and updated on subsequent saves if changes were made to project options.
.RES	Compiler resource file	Binary file containing the application icon and other outside resources used by the project.
.~DP	Project backup file	Generated on the second save of a project and updated on subsequent saves, this file stores a copy of the .DPR file as it existed before the most recent save.
.~PA	Unit backup file	This file stores a copy of a .PAS file as it existed before the most recent save.If a .PAS file changes, this parallel file is generated on the second project save, or any save of the .PAS unit file after a change has occurred. It is updated on subsequent saves if the .PAS file has changed.
.~DF	Graphical form backup	If you open a .DFM file as text in the Code Editor and make changes, this file is generated when you save the .DFM file. It stores a binary-format copy of the .DFM file as it existed before the most recent save.
.DSK	Desktop settings	This file stores information about the desktop settings you have specified for the project in the Environment Options dialog box.

See also

Compiler-generated project files
Non-Delphi resource files

Compiler-generated project files

See also

Some project files are created on disk at compile time. These include the application executable file that you can deploy and distribute when the project is complete, and the object code for the project unit files (each compiled separately, to minimize time required for compilation). The following table summarizes the project files created and/or written at compile time.

Extension	Description	Purpose
.EXE	Compiled executable file	This is the distributable executable file for your application. This file incorporates all necessary .DCU files when your application is compiled. There is no need to distribute .DCU files with your application.
.DCU	Unit object code	Compilation creates a .DCU file corresponding to each .PAS file in the project.
.DLL	Compiled dynamic-link library	For information on creating DLLs, see the Delphi Component Writer's Guide.

See also

Files generated at design time

Non-Delphi resource files

Non-Delphi resource files

See also

Some files that you use in a Delphi project can be files created with applications other than Delphi, and need not reside in the project directory. When you correctly specify such files in a project, Delphi integrates them when you save or compile the project. Some of these files might be common to multiple projects. Some common examples of non-Delphi files used in Delphi projects are:

Image files	The bitmaps (.BMP, .WMF files) that you use in TImage-type components, or as glyphs on TBitBtn components, can reside anywhere on your system. When specified as properties of these graphical components, Delphi takes a "snapshot" of the disk file and stores this in the binary form (.DFM) file. They are eventually compiled into the project executable file.
Icon files	The icons (.ICO files) that you specify in the Icon property of forms and in the Project Options dialog box can also reside anywhere. They are integrated into the project in the same way as bitmap image files. .BMP and .ICO files can also be created with the Delphi Image editor, or you can choose from the bitmaps and icons available in the Delphi Image Library.
Windows Help files	The online Help (.HLP) file that you specify for the project in the Project Options dialog box can also reside anywhere. The .HLP file can be compiled using either the Microsoft Help Compiler shipped with Delphi, or another third-party Help compiling system. Only the file name and its path (if specified) are integrated into the project, not the Help file itself. This can affect the accessibility of Help when you deploy or distribute your application. For more information, see Setting Project Options.

See also

Project files generated at design time

Compiler-generated project files

Basic project tasks

See also

The following tasks describe how to perform basic functions while you are developing your Delphi project.

To start a new project,

1. Choose File|New to open the Select New Object Type dialog box.
2. Select either Application or Library.

To open an existing project,

- Choose File|Open.

To save a project,

- Choose File|Save.

To save a copy of a project,

- Choose File|Save As.

Note: Save As enables you to save the project file using different name or in a location. Choosing Save As also and updates all files included in the project. If files are shared by other projects,future modifications to the files are reflected in every project that references them. If you want to specify that project files not be shared, save them individually in a distinct project directory.

To save individual files,

- Choose File|Save As, and specify the directory where the project file resides.

To run your project, choose from the following methods:

- Choose Run|Run.
- Click the Run button on the SpeedBar.

To stop running your project,

- Double-click the Control-menu box of the application's main form.

To close your project,

- Choose File|Close All.

See also

Saving projects and files
Overview of projects
Managing projects and directories
Working with units and forms
Project Manager basics
The project file

Working with units and forms

Developing applications in Delphi involves working with forms and unit source code. The following topics describe basic tasks that you can perform with units and forms, independent of the project file.

Integrating units and forms into a project

Viewing units and forms in a project

Removing units and forms from a project

Saving units and forms

Form files

See also

The form is the focal point of a Delphi application. Whether you are adding components to the form, editing properties, or coding, you are editing the form.

Forms are stored in two separate files.

- The .DFM file stores a binary image of the form. Any edits you make to the form's visual properties, such as changing the height, color, border, and so forth, are stored in the .DFM file.
- The unit file (.PAS) stores the source code for the .DFM file. In the .PAS file you write event handlers that specify how the form and its components behave in the running application.

Delphi keeps the binary and source code files synchronized as you create and modify the form. Whenever you add a new form to your application, Delphi creates the associated unit file and add it to the **uses** clause in the Project file.

See also

Managing projects and directories
Units
Form unit file
Project files generated at design time

Managing projects and directories

See also

Choose one of the following topics for information about managing projects and directories:

Project Manager basics

Specifying the Help and icon files for the project

Specifying the main form for the project

Using project templates

Compiling, building, and running projects

Setting project options

Version control

See also

Managing projects and directories
Overview of projects
Saving projects and files
The project file
Working with units and forms

The project file (.DPR)

See also

When you begin a new project, Delphi generates a project file and maintains this file throughout the development of the project.

The project file controls a Delphi application. The project file is saved with a .DPR extension. There can only be one .DPR file per project.

Delphi generates the following source code for the project file:

```
program Project1
uses
  Forms,
  Unit1 in 'UNIT1.PAS' {Form1};

{$R *.RES}

begin
Application.Create(TForm, Form1);
Application.Run(Form1);
end.
```

Each time you add a new form or unit to the project, Delphi adds it to the **uses** clause in the project source code file.

Caution: Avoid manually editing the project file. If you edit a project file, you circumvent Delphi's automated project management mechanisms and risk maintaining inaccurate information about project composition. Compilation failures and other problems can result.

See also

Managing projects and directories
Saving projects
Unit (.PAS) file
Working with projects
Integrating units and forms into a project

Unit source code (.PAS) files

See also

Units are the main building blocks of Delphi applications. Unit files are identified by a .PAS file extension. They contain the Object Pascal source code for the elements of the Delphi applications you build. For example, every form has its own unit source code file. Unit source code files might belong to a project (that is, they are registered in the **uses** clause of the project .DPR file), but this is not required. You can create and save them as standalone files that any project can use.

If you open and save a default new project, the project directory initially contains one unit source code (.PAS) file, and one associated graphical form (.DFM) file. The .PAS file contains the source code for the form and its event handlers; the .DFM file contains binary code that stores the visual image data for the form.

Not all unit files need be associated with forms. If you write your own procedures, functions, DLLs, or components, their source code resides in a .PAS file, which you need to create, but no form is associated with the unit source code file. The rest of this section explains how to generate a default unit (.PAS) file for each purpose.

Form-associated unit files

The type of unit source code file you probably work with most is the form-associated unit. This is the type of unit created whenever you open a new form in the IDE. The following example represents the code that Delphi generates for Unit1 of a default blank project, and for every unit with an associated blank form that you add to the project. Of course, the default unit identifier will be incrementally updated (Unit2, Unit3, and so on) as you add new forms.

```
unit Unit1;
interface
uses
SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs;
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
end.
```

You can change which form is specified as your application's main form. For more information, see Tools|Repository

Form unit type declaration

Introduces the form as a class. A class is simply an object All the elements that make up a Delphi application, such as buttons, labels, fonts and so forth, are objects. Delphi objects encapsulate Windows behaviors, enabling you to interact with the object instead of directly with Windows.

Form unit var declaration

Declares the form as an instance of TForm. This means it contains all the behaviors and characteristics of the TForm object.

Form unit compiler directive

The $R compiler directive links the TForm's binary file (.DFM). This adds the .DFM file(s) in your project to the compiled executable.

Warning: Do not remove the $R *.DFM directive from a form unit file. Doing so will result in code that will never work correctly.

Unit files for components

If you elect to write a new Delphi component, you need to create a unit file for the source code. The default structure of the unit source code for a component differs somewhat from that of a unit associated with a form because the new component must be installed onto the Component palette.

To create a new component unit file,

1. Choose File | New.
2. Select Component.

Unit files for procedures and functions

You can write custom procedures or functions within any source code unit. However, those functions or programs can't be called from other code without making the unit where they reside part of the project that needs to use the procedure. If this would be unacceptable, then you might want to write the procedures or functions as standalone units that you can incorporate into projects and call from any unit. If this is what you want to do, you need to create a new unit source code file that has no associated form.

To create a unit file not associated with a form,

1. Choose File | New.
2. Select Unit.

It is not necessary to have a project open in the IDE unless you want the new unit to be part of a project.

The following code sample shows the default source code for a unit not associated with a form.

```
unit Unit1;
interface

implementation

end.
```

See also

Form-associated unit files
Unit files for components
Unit files for procedures and functions
Code Editor
Managing projects and directories
The project File

Integrating units and forms into a project

See also

You can use either the Project Manager or menu commands to create new forms or new unit files in a project, or to integrate existing files from locations outside the project directory. In order for any form or unit to "belong" to a project, the project must be open in the IDE so that Delphi can update the project source code (.DPR) file as new or shared unit files are added.

When you add an existing unit or form to a project and then make modifications to the form or unit, the modifications you make show up in all other projects using the form or unit.

Note: Adding a form to your project does not automatically give other units in the project access to the form or to routines declared in the new form's unit. You need to add the unit identifier whose routines or form you want to access to the appropriate **uses** clause in the unit accessing the routines or form.

To add a new form to your project, choose from the following methods:

- With a project open, choose File|New|Form.
- Click the New Form button on the SpeedBar.
- Choose New Form from the Project Manager SpeedMenu.

A new, blank form appears, along with a page in the Code Editor that contains the unit source-code file.

To add a new unit without adding a form, choose from the following methods:

- With a project open, choose File|New|Unit.
- Click the New Unit button on the SpeedBar.
- Choose New Unit from the Project Manager SpeedMenu.

A new page is added to the Code Editor, with the default unit source code displayed.

To add an existing form or unit to your project, choose from the following methods:

- Choose File|Add to Project.
- Click the Add button on the Project Manager SpeedBar.
- Choose Add File from the Project Manager SpeedMenu.

The Add to Project dialog box opens, where you can choose a unit or form file to include in the current project.

Note: You can open other files using this dialog box, but only units and forms will actually be added to the project. Other files, such as text files, will open in the Code Editor, where you can view or edit them, but will not be considered part of the project.

See also

Bringing existing units into a project
Project Manager basics
Removing units and forms from a project
Using project templates
Viewing units and forms in a project
Working with projects

Bringing existing units into a project

See also

A project can use existing form and unit files which it did not create, and which don't reside in the current project directory, or any subdirectory of that directory. Such files are referred to as shared files. This distinction is a purely conceptual one that can help you keep track of how you are using files in your projects. Delphi itself does not care whether the files that make up a project reside in the project directory, a subdirectory of the project directory, or any other location.

If you add a shared file to a project, bear in mind that the file is not copied to the current project directory; it remains in its current location. Adding the shared file to the current project simply registers the file name and path in the **uses** clause of the project's .DPR file. Delphi automatically does this as you add units to the project. The compiler treats shared files the same as those created by the project itself.

To add a shared file as part of the current project, do one of the following:

- Choose File|Add to Project.
- Click Add File on the Delphi Speedbar.
- Click Add on the Project Manager Speedbar.
- Choose Add File from the Project Manager SpeedMenu.

Any of these actions displays the Add To Project dialog box, in which you can select the file you want the current project to use. The Path column of the Project Manger's file list will display the path to the shared file.

See also

Integrating units and forms into a project
Removing units and forms from a project
Project Manager basics
Working with projects

Viewing units and forms in a project

See also

You can easily toggle between viewing the current unit or the current form by using commands from the View menu. You can also view any unit or form in a project by choosing commands from the View menu, buttons on the Project Manager SpeedBar, or commands from the Project Manager SpeedMenu.

To view a unit using the Project Manager,

1. In the Project Manager window, select the unit you want to view.
2. Choose one of the following methods:

- On the SpeedBar, click View Unit.
- From the SpeedMenu, choose View Unit.
- In the Unit name column, double-click the Unit.

The source code for the unit becomes the active page in the Code Editor. If the unit is not currently open, Delphi opens it for you.

To view a unit using the View menu,

- Choose Units.

To view a form using the Project Manager,

1. In the Project Manager window, select the form you want to view.
2. Choose from the following methods:

- On the SpeedBar, click the View Form button.
- From the SpeedMenu, choose the View Form command.
- In the Form name column, double-click the Form class name.

The form appears in its design-time mode. If the form is not currently open, Delphi opens it for you.

To view a form using the View menu,

- Choose Forms.

See also

Integrating units and forms into a project
Project Manager basics
Removing units and forms from a project
Viewing the project file
Working with projects

Viewing the project file

See also

You can view the project (.DPR) file to determine which units and forms are included in your project and to see which form is specified as your application's main form.

Important: Manually editing the .DPR file circumvents Delphi's automatic project updating mechanisms and is therefore not recommended.

To view the .DPR file,

- Choose View|Project Source.
- Choose the View Project command on the Project Manager SpeedMenu.

If the project file is not currently open, Delphi opens it for you.

To close the .DPR file,

- Choose File|Close.
- Choose Close Page from the Code Editor SpeedMenu.

Note: In order to close the project file it must be the active page in the Code Editor.

See also

Project Manager basics
Viewing units and forms in a project
Working with projects

Removing units and forms from a project

See also

You can remove forms and units from a project at any point during project development. The removal process deletes the reference to the file in the **uses** clause of the .DPR file.

Important: Removing a file from the project ends its relationship with the project, it does not delete the file from disk. Use the Windows File Manager or another utility to delete unwanted files from disk. See the Warning in this topic first.

To remove a unit from a project, open the project and choose any of these methods:

In the Project Manager window,

- Select the unit or units you want to remove, then choose the Remove button on the SpeedBar.
- Select the unit or units you want to remove, then choose Remove File from the SpeedMenu.

In the Code Editor,

- Select the tab for the unit you want to remove and choose File|Remove File from the Delphi menu bar.
- Select the tab for the unit you want to remove and choose the Remove File button on the Delphi SpeedBar.

Warning: Do not use Windows file management programs or DOS commands to delete Delphi project files from disk until you have performed the preceding removal process in every Delphi project that uses the files. Otherwise, the project (.DPR) file of each project using the deleted files will retain references to them in the **uses** clause. When you open the project again, Delphi will attempt to find the deleted files and will display error messages for each file it cannot find. When the project opens, the information about its constituent files in the Project Manager will be inaccurate.

Removing Graphical Form files

Because a graphical form (.DFM) file is always associated with a source code unit (.PAS) file, you cannot remove one without removing the other. You can, however, remove a unit file that has no associated form without affecting the relationship of other units to the project. Remember to remove any **uses** clause references to the removed unit that might exist in remaining units.

See also

Integrating units and forms into a project
Project Manager basics
Viewing units and forms in a project
Working with projects

Specifying the project Help and icon files

See also

If you have written a Help file for your application, you need to specify where it exists so that the application can open the file when the user requests Help.

Similarly, you can select the icon that appears when your application is minimized on the Windows desktop. Delphi links this icon into the .DPR file at compile time.

You specify the location of both these files by means of the Application options page of the Project Options dialog box.

To specify an icon or Help file for the project,

1. Choose Options|Project.

   The Project Options dialog box appears.
2. Select the Application Options page.
3. In the Auxiliary files section, specify the name of the application icon (.ICO) file, and/or the Help file that you want to use. Use the Browse button if you are not sure of the location of the files.

   The application icon file displays when your application appears in a minimized state on the user's desktop. Delphi links this file into the .DPR file at compile time.
4. Choose OK to accept your changes and close the dialog box.

See also

Non-Delphi resource files
Application Options dialog box
Working with projects

Specifying the main form for the project

See also

A project's main form is the form that users first see when they open your Delphi application. When the users or the program itself closes the main form, the application terminates.

Delphi automatically inserts the name each form added to a project into an Application . CreateForm statement. The application main form is the one specified in the first Application.CreateForm statement listed in the .DPR file, which lists the forms in the order you add them to the project. Thus, the first form you create or add is always the application main form.

You can, however, change the main form designation, specifying any form in the project as the application's main form. This is particularly useful if you want to test run a single form without having to access it from other forms.

To change the main application form,

1. Choose the Forms page of the Project Options dialog box.
2. Use the Main Form combo box to select the form that you want.
   This list box shows all forms in the current project.
3. Choose OK to accept your changes and exit the dialog box.

See also

Application (Options|Project Options)

Working with projects

Saving projects and files

See also

You can save your currently open project file as is, or under a different name and/or location. You can also save your project as a template so that you can reuse it. If you save a project as a template, you can continue to work on the project, and then save the project again without affecting the template you saved earlier.

Once all project files have been named, File|Save and File|Save As both prompt you to name only new (not modified) units; both then save any modifications to previously named project files.

When you choose File|Save As Delphi expects you to save the .DPR file file to a new location. If you do not specify either a new name or new location for the .DPR file, Delphi prompts you to specify whether you want to overwrite the existing .DPR.

See also

Using project templates

Working with projects

Saving files

See also

You can save individual files in a project or non-project files (such as text files) you might have open in the Code Editor. To save a file, it must be open.

To save an individual file,

1. Choose File|Save.

   If this is the first time you have saved the file, you are prompted to name it.
2. If necessary, enter name the file
3. Click OK.

   Delphi saves the file.

To save a file under a different name or location,

1. Choose File|Save As.

   The Save As dialog box appears.
2. Specify the new file name, location, or both, and choose OK.

   Delphi saves a copy of the file under the name and location you specify.

Note: This changes the name of the file, and if it is already part of the project, includes the file with the new name in the project. The older unit name still exists as a file but is not included in the project any longer.

See also

Saving projects

Working with projects

Saving projects

See also

To save all open project files, use one of the folowing methods:

- Choose File|Save.
- Click the Save Project button on theDelphi SpeedBar.
- Choose the Save command on the Project Manager SpeedMenu.

If you have not previously saved the project, the Save As dialog appears, displaying the name of the first unit in the project. If you have previously saved the project, you are prompted to save only those files that are new.

To save a newly created or added file, choose one of the methods above, then

1. Specify a file name for the unit, and choose OK.

   You continue to be prompted to save all remaining unnamed units in the project. (You are not prompted to save graphical form (.DFM) files separately.) You are then prompted to save the project file itself, if it has not previously been saved. This ensures that the unit and form names you have just specified will be reflected correctly in the .DPR file.

2. Specify a name for the project, and choose OK.

   Note that you need to give the project a unique name, that is, a name not previously used by any of the units in the project. If you attempt to give the project the same name as a unit, you will receive an error message. Simply rename the .DPR file, and choose OK.

See also

Saving files
Project Manager basics
Saving a separate version of the project
Updating the .DPR file
Working with projects

Saving a separate version of the project

See also

Once you have saved all the component files of a project, choosing File|Save As automatically updates any modifications you have made since last saving the files. It saves your changes to the project files just as a File|Save but saves only the .DPR file with the new name and/or location you specify.

For example, if you want to save modifications you have made to forms or units in the project without affecting other projects that might be using those same forms or units, you should separately save each file by choosing File|Save As.

To save the current project under a different name,

1. Choose File|Save As.
2. In the Save As dialog box, specify either a new project name, or a new location, or both.

   If a .DPR file with the same name exists in the directory you specify, you are prompted to specify whether you want to overwrite the existing file.
3. Choose OK to accept your changes and exit the dialog box.

   Delphi saves the .DPR file, and the .OPT (project options) and .RES (project resources) files under the file name and/or new location you specify. Delphi also saves any changes you might have made to open project files, so you are not prompted to save these changes again when you close the project. Changes to files saved with the Save Project As operation for one project are reflected in all projects that use the files.

At the end of the File|Save Project As process, the new copy of the project becomes the currently open project in the IDE. If you check the constituent file list in the Project Manager, you will see that all the files in the currently open version of the project reside in a directory other than the current project directory.

If you leave this new project version unchanged, it continues to use the files in their present (that is, old) location as shared files, which might or might not be what you want. If you don't understand how the new project is using its constituent files, you can run into problems later.

Warning: Do not use file management tools other than those in Delphi to save a copy of a project to a new location.

See also

Creating a backup of an entire project
Project Manager basics
Saving projects
Bringing in existing units
Sharing forms with other projects
Updating the .DPR file

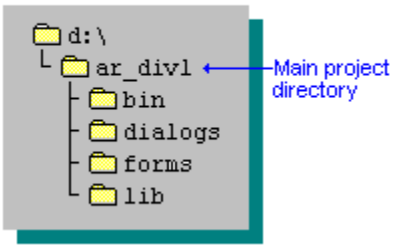
Creating a backup of an entire project

See also

Backing up a project can be a simple matter of copying directories or can involve some additional steps. This depends upon how your project directories are structured and whether or not the project uses files from outside its own directory tree.

The project directory isn't encoded into the project .DPR file. The project file does, however, record the location of all constituent project files. If these files reside in subdirectories of the main project directory, then all path information is relative, which makes backup easy, as shown in the following figure:

Example project directory tree



You could back up this entire project simply by copying the directory tree to another location. If you open the project at the backup location, all the project files that reside within that structure are present, and the project will compile.

If this project uses files that reside outside this tree, the project might or might not compile at the backup location. Check the Project Manager's file list to see if these outside files are accessible from the backup location. If they are, the project will compile. If another backup process already preserves these outside files, then there is probably no need to make separate backup copies of them in the backup project directory.

See also

Project Manager basics
Saving projects
Bringing in existing units
Sharing forms with other projects
Updating the .DPR file

Updating the .DPR file

See also

When you add and remove files from a project by using either the Project Manager or commands on the File menu, Delphi maintains the .DPR file for you, adding and removing unit and form names as appropriate.

If, however, you manually edit the .DPR file, Delphi has no way of tracking your changes and therefore cannot automatically stay synchronized with them. In this case, choose the Update button to resolve any discrepancies between your .DPR file and what is displayed in the Project Manager window.

To update the .DPR file,

- Click the Update button on the Project Manager SpeedBar.

The Update button remains dimmed unless there is a need to use it, so in general, if the button is available you should choose it.

See also

Project Manager basics
Saving projects
Saving a separate version of the project

File handling in the Project Manager

See also

The files that appear in the Project Manager are inclusive of every file in your project. Other files not displayed in the window might be compiled by Delphi at compile time, although they are not considered part of the project.

For example, the Project Manager window displays all the units listed in the **uses** clause of your project .DPR file, but not units listed in the **uses** clause of other units. Similarly, the Project Manager window does not display resource files that must be precompiled, such as icon files.

The following tables list the types of files you might find in your project directory, and describes how they are handled by the Project Manager.

Source files

File type	Project Manager displays?	Delphi compiles?	Save All button saves?
.DPR	No	Yes	Yes
.PAS	Yes	Yes	Yes
.DFM	Yes	Yes	Yes

Compiled files

File type	Project Manager displays?	Delphi compiles?	Save All button saves?
.EXE	No	Yes	No
.DLL	No	Yes	No
.DCU	No	Yes	No
.OPT	No	No	Yes
.DSK	No	No	Yes

Linked files

File type	Project Manager displays?	Delphi compiles?	Save All button saves?
.ICO	No	No	No
.RES	No	No	No
.OBJ	No	No	No

Other files

File type	Project Manager displays?	Delphi compiles?	Save All button saves?
.HLP	No	No	No

.EXE file
The compiled executable for your Delphi application, which you can deploy/distribute.

.DLL file
A compiled dynamic-link library file, which you can deploy/distribute.

.DCU file
Compiled .PAS (unit source code) file.

.OPT file
Stores modified project option settings.

.DSK file
Stores saved Desktop settings.

.ICO file
Windows icon files. Delphi adds the Application icon to the .DPR file.

.RES file

Resource files. Can be included in the project with $R compiler directive.

.OBJ file
Object code files. Can be included in the project with $L directive.

.HLP file

Compiled Windows Help file. Specify the Help file for your project from the Application page of the Project Options dialog box. For information on authoring Help, see Creating Windows Help.

.DPR file
Main project file source code. Lists the other units used in the project and their location.

.PAS file
Unit source code file. .PAS files for form units contain event handler code and local procedures, and have a parallel .DFM file.

.DFM file
Binary file containing the graphical image of a form. Has a parallel .PAS file.

See also

Project Manager basics

Working with projects

Using project templates

Delphi provides project templates, predesigned projects you can use as starting points for your own projects. Project templates are part of the Object Repository (located in the OBJREPOS subdirectory), which also provides form objects and experts.

When you start a project from a project template, (other than the blank project template), Delphi prompts you for a *project directory*, a subdirectory in which to store the new project's files. If you specify a directory that doesn't currently exist, Delphi creates it for you. Delphi copies the template files to the project directory. You can then modify it, adding new forms and units, or use it unmodified, adding only your event-handler code. In any case, your changes affect only the open project. The original project template is unaffected and can be used again.

Choose from one of these topics for more information:

Starting a new project from a project template
Adding a project to the Object Repository

See also

Managing projects and directories

Starting a new project from a project template

See also

Delphi provides project templates, predesigned projects you can use as starting points for your own projects. Project templates are part of the Object Repository (located in the OBJREPOS subdirectory), which also provides form objects and experts.

When you start a project from a project template, (other than the blank project template), Delphi prompts you for a *project directory*, a subdirectory in which to store the new project's files. If you specify a directory that doesn't currently exist, Delphi creates it for you. Delphi copies the template files to the project directory. You can then modify it, adding new forms and units, or use it unmodified, adding only your event-handler code. In any case, your changes affect only the open project. The original project template is unaffected and can be used again.

To start a new project from a project template,

1. Choose File|New to open the New Items dialog box.
2. Choose the Projects tab.
3. Select the project template you want to use in the new project.
4. Click OK.
5. In the Select Directory dialog box, specify a directory for the new project's files.

   A copy of the project template opens in the specified directory.

See also

Adding a project to the Object Repository

Using project templates

Adding a project to the Object Repository

See also

You can add your own projects and forms to those already available in the Object Repository. This is helpful in situations when you want to have a standard framework for programming projects throughout an organization.

To add a project to the Object Repository,

1. Open the project you want added to the Object Repository.
2. Choose Tools|Repository.
3. On the Project Template page, choose Add to display the Save Project Template dialog box.
4. In the Title text box, type a project title.

   The title for the template will appear in the Object Repository window.
5. In the Description field, enter text that describes the template.

   This text will appear in the Object Repository window's status bar.
6. In the Page field, choose the name of the page in the New Items dialog box (probably Projects) you want the template to appear on.
7. In the Author field, entet text indentifying the author of the application.

   Author informaton only appears when you select the View Details option from the SpeedMenu.
8. Click the Browse button to select an icon to represent this template in the Object Repository .
9. Click OK to save the current project as a projec template.

Note: If you make changes to a project template, those changes automatically appear in new projects created from that template. The will not affect previous projects created with that template.

See also

Starting a new project from a project template

Using project templates

Compiling, building, and running projects

See also

All Delphi projects have as a target a single distributable executable file, either an .EXE or a .DLL. You can view or test your application at various stages of the development. See the followint topics:

Checking source code syntax

Compiling a project

Building a project

Running a project

See also

Compiler directives

Managing projects and directories

Checking source code syntax

See also

The Compile menu on the Delphi menu bar provides you the option of running a syntax check on your project's source code. The checking is identical to that for a compilation, and errors in your code generate the same error messages. However, the process is faster because Delphi makes no attempt to generate object code, and if the check succeeds, no executable file is generated.

To check the syntax of any source code module,

1. Save the project source code (recommended but not required).
2. Choose Project|Syntax Check from the Delphi menu bar.

If there is an error in your code,

- The Code Editor window comes to the front.
- The unit source file page containing the error comes to the top of the Code Editor.
- The line containing the error is highlighted in the Code Editor.
- The Code Editor message window displays the error message.

Help for the error message is available by pressing F1.

See also

Compiling a project
Building a project
Running a projects
Compiling, building, and running projects

Compiling a project

See also

Compiling a project compiles all the source code files that have changed since the last time you compiled them.

To compile your project,

- Choose Project|Compile.

When you choose Project|Compile,

- The compiler compiles source code for each unit if the source code has changed since the last time the unit was compiled. This creates a file with a .DCU extension for each unit.
  If the compiler cannot locate the unit, the unit is not recompiled.
- If the **interface** part of a unit's source code has changed, all the other units that depend on it are recompiled.
- If a unit links in an .OBJ file (a file containing assembly language code), and the .OBJ file is newer than the unit's .DCU file, the unit is recompiled.
- If a unit contains an include (.INC) file, and the include file is newer than the unit's .DCU file, the unit is recompiled.

Once all the units that make up the project have been compiled, Delphi compiles the project file and creates an .EXE file named after the project. This .EXE file now contains all the source code and forms found in the individual units, and the program is ready for you to run.

You can choose to compile only portions of your code if you use compiler directives and predefined symbols in your code.

Obtaining compile status information

You can get information about the compile status of your project by displaying the Information dialog box (Choose Information from the Compile menu). This dialog box displays information about the number of lines of source code compiled, the byte size of your code and data, the stack and heap sizes, and the compile status of the project.

You can get status information from the compiler as a project compiles by checking the Show Compiler Status box in the Environment Options dialog box.

See also

Checking source code syntax
Building a project
Running a project
Compiling, building, and running projects

Building a project

See also

Building a project compiles every source code file in your project, regardless of when they were last compiled.

To build your project,

- Choose Project|Build.

The Project|Build command is similar to the Project|Compile command except it compiles everything, even if a file has not changed.

See also

Checking source code syntax
Compiling a project
Running a project
Compiling, building, and running projects

Running a project

See also

Running your project first compiles it (if it has changed since the last compile), then launches the .EXE file that runs your application.

To run your application, choose one of the following methods:

- Choose Run|Run.
- Click the Run button on the SpeedBar.

The Run|Run command does the same thing as the Project|Compile command except that when the compilation is complete, Delphi runs your application.

Because the compiler always creates a fully compiled executable file (.EXE), you can also run your application from the Windows Program Manager as you would run any other Windows application after you have compiled or built it.

See also

Checking source code syntax
Compiling a project
Building a project
Compiling, building, and running projects

Setting project options

See also

Delphi lets you choose from several options that affect the way your code is compiled. You can choose from two methods:

- Select the options you want by using the Project Options dialog box.
- Insert compiler directives into your source code.

Setting Options with the Project Options dialog box

You can use the Project Options dialog box to change the default compiler, linker, and directory options, and to define conditional symbols.

To display the Project Options dialog box,

- Choose Options|Project.

If you are unsure what a particular option does, press F1 or click Help to display information about these options.

Inserting compiler directives in code

Instead of using the Project Options dialog box to set compiler options, you can insert a compiler directive in your code.

For example, you can turn range checking on for your application by choosing the Range Checking option in the Project Options dialog box, or by placing the {$R} compiler directive in your code.

See also

Compiler directives

Managing projects and directories

Adding menu items

See also

1. Open the Menu Designer.
2. Select the position where you want to create the menu item.
3. Begin typing to enter the caption. Delphi automatically changes the name of the menu item to reflect the caption.
4. Press Enter.

   The next placeholder for a menu item is selected.
5. Enter values for the Name and Caption properties for each new item you want to create, or press Esc to return to the menu bar.

   Use the arrow keys to move from the menu bar into the menu, and to then move between items in the list; press Enter to complete an action.

Adding a separator bar to a menu

- Enter a hyphen as the caption of the menu item.

See also

Adding menu items dynamically
Inserting a menu item
Deleting a menu item
Specifying accelerator keys and keyboard shortcuts

Inserting a menu item

See also

- Place the cursor on a menu item, then press the Insert key.

  Menu items are inserted to the left of the selected item on the menu bar, and above the selected item in the menu list.

Deleting a menu item

See also

1. Place the cursor on the menu item you want to delete
2. Press the Delete key.

Note: You cannot delete the default placeholder that appears below the item last entered in a menu list, or next to the last item on the menu bar. This placeholder does not appear in your menu at run time.

See also

Adding menu items

Inserting a menu item

Deleting a menu item

Specifying accelerator keys

See also

- Add an ampersand (&) in front of the appropriate letter in the caption. The letter after the ampersand appears underlined in the menu.

Accelerator keys enable the user to access a menu command from the keyboard by pressing Alt+ the appropriate letter, indicated in your code by the preceding ampersand. The letter after the ampersand appears underlined in the menu.

Caution: Delphi does not check for duplicate accelerators, you must track values you have entered in your application menus.

Specifying keyboard shortcuts

See also

- Enter a value for the ShortCut property, or or select a key combination from the drop-down list. However, this list is only a subset of the valid combinations you can type in.

Keyboard shortcuts enable the user to perform the action without accessing the menu directly by typing the shortcut key combination.

Caution: Delphi does not check for duplicate shortcut keys, you must track values you have entered in your application menus.

See also

Adding menu items

Specifying accelerator keys and keyboard shortcuts

Creating nested menus

1. Select the menu item under which you want to create a nested menu.
2. Press Ctrl+Right arrow to create the first placeholder, or choose Create Submenu from the SpeedMenu.
3. Enter a name for the nested menu item.
4. Press Enter to create the next placeholder.
5. Repeat steps 3 and 4 for each item you want to create in the nested menu.
6. Press Esc to return to the previous menu level.

Organizing your menu structure using nested menus can save vertical screen space. However, for optimal design purposes you probably want to use no more than two or three menu levels in your interface design. (For pop-up menus, you might want to use only one such nested level, if any.)

Shortcut: You can also create a nested menu by inserting a menu item from the menu bar (or a menu template) between menu items in a list. When you move a menu into an existing menu structure, all its associated items move with it, creating a fully intact nested menu. This pertains to nested menus as well; moving a menu item into an existing nested menu just creates one more level of nesting.

Moving menu items

During design time, you can move menu items simply by dragging and dropping. You can move menu items along the menu bar, or to a different place in the menu list, or into a different menu entirely.

The only exception to this is hierarchical: you cannot demote a menu item from the menu bar into its own menu; nor can you move a menu item into its own nested menu. However, you can move any item into a different menu, no matter what its original position is.

While you are dragging, the cursor changes shape to indicate whether you can release the menu item at the new location. When you move a menu item, any items beneath it move as well.

To move a menu item along the menu bar,

1. Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new location.
2. Release the mouse button to drop the menu item at the new location.

Note: When you move a menu item to a different place on the menu bar, all the sub-items beneath it move as well.

To move a menu item into a menu list,

1. Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new menu.

   This causes the menu to open, enabling you to drag the item to its new location.

   Note: You cannot move a menu item down a level into its own menu.
2. Drag the menu item into the list, releasing the mouse button to drop the menu item at the new location.

Moving Menu Items into Submenus

When you moving a menu item off the menu bar, its sub-items become a submenu. Similarly, if you move a menu item into an existing submenu, its sub-items then form another nested menu under the submenu.

You can move a menu item into an existing submenu, or you can create a placeholder at a nested level next to an existing item, and then drop the menu item into the placeholder to nest it.

Editing menu items without opening the Menu Designer

See also

When you edit a menu item by using the Menu Designer, its properties are still displayed in the Object Inspector. You can switch focus to the Object Inspector and continue editing the menu item properties there. Or you can select the menu item from the Component list and edit its properties without ever opening the Menu Designer.

To edit a menu item without opening the Menu Designer,

- Select the item from the Component list.

To close the Menu Designer window and continue editing menu items,

1. Switch focus from the Menu Designer window to the Object Inspector by clicking the properties page of the Object Inspector.
2. Close the Menu Designer as you normally would.

   The focus remains in the Object Inspector, where you can continue editing properties for the selected menu item.

   To edit another menu item, select it from the Component list.

See also

Designing menus

Menu Designer SpeedMenu

Menu Designer SpeedMenu

See also

The Menu Designer SpeedMenu provides quick access to the most common Menu Designer commands, and to the menu template options.

To display the Menu Designer SpeedMenu,

Choose one of the following methods:

- Right-click anywhere on the Menu Designer.
- Press Alt+F10 when the cursor is in the Menu Designer window.

The first three commands on the Menu Designer SpeedMenu directly perform an action.

Command	Action
Insert	Inserts a placeholder above or to the left of the cursor
Delete	Deletes the selected menu item (and all its sub-items, if any)
Create SubMenu	Creates a placeholder at a nested level and adds an arrow to the right of the selected menu item

The rest of the commands on the Menu Designer SpeedMenu open dialog boxes. Choose a command for more information.

Select Menu

Save As Template

Insert From Template

Delete Templates

Insert from Resource

See also

Designing menus
Setting menu item properties by using the Object Inspector
Switching among menus at design time

Insert (Menu Designer SpeedMenu)

Choose Insert from the Menu Designer SpeedMenu to add a menu item placeholder before the selected menu item.

Delete (Menu Designer SpeedMenu)

Choose Delete from the Menu Designer SpeedMenu to remove the selected menu item.

Create Submenu (Menu Designer SpeedMenu)

Choose Create Submenu from the Menu Designer SpeedMenu to insert a menu item placeholder to the right of the selected menu item and add an arrow to the selected item to indicate a nested level.

Select Menu (Menu Designer SpeedMenu)

See also

Choose Select Menu from the Menu Designer SpeedMenu to open the Select Menu dialog box.

Select Menu dialog box

Use this dialog box to quickly select from among the existing form menus.

See also
Switching among menus at design time

Save As Template (Menu Designer SpeedMenu)

See also

Choose Save As Template from the Menu Designer SpeedMenu to open the Save Template dialog box, which enables you to save a menu for later reuse.

Save Template dialog box

Use this dialog box to save a menu for reuse.

See also
Saving a menu as a template

Insert From Template (Menu Designer SpeedMenu)

See also

Choose Insert From Template from the Menu Designer SpeedMenu to open the Insert Template dialog box.

Insert Template Dialog Box

Use this dialog box to add a predesigned menu to the active menu component.

See also

Using menu templates

Delete Templates (Menu Designer SpeedMenu)

See also

Choose Delete Templates from the Menu Designer SpeedMenu to open the Delete Templates dialog box.

Delete Templates dialog box

Use this dialog box to select and remove a predesigned menu.

Note: After you delete a template, you cannot retrieve it.

See also

Using menu templates

Insert from Resource (Menu Designer SpeedMenu)

See also

Choose Insert From Resource from the Menu Designer SpeedMenu to open the Insert Menu From Resource dialog box.

Insert Menu From Resource dialog box

Use this dialog box to import a menu from a Windows resource (.RC) file. You first need to save each individual menu as a separate resource file.

Dialog box options

File Name

Enter the name of the file you want to use, or enter wildcards to use as filters in the Files list box.

Files

Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type

Choose the type of file you want to open; the default file type is a menu file (.MNU). All files in the current directory of the selected type appear in the Files list box.

Directories

Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name input box or the file type in the List Files of Type combo box appear in the Files list box.

Drives

Select the current drive. The directory structure for the current drive appears in the Directories list box.

See also
Importing menus from resource files

Switching among menus at design time

See also

If you are designing several menus for your form, you can use the Menu Designer SpeedMenu or the Object Inspector to easily select and move among them.

To use the SpeedMenu to switch among menus in a form,

1. Right-click in the Menu Designer to display the SpeedMenu.
2. From the SpeedMenu, choose Select Menu.

   The Select Menu dialog box appears. This dialog box lists all the menus associated with the form whose menu is currently open in the Menu Designer.
3. From the list in the Select Menu dialog box, choose the menu you want to view or edit.

To use the Object Inspector to switch among menus in a form,

1. Give focus to the form whose menus you want to choose from.
2. From the Component list, select the menu you want to edit.
3. On the Properties page of the Object Inspector, select the Items property for this menu, and then either click the ellipsis button (...), or double-click [Menu].

See also
Menu Designer SpeedMenu

Using menu templates

See also

Delphi provides several predesigned menus, or menu templates, that contain frequently used commands. You can use these menus in your applications without modifying them (except to write code), or you can use them as a starting point, customizing them as you would a menu you originally designed yourself. Menu templates do not contain any event handler code.

Menu templates are stored in the file DELPHI.DMT. The menu templates shipped with Delphi also reside in this file. In a default installation, this file is in the \DELPHI\BIN directory. If you want to store the DELPHI.DMT file in a different directory, add the following lines to your WINDOWS\DELPHI.INI file, replacing "directory" with a directory you choose:

```
[Globals]
PrivateDir=directory
```

You can also save as a template any menu that you design using the Menu Designer. After saving a menu as a template, you can use it as you would any predesigned menu. If you decide you no longer want a particular menu template, you can delete it from the list.

To add a menu template to your application,

1. Right-click the Menu Designer window.

   The Menu Designer SpeedMenu appears.
2. From the SpeedMenu, choose Insert From Template.

   (If there are no templates, the Insert From Template option appears dimmed in the SpeedMenu.)

   The Insert Template dialog box opens, displaying a list of available menu templates.
3. Select the menu template you want to insert, then press Enter or choose OK.

   This inserts the menu into your form at the cursor's location. For example, if your cursor is on a menu item in a list, the menu template is inserted above the selected item. If your cursor is on the menu bar, the menu template is inserted to the left of the cursor.

To delete a menu template,

1. Right-click the Menu Designer window.

   The Menu Designer SpeedMenu appears.
2. From the SpeedMenu, choose Delete Templates.

   (If there are no templates, the Delete Templates option appears dimmed in the SpeedMenu.)

   The Delete Templates dialog box opens, displaying a list of available templates.
3. Select the menu template you want to delete, and press Del.

   Delphi deletes the template from the templates list and from your hard disk.

See also

Designing menus

Saving a menu as a template

Saving a menu as a template

See also

Any menu you design can be saved as a template so you can use it again. You can use menu templates to provide a consistent look to your applications, or use them as a starting point which you then further customize.

The menu templates you save are stored in the DELPHI.DMT file in your \DELPHI\BIN directory.

You edit the template file by using the template commands from the Menu Designer SpeedMenu.

To save a menu as a template,

1. Choose Save As Template from the Menu Designer SpeedMenu to open the Save Template dialog box.
2. In the Template Description edit box, enter a brief description for this menu.
3. Click OK.

   The Save Template dialog box closes, saving your menu design and returning you to the Menu Designer window.

Note: The description you enter is displayed only in the Save Template, Insert Template, and Delete Templates dialog boxes. It is not related to the Name or Caption property for the menu.

When you save a menu as a template, Delphi does not save its Name, since every menu must have a unique name within the scope of its owner (the form). However, when you insert the menu as a template into a new form by using the Menu Designer, Delphi then generates new names for it and all its items.

Delphi also does not save any event handlers associated with a menu saved as a template, since Delphi cannot test whether the code would be applicable in a new form. You can associate menu items in the template with existing event handlers in the form.

See also

Associating menu events with code

Using menu templates

Importing menus from resource files

See also

Delphi supports menus built with other applications, so long as they are in the standard Windows resource (.RC) file format. You can import such menus directly into your Delphi project, saving you the time and effort of rebuilding menus that you created elsewhere.

To load an existing .RC menu file,

1. In the Menu Designer, place the cursor where you want the menu to appear.

   The imported menu can be part of a menu you are designing, or an entire menu in itself.
2. From the Menu Designer SpeedMenu, choose Insert From Resource.

   The Insert Menu From Resource dialog box appears.
3. In the dialog box, select the resource file you want to load,
4. Choose OK.

Note: If your resource file contains more than one menu, you first need to save each menu as a separate resource file before importing it.

See also
Designing menus

What's new in Delphi 2.0

Delphi 2.0 includes several features that differ from previous versions. These include

- Compiler and run-time library changes
- New data types
- New compiler directives
- Compiler optimizations
- Calling conventions
- Unit finalization section
- Obsolete 16-bit features
- VCL changes
- OLE controls
- OLE automation
- New components
- Changes to TDBGrid
- Database changes
- Data modules
- Scalable data dictionary
- TField support for lookup fields
- Local filtering on tables
- Development-environment changes
- Visual form inheritance
- Visual form linking
- New compiler messages
- Fields Editor redesign
- Fields Editor drag-and-drop
- Menu changes
- Changes to the Gallery
- Moving from Delphi 1.0 to Delphi 2.0

New data types

Delphi 2.0 defines several new data types that reduce the limitations set by Windows 3.1.

The following are the new data types:

- Character types
- String types
- Variant type
- Currency type

The implementation-dependent types Integer and Cardinal are 32-bit values in Delphi 2.0, where they were 16-bit values in Delphi 1.0. To explicitly declare 16-bit integer data types, use the SmallInt and Word types.

Character types

See also New data types

Delphi 2.0 introduces new wide character types to support Unicode and other wide character sets. Delphi 1.0 treated characters as 8-bit ANSI values of type Char.

In this version of Delphi, the default character size is still 8 bits.

When writing code that might have to deal with characters of either size, be sure to use the SizeOf function, rather than hard-coding the character size or assuming a character size of one byte.

Delphi 2.0 also introduces new pointer types that represent pointers to zero-based arrays of the corresponding character types. The semantics of all the character-pointer types are identical. The only thing that varies is the size of the character pointed to.

See also

Character types

Character-pointer operators

String types

See also New data types

Delphi 2.0 supports strings of nearly unlimited length in addition to the 255-character counted strings previously supported. A new compiler directive, **$H**, controls whether the reserved word **string** represents a short string or a new, long string. The default state of **$H** is **$H+**, using long strings by default.

Note: All Delphi 2.0 components use the new long string type. If you have existing components with string-type properties, you should compile them in the **$H+** state to remain compatible.

Note that you can always specify a short string of a specified maximum length, no matter what the setting of the **$H** compiler directive. Specifying a string length in brackets always creates a subtype of short string, as shown in the following declarations.

```
{$H+}{ turn on long strings }
var
  S1: string;       { S1 is of type ANSIString }
  S2: string[80];   { S2 is a short string with maximum length of 80 characters }
```

Standard procedures

Delphi provides two new standard procedures for dealing with long strings:

- SetLength
- SetString

There are also three new functions in the SysUtils unit (Trim, TrimLeft, TrimRight) for trimming whitespace from strings. Trimming whitespace removes all leading and/or trailing whitespace, including space characters and control characters.

See also

String types
Long string types
Short string types

Variant type

See also New data types

Delphi 2.0 introduces variant types to give you the flexibility to dynamically change the type of a variable. This is useful when implementing OLE automation or certain kinds of database operations where the parameter types on the server are unknown to your Delphi-built client application.

A variant type is a 16-byte structure that has type information embedded in it along with its value, which can represent a string, integer, or floating-point value. The compiler recognizes the standard type identifier Variant as the declaration of a variant.

In most cases you can use a variant just as you would any other type of variable. When performing OLE automation, variants can respond to method calls from the OLE server.

See also

Variant types

Variants and OLE automation objects

Currency type

See also New data types

Delphi 2.0 defines a new type called Currency, which is a floating-point type specifically designed to handle large values with great precision. Currency is assignment-compatible with all other floating-point types (and variant types), but is actually stored in a 64-bit integer value much like the Comp type.

Currency-type values have a four-decimal-place precision. That is, the floating-point value is stored in the integer format with the four least significant digits implicitly representing four decimal places.

See also
Real types

New compiler directives

Delphi 2.0 introduces two new compiler directives:

- a local switch directive, **$H**, which controls the use of long string types
- a local switch directive, **$J**, which controls the writeability of typed constants

In addition, there are now long compiler directives corresponding to most of the single-letter directives. For example, the new **$H** directive has a corresponding **$LONGSTRINGS** directive.

The Object Pascal language Help has a complete list of all compiler directives, long and short.

Compiler optimizations

Delphi's 32-bit compiler performs several kinds of optimizations when generating object code.

Benchmarks indicate that using optimizations can speed execution of computation-intensive code by a factor of three or four times. However, the optimizations will not rearrange your code so as to change the meaning of your algorithms.

The following table summarizes the optimizations available.

Type of optimization	Meaning
Register optimizations	Places heavily used variables and parameters into CPU registers to reduce the number of clock cycles required to access the items. Includes "lifetime analysis" of variables so that variables used in different parts of code can share the same register.
Call-stack overhead elimination	Delphi passes parameters in CPU registers when possible, rather than pushing them onto the stack. This eliminates the need to set up a stack frame, speeding up procedure, function, and method calls. For further details, see Calling conventions.
Common subexpression elimination	The compiler locates repeated computations in complex mathematical expressions and reduces the number of times the repeated code must execute. You can therefore write your code in the clearest, easiest-to-read way, and the compiler will ensure that it produces efficient code.
Loop induction variables	Replaces index variables for arrays and strings used in loops with an incremented pointer, reducing the number of multiplications required for access. In addition, if the elements accessed have a size of 2, 4, or 8 bytes, the compiler uses Intel scale indexing to further increase efficiency.

You can turn off optimizations by choosing Project|Options and unchecking the Optimization item on the Compiler page. However, we strongly advise against changing this setting.

Calling conventions

See also

The Delphi 2.0 compiler provides more options for parameter-passing conventions. In addition to the **pascal**, **cdecl**, and **stdcall** conventions used in Delphi 1.0, Delphi 2.0 adds a **register** convention that is a speedy hybrid of the Pascal and C calling conventions. Delphi uses the **register** convention by default.

The **register** convention uses registers when possible to pass its parameters. The first three parameters that will fit into 32-bit registers are passed in the EAX, EDX, and ECX registers, in that order. All remaining parameters follow the Pascal calling convention. In addition, the **register** convention guarantees that the EBX, ESI, and EDI registers are preserved.

See also
Calling conventions

OLE controls

Delphi 2.0 enables you to import OLE controls (or OCX controls) and treat them as Delphi components, just as you could with VBX controls in Delphi 1.0.

To import an OLE control,

1 Choose Component|Install to open the Install Component dialog box.

2 Choose OCX to open the Import OLE Control dialog box.

3 OLE controls must be registered with Delphi before you can use them.

   If the OLE control you want to use appears in the Registered Controls list box, select it and proceed to step 6. Otherwise, choose Register to open the Register OLE Control dialog box.

4 Use the Register OLE Control dialog box to locate and open an OLE control. To open the control click Open.

   Delphi returns you to the Import OLE Control dialog box where the newly registered OLE control appears in the Registered Controls list box.

5 Select the OLE control from the Registered controls list.

   When you import an OLE control, Delphi must generate a component wrapper for that control. The Unit File Name and the Class Name edit boxes enable you to specify a file name and a class name for the OLE control. However, when you select the control Delphi automatically enters names into these edit fields.

6 Click OK.

   Delphi closes the Import OLE Control dialog box, and returns you to the Install Component dialog. The new OCX control appears in the Installed Units list.

7 On the Install Component dialog box, click OK. Delphi closes this dialog box, rebuilds the component library, and installs the OCX component on the Component palette.

Unit finalization section

See also

You can include an optional finalization section in a unit. Finalization is the counterpart of initialization, and takes place when the application shuts down. You can think of the finalization section as "exit code" for a unit. The finalization section corresponds to calls to ExitProc and AddExitProc in Delphi 1.0.

The finalization begins with the reserved word **finalization**. The finalization section must appear after the **initialization** section, but before the final **end.**

See also

Units

OLE automation

See also

OLE Automation is a mechanism for Windows applications to manipulate one another, much like a shared macro language. An application that can be automated is called an automation object or automation server. An application that automates another is an automation controller or automation client. OLE Automation is essentially a protocol by which one application can control the actions of another.

Delphi fully supports OLE 2.0 automation of applications. You can use an application written with Delphi to automate another application, or you can set up your application as an OLE automation server.

You can find complete examples of OLE automation clients and servers in the DEMOS\OLEAUTO directory in your Delphi installation.

This material is not intended to explain the deep, technical details of the OLE Automation system in Windows. Rather, it describes what you need to know to perform the most common automation tasks with Delphi:

- Automating another application
- About OLE automation servers
- About OLE automation objects
- Creating an OLE automation server

See also

Variants and OLE automation objects

OLEAuto unit

Automating another application

See also Example OLE automation

Delphi applications can automate other applications that are OLE automation servers. That is, the other applications must provide a run-time interface through OLE objects. Delphi applications access such OLE objects through variants.

There are three parts of automating an OLE application:

- Creating the OLE object instance
- Setting OLE object properties
- Calling OLE object methods

Creating the OLE object instance

Each OLE Automation server has a key called ProgID in the system registry, which identifies the server to its clients. In order to control that object, your automation client must create an instance of the automation object based on its ProgID.

To create an OLE object instance, call the CreateOleObject function and assign its result to a variant.

CreateOleObject takes a single string as its parameter, which is the ProgID in the registry for the automation object. If there is no automation object registered with that ProgID, CreateOleObject raises an exception.

For example, to create an OLE object that controls Microsoft Word, you would declare a variant variable and create an OLE object for it:

```
var
  MSWord: Variant; { declare holder for OLE object }
begin
  MSWord := CreateOleObject('Word.Basic');       { create instance from ProgID }
  ...{ use the object }
end;
```

Although the OLE object is not truly an object in the sense of the objects and classes in Object Pascal, Delphi allows you to manipulate them using a similar syntax. That is, given a variant that contains an OLE object, you can set its properties and call its methods much as you would those of a "real" object.

Setting OLE object properties

Many automation objects include properties in their interfaces. Properties represent the state or content of the automation server.

Although the implementation of properties in the OLE Automation interface is not the same as the property interface for Delphi classes, the Delphi compiler allows you to use the same syntax to refer to automation-object properties you would use for other properties.

Calling OLE object methods

Nearly all automation object include methods in their interfaces. As with Delphi objects, methods represent the actions associated with the server.

The implementation of methods in the OLE Automation interface is not the same as the method interface for Delphi classes, the Delphi compiler allows you to use the same syntax to call automation-object methods you would use for other properties.

The compiler cannot determine whether a particular method name or its parameters will be valid when actually calling into the OLE object at run time. It therefore packages the method name, along with any parameters, into a packet to be dispatched at run time.

All parameters are passed to OLE objct methods as variants, and any results returned are also interpreted as variants.

Example

The following example shows a simple example that takes data from a query, inserts it into a Microsoft Word document, and formats the inserted text as a table. You must have Microsoft Word running with a document open for the automation to work.

1 Create a new application with a blank form.

2 Add a **uses** clause to the **implementation** part of the blank form's unit that adds the OleAuto unit:

```
uses OleAuto;      { put this in the implementation part }
```

3 Place a Query component on the form.

4 Set the query's DatabaseName property to the DBDEMOS alias.

5 Place a Button component on the form.

6 Attach the following handler to the button's OnClick event:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MSWord: Variant;
  I: Integer;
  S: string;
begin
  Query1.SQL := 'select company, contact, phone from customer ' +
    'where lastinvoicedate >= ''1/1/95''';       { SQL statement for query }
  with Query1 do
  begin
    Open;  { open the query }
    try
      while not EOF do
      begin
        for I := 0 to Query1.FieldCount - 1 do
          S := S + Query1.Fields[I].AsString + #9;      { add tab-separated fields to
string }
        Next;      { repeat for each record }
      end;
      MSWord := CreateOleObject('word.basic');   { create object for server }
      MSWord.Insert(Copy(S, 1, Length(S) - 1));  { insert text }
      MSWord.ParaUp(1, 1);{ move to top of paragraph }
      MSWord.TextToTable(ConvertFrom := 1, NumColumns := FieldCount,
        Format := 16);    { convert to table }
    finally
      Close;        { always close the query }
    end;
  end;
end;
```

See also
Creating an OLE automation server

About OLE automation servers

See also OLE automation

An OLE automation server is an application or dynamic-link library (DLL) that exports OLE objects to OLE automation clients.

There are three important aspects of automation servers you need to understand before you create them:

- In-process and out-of-process servers
- Automation object instancing
- The Automation object

You can find a complete example of an OLE automation server and a client that automates it in the DEMOS\OLEAUTO\AUTOSERV directory.

See also

Creating an OLE automation server

About OLE automation objects

In-process and out-of-process servers

See also About OLE automation servers

There are two kinds of OLE automation servers in Windows: in-process and out-of-process (also called local) servers. You can create either kind of server with Delphi.

- An in-process server is a dynamic-link library (DLL) that exports automation objects. Because the automation objects come from a DLL, they are part of the same Windows process as the client application.

  In-process servers are useful for creating program modules shared by a number of applications that might use different languages. They have the advantage that they run in the same address space as the calling application, so calls to the server don't have to be marshalled, which adds a huge message-handling overhead to each call.

- An out-of-process server is a standalone application that exports automation objects.

See also

Automation object instancing

The Automation object

Automation object instancing

See also About OLE automation servers

Some OLE automation servers can only create and export a single OLE object, while others can handle multiple OLE objects. Still others cannot export OLE objects at all, keeping their OLE objects only for internal use. This relationship between a server and the number of objects it can export is called instancing.

When you create an OLE automation object, you specify its instancing. That way, Windows knows whether it needs to create a new instance of the server when asked for an instance of a particular automation object.

The following table describes the three kinds of instancing.

Instancing type	Meaning
Internal	OLE objects are internal to the application. The OLE objects are not registered, and therefore external processes cannot create them.
Single	Each instance of the server can only export one instance of the OLE object. If clients request multiple instances of the OLE object, Windows starts a new instance of the server for each one.
Multiple	Each server instance can create and export multiple instances of the OLE object. In-process servers (DLLs) are always multiple-instance.

See also

In-process and out-of-process servers

The Automation object

The Automation object

See also About OLE automation servers

Every project that uses the OLEAuto unit automatically incorporates an object called Automation. Automation is a nonvisual object. Much as the Application and Screen components generated as part of every Delphi application encapsulate portions of the application environment, Automation encapsulates the OLE automation aspects of the server project.

The two most important aspects of the Automation object are its StartMode property and its OnLastRelease event.

StartMode indicates how and why the OLE server was started. The following table describes the four possible values of StartMode.

Value	Meaning
smStandAlone	A user started the application.
smAutomation	Windows started the application for the purpose of creating an OLE object.
smRegServer	The application was started solely to register one or more OLE objects.
smUnregServer	The application was started solely to unregister one or more OLE objects.

OnLastRelease is an event indicating that a server started by the system for automation purposes (StartMode is smAutomation) is no longer needed as a server, because all clients have released all OLE objects created by the server. By default, the server shuts down in that case, but you can handle the OnLastRelease event to provide other checks to determine whether to actually shut down the server. OnLastRelease gets a Boolean **var** parameter called ShutDown, which is True by default. By setting ShutDown to True, you can prevent the server from shutting down when its last OLE object is released.

For more details on the Automation component, see the OLEAuto unit source code.

See also
In-process and out-of-process servers, Automation object instancing

About OLE automation objects

See also OLE automation

All OLE objects exported by Delphi automation servers descend from the class TAutoObject. There are several important things to know about these objects.

- Never call the Free or Destroy methods of an automation object.

  OLE objects are reference-counted, because they might be in use by more than one client. When finished using an OLE object, call its Release method. Release decrements the reference count, and if that brings the reference count to zero, calls Free to destroy the object.

- OLE objects are generally externally created. You can, however, construct one within the server by calling its Create constructor as you would for any other class. Keep in mind, however, the preceeding warning about calling Release, rather than Free, when done.
- OLE object constructors are virtual. If you override the constructor of an OLE class type, be sure to add the **override** directive to the declaration.
- Always export OLE objects as variants. Any method or property that returns an OLE object must return it as a variant containing the OLE object. TAutoObject provides a property named OLEObject for just this purpose. It is important that you not export classes or pointers to classes out of the server. Instead, always reference the OLEObject property of the OLE object, and export it as a variant.

  The methods in the MemoAuto unit in the AutoDemo example show this clearly.

See also

About OLE automation servers
Automating another application
Creating an OLE automation server

Creating an OLE automation server

See also OLE automation

Whatever kind of automation server you create, you must define its interface to clients, which consists of defining and registering the OLE object and automating properties and methods so that clients can reference them.

When you change the interface of an existing automation server, you should always make sure the interface is backward-compatible. That is, don't remove properties or methods already included, as that will cause errors in existing clients. You should only add to existing interfaces.

- If you modify an existing interface in ways that are not backward-compatible, you should change the object's name as well.

There are three parts to creating an automation server:

- Creating the automation server
- Adding the automation object
- Automating properties and methods

See also

About OLE automation servers

About OLE automation objects

Automating another application

Creating the automation server

See also Creating an OLE automation server

The first step in creating an OLE automation server is to create the server itself. That is, you create the application or dynamic-link library (DLL) that will create and export OLE objects. The choice of whether to create a DLL or application rests largely on whether you will be creating an in-process or out-of-process server.

The initial steps are no different from creating any other application or DLL, but each then has its own particular steps.

To create an in-process automation server (DLL),

1. Create a DLL.

   You'll add units and forms as usual, and you can also add OLE objects.
2. Add the OLEAuto unit to the **uses** clause in the project source file.
3. Export four standard entry points from the OLEAuto unit from the DLL.

   If you already have items exported from the DLL, you can add the four OLE automation entry points to the existing exports clause. Otherwise, you add the following code to the DLL:

```
exports
  DllGetClassObject, DllCanUnloadNow, DllRegisterServer, DllUnregisterServer;
```

   - You must spell these entry points exactly as shown, including the capitalization. Unlike all other items in Object Pascal, these items are case-sensitive.

To create an out-of-process automation server,

1. Create a Delphi application.
2. Add the following line of code immediately after the begin in the project source-code file:

```
if Automation.ServerRegistration then Exit;
```

After you create your server, you are ready to add the OLE automation object.

See also

Adding the automation object

Automating properties and methods

Adding the automation object

See also Creating an OLE automation server

Once you have created a server, whether an application or a dynamic-link library, you can then add an OLE automation object. The process is largely automated, but you need to supply some information to the Automation Object Expert.

To add an automation object to your server,

1 Choose File|New, and choose Automation Object from the Object Repository.

   Delphi opens the Automation Object Expert.

2 Name the automation object.

   This is the name used internally by your server to identify the OLE object. It must be a valid Object Pascal identifier, and by convention, its name should start with the letter T.

3 Name the OLE class.

   This is the name used externally to create these objects. When your server registers the OLE object with Windows, it places this name in the system registry. Client applications use this name when calling CreateOLEObject (called CreateObject in Visual Basic).

4 Describe the object being exported. This string goes into the registry.

5 Specify the instancing for the object.

   For in-process servers (DLLs), this is always Multiple Instance, applications are more typically Single Instance.

6 Choose OK to generate the automation object.

The Automation Object Expert produces the following:

- An empty automation object declaration, descending from TAutoObject.
- Registration code that calls Delphi's OLE Automation manager, which in turn registers the object with Windows, placing the object and server in the system registry.

Note: Part of the process of generating registration code for your automation object includes generating a globally-unique ID (GUID) for the server. In general, you should never change this once generated. The pairing of the OLE class name and GUID is unique, and if you change one or the other, you will cause errors in applications using those automation objects. You can, however, safely edit the portions of the registration information containing the description and instancing.

See also

Creating the automation server

Automating properties and methods

Automating properties and methods

See also Creating an OLE automation server

OLE automation clients generally manipulate OLE objects by calling their methods and setting and reading properties. Your server defines what properties and methods are available by adding those properties and methods to the OLE objects it exports.

The properties and methods of an automation object are just like those of any other Object Pascal class. There are a few restrictions, listed below.

To add a property or method to the interface of an OLE object, declare it in the **automated** part of the class declaration.

The **automated** part of a class declaration is just like a **public** part, but the compiler creates an entry in the automation table for that class.

The following restrictions apply to code in the **automated** part of a class declaration:

- Only properties and methods can be declared. Field declarations are not allowed.
- All property types, parameter types, and function result types used in property and method declarations must belong to the following set of types:
  SmallInt, Integer, Single, Double, Currency, TDateTime, String, WordBool, and Variant
- Property declarations can only include access specifiers (**read** and **write**). No other specifiers (**index**, **stored**, **default**, **nodefault**) are allowed.
- Access specifiers must list a method identifier. Field identifiers are not allowed.
- Property-access methods must use register calling conventions.
- Array properties are supported.
- Property overrides (property declarations that don't include the property type) are not allowed.
- Method declarations must use register calling conventions. Methods can be virtual, but not dynamic. Method overrides are allowed.
- A property or method declaration can include an optional **dispid** directive, which much be followed by an integer constant expression that gives the dispatch ID of the property or method.
  If a **dispid** clause is not present, the compiler automatically picks a number one larger than the largest dispatch ID used by any property or method in the class and its ancestors.
  Specifying an already-used dispatch ID in a **dispid** clause causes an error.

See also

Creating the automation server

Adding the automation object

Menu changes

To better conform with emerging development-tool menu standards, the menu structure of the Delphi 2.0 IDE is different than the menu structure of Delphi 1.0. The primary differences in Delphi 2.0 are the redesign of the File menu, the absence of Compile and Options menus and the presence of Project, Component, and Database menus.

The following table summarizes the major menu changes between Delphi 1.0 and Delphi 2.0:

Delphi 1.0	Delphi 2.0
File|New Project	File|New... Choose Application from the dialog box.
File|Open Project	File|Open... Use the Open dialog box to open an existing project.
File|Save Project	File|Save All Saves all open files, including the current project.
File|Close Project	File|Close All
File|New Form	File|New... Choose Form from the dialog box.
File|New Unit	File|New... Choose Unit from the dialog box.
File|Save File	File|Save
File|Save File As	File|Save As
File|Close File	File|Close
File|Add File	File|Add To Project or Project|Add To Project
File|Remove File	File|Remove From Project or Project|Remove From Project
Compile|Compile	Project|Compile
Compile|Build All	Project|Build All
Compile|Syntax Check	Project|Syntax Check
Options|Project	Project|Options
File|New Component	Component|New or File|New... and choose Component from the dialog box.
Options|Install Component	Component|Install
Options|Open Library	Component|Open Library
Options|Rebuild Library	Component|Rebuild Library
Options|Environment	Tools|Options
Options|Gallery	Tools|Repository The Object Repository replaces the Gallery in Delphi 2.0
Options|Tools	Tools|Tools
Help|Database Form Expert	Database|Database Form Expert

The following table lists new menu commands for Delphi 2.0:

New menu command	Meaning
File|New...	Choose new object from dialog box
File|Open...	Choose existing object from dialog box
File|Save As...	Choose object type from dialog box
File|Use Unit	Add a unit to a form unit's **uses** clause
Project|Add To Project	Same as File|Add To Project
Project|Remove From Project	Same as File|Remove From Project
Component|Configure Palette	Opens the Palette dialog box for customizing the Component Palette.
Database|SQL Monitor	Run the SQL Monitor
Database|Explore	Run the Database Explorer
View|Threads	Displays thread information while debugging

Also new on the File menu is a list of previously-opened files numbered from 0 to 10. Items 0 to 5 are the most recently opened projects, while items 6 to 10 are the five most recently opened files of any type.

The following table lists menu commands from Delphi 1.0 that are not supported in this version of Delphi.

Tools|ReportSmith
Tools|Image Editor
Tools|Database Desktop
Tools|BDE Config
Help|Interactive Tutors

- To open Delphi project files in the code editor (instead of as a project), choose File|Open..., select Text files (*.TXT) in the Files of type list, and type the name of the project (for example, PROJECT1.DPR) in the File name box.

New compiler messages

The Delphi compiler generates new, clear error messages, warnings, and hints in a message pane in the Code Editor window. You can control the generation of hints and warnings through project options and local compiler directives.

The compiler can also generate multiple error messages instead of always stopping at the first error.

When you receive messages from the compiler, you can click a message in the message pane and press F1 to get an explanation of the error message, including examples of what causes the error and how to solve the problem.

New components

Delphi 2.0 includes components representing the new Windows 95 common controls and a new multiple-record database grid. The source code for the new Windows 95 components is in SOURCE\VCL\COMCTRLS.PAS. The VCL reference in Help contains information about all the new components and their properties, methods, and events.

The new components are

TTabControl	TPageControl	TTreeView	TTrackBar
THeaderControl	TProgressBar	TRichEdit	TUpDown
TListView	TStatusBar	TDBCtrlGrid	

Moving from Delphi 1.0 to Delphi 2.0

Delphi 2.0 is fully compatible with Delphi 1.0. To make your 16-bit Delphi 1.0 application into a full 32-bit application, you just recompile it with Delphi 2.0. However, some code changes might be needed to accomodate Delphi's new data types and new Windows features.

The following VBX controls which were included with Delphi 1.0 are not included in this version of Delphi. You cannot compile a Delphi 1.0 project with Delphi 2.0 that uses any of these components:

- TBiSwitch
- TBiGauge
- TBiPict
- TChart

You can share both source code files and forms between 16-bit Delphi 1.0 projects and 32-bit Delphi projects, provided that none of the components listed above are used in the code or forms.

- You cannot conditionally compile a unit for both Delphi 1.0 and Delphi 2.0. For example, you can't use compiler directives such as **$IFDEF** in a single source code file to use a TTabbedNotebook for Delphi 1.0 or a TPageControl for Delphi 2.0.

You can't share a .DPR project file between Delphi 1.0 and Delphi 2.0 because the associated .RES resource file must be 16-bit for Windows 3.1 and 32-bit for Windows 95. Instead, you must create a new project file with Delphi 2.0 and add forms and units to it as needed. Maintain a separate project file for Delphi 1.0.

Delphi 2.0 has merged the WinTypes and WinProcs units into the Windows unit. To facilitate your migration, Delphi aliases the two obsolete units to the new unit so your port appears seamless. This also enables you to install your 16-bit components into the component library, CMPLIB32.DCL.

- You can open a Delphi 1.0 project file with Delphi 2.0, but if you recompile or save any changes, you will no longer be able to use the resource file (*.RES) with the Delphi 1.0 project. Backward compatibility is not maintained for resources.

To handle a 32-bit return value, the optional last parameter of the BlockRead and BlockWrite procedures is now an Integer type. In Delphi 1.0 the last parameter was of type Word. The Count parameter is also now an Integer type.

Obsolete 16-bit features

A number of features present in 16-bit versions of Delphi do not appear in Delphi 2.0. These include certain run-time library routines that dealt with the segmented memory architecture. The following table lists the features not supported in Delphi 2.0.

CSeg	DSeg	SSeg
PrefixSeg	PtrRec	SPtr
Seg	Ofs	Ptr
EBreakpoint	EFault	EGPFault
EInavlidOpCode	EPageFault	EProcessorException
ESingleStep	EStackFault	

Visual form inheritance

Example

In Delphi 1.0, all forms designed in the Forms Designer descended directly from the same ancestor, TForm. Delphi 2.0 enables you to derive forms from other ancestors, inheriting their components, properties, and code as a starting point for your own forms. Ancestor forms can be any forms already in your project or in the Object Repository.

Form inheritance lets you create a library of standard form templates, either within a single application or across an entire suite of applications. Any changes made to the ancestor form immediately appear in your descendant forms. You can customize each form type you derive from your library while still retaining the ability to modify the standard form and have those changes reflected in the derived forms.

To use visual form inheritance, you should understand

- Creating an inherited form
- Calling inherited event handlers

Example

If you installed the Delphi sample code, you can find a complete example that demonstrates form inheritance in the DEMOS\DB\GDSDEMO directory. That application includes two forms called GridViewForm and RecViewForm that share a common ancestor, StdDataForm. Changes made to StdDataForm appear immediately in the two descendant forms.

Creating an inherited form

See also Visual form inheritance

When you create a new form in Delphi, you have the option of either starting with a blank form or starting from an already-designed form type. When you start from an already-designed type, you can either copy that form, inherit from it, or use it.

When browsing items in the Object Repository, you'll find the option to copy or inherit on the right side of the dialog box.

- When you copy the template form, Delphi makes a duplicate of the form and all its components and code. If you make several copies in the same project, you generate duplicates of the copied code for each copy.
- When you inherit from the template form, you create a reference to the ancestor form, and only generate additional code for added components and event handlers. If you inherit several forms in the same project, they share the inherited code. All the ancestors of the chosen form are also added to the project.

Inheriting forms is a good way to reduce the size of projects that use a number of similar forms. It also provides a way to create and maintain a set of standard form templates that work in a number of projects, even when each project requires unique customizations.

- Inherit is not enabled when a form Expert is selected, since these cannot be ancestor forms. Choosing Copy on a non-inherited form works the same as before: creating a new form that is a copy of the form chosen.

See also
Calling inherited event handlers

Calling inherited event handlers

See also Example Visual form inheritance

When you attach an event handler to an event in a visually-inherited form, you can invoke the inherited handler from within the new handler. This enables you to modify the inherited behavior without having to duplicate it.

To invoke an inherited event handler, use the **inherited** reserved word.

Example

If you create a new form that inherits from a form called TStandardForm that contains a button called DoItButton, you can modify the action performed when the user clicks DoItButton by attaching a handler to its OnClick event, and within that handler calling the inherited handler:

```
procedure TForm1.DoItButtonClick(Sender: TObject);
begin
  { perform special actions }
  inherited;       { perform TStandardForm's handling }
end;
```

If the ancestor form does not have a corresponding event handler, the call to **inherited** does nothing. If you later add a handler to the ancestor, **inherited** then calls it.

See also
Creating an inherited form

Visual form linking

Components that refer to other components in their properties can now refer to components that are not on the same form. The most common kind of components that use such links are data-access components.

For example, you can now have a table component on one form in an application and allow several different forms to provide different views onto the same data set, such as a grid view and a form view.

Delphi links forms by linking their associated units. Given two forms, Form1 and Form2, and their associated units, Unit1 and Unit2, respectively, components on Form1 can refer to components on Form2 if Unit1 contains Unit2 in one of its **uses** clauses.

- You can use the File|Use Unit menu command to add one or more units in a project to the **uses** clause of another unit.

Fields Editor redesign

Delphi 2.0 has a completely new Fields Editor. Like its predecessor, the new Fields Editor is a modeless dialog box that displays all the fields in the selected data set.

These are the changes in the Fields Editor:

- The Fields Editor no longer has any buttons. All actions come from a popup menu activated by a right-mouse-button click.

This facilitates use with the editor's drag-and-drop capability.

- Support for lookup fields as well as calculated and data fields.
- Access to extended field attributes through the scalable data dictionary.

Fields Editor drag-and-drop

You can use the Fields Editor for a data-set (table or query) component to create data-editing controls on a form. Once you have a data set with fields defined, you can drag and drop a field name to the form containing the data set. If you have not already added a data-source component to the form, Delphi will add one automatically when you drop the first field on the form.

Delphi creates a control with a type appropriate to the field type. For example, dropping a field of type TStringField will create a DBEdit component, while dropping a field of type TGraphicField will create a DBImage component.

Changes to the Gallery

See also

The Gallery in previous versions of Delphi has been replaced by an Object Repository.

Unlike the Gallery, the Object Repository is not a static directory that holds Forms and Projects you can copy into your projects. Instead, it is a referential file that points to the locations of the Forms and Projects in the Object Repository.

The following topics are important for using the new Object Repository:

- Adding items to the Object Repository
- Customizing the Object Repository

See also

About the Object Repository

Object Repository file format

Object Repository file format

The Object Repository is an editable (.INI format) text file that contains a list of Forms and Projects. This file contains relevant information about each Object Repository item. Although you can edit the Object Repository file manually, we strongly recommend that you maintain it through the Object Repository dialog box instead.

The sections in the DELPHI32.DRO file are the path and file names of the project or form, or the IDString of experts.

There are four types of Object Repository Items, each with a different set of information:

- FormTemplate
- ProjectTemplate
- FormExpert
- ProjectExpert

The template types have the following information, most of which affect their appearance in the "New Items" dialog box:

Section type	Item	Value
FormTemplate	Name	Title of template
	Icon	Full path name of 32x32 icon (.ICO) file
	Description	Brief description of Object Repository item
	DefaultMainForm	Boolean value of whether to use this as the default main form
	DefaultNewForm	Boolean value of whether to use this as the default new form
	Ancestor	Path and file name of form which this form inherits from (must be in Object Repository)
ProjectTemplate	Name	Title of template
	Icon	Path and file name of 32x32 icon (.ICO) file
	Description	Brief description of Object Repository item
	DefaultProject	Boolean value of whether to use this as the default new project
FormExpert	DefaultMainForm	Boolean value of whether to use this as the default main form
	DefaultNewForm	Boolean value of whether to use this as the default new form
ProjectExpert	DefaultProject	Boolean value of whether to use this as the default new project

Adding items to the Object Repository

See also

Adding projects or forms to the Object Repository is the same as for the Gallery in Delphi 1.0. However, since the form or project in the Object Repository is a reference to the template locations, the templates must be saved (unmodified) before being added to the Object Repository.

Before you can add an inherited form to the Object Repository, you must first explicitly add all that form's ancestors to the Object Repository.

See also

Adding a form to the Object Repository

Using project templates

Changes to TDBGrid

The database-grid component, TDBGrid, now allows you to specify display properties on a column-by-column basis, independent of the properties of the underlying field. In this preliminary release, TDBGrid supports data cell color, width, field name, and title (column heading) text-column attributes.

To understand the new DBGrid features, you should understand

- DBGrid operating modes
- Defining column properties

DBGrid operating modes

See also Changes to TDBGrid

TDBGrid operates in one of two modes: without column definitions and with column definitions.

- When no column attributes are defined, the DBGrid behaves the same as the 16 bit TDBGrid: "column" information such as title text comes directly from the database fields, and modifications to the column width or column order will modify the associated field's DisplayWidth and Index properties, respectively. Changing the width or order of a field in one grid will affect all other grids using that same data set.
- When column definitions are defined, they completely define the appearance of the DBGrid. Columns may be assigned field names in any order and in any combination, independent of the field order in the underlying table. If a field name is not assigned to a column, or the field name cannot be found in the DBGrid's current data set, the column is blank and read-only. One field can be used in multiple columns of the same dbgrid (say, invoice number at both the left and right ends of a large grid), and the grid can be configured to show only a subset of the fields available in a data set.

See also
Defining column properties

Defining column properties

See also Changes to TDBGrid

A column obtains default values for all its column attribute properties from the DBGrid and/or from the field associated with the column. If you assign different values to the column properties, you override the defaults. These new values are persistent (stored and loaded with the grid).

For example, the width of a column is selected from one of the following sources, in order of preference:

- a value assigned to the Column.Width property
- a pixel-width calculated from the DisplayWidth property of the column's field
- the grid's DefaultWidth property (if no field is associated with the column)

The data cell color of a column is the color that was assigned to the column's Color property, or if no assignment has been made to the column's color, from the grid's Color property.

When a DBGrid has columns defined, modifications to the display of the columns are stored in the columns, and do not affect the underlying database fields. Changing the order of columns in the grid will not change the order of the fields in the database table. Similarly, changing the width of a column will not affect the DisplayWidth property of any associated field. Two DBGrids with column definitions can now present completely different views of the same data set, showing different subsets of the data set's fields, in different column order, with different widths, colors, and titles.

The Columns property

The DBGrid stores its column-property information in column objects of type TDBColumn. You access these column definitions through the DBGrid's new Columns array property. If a DBGrid does not have columns defined, Columns.Count is zero and indexing into the Columns property will raise an exception.

The TDBColumn.Field property returns the database field associated with the column, if any. If the DBGrid's data-source/data-set link has not been established, or if the column's FieldName property is blank, or if FieldName is invalid in the current data set, TDBColumn.Field returns **nil**.

Column indexes match the true column order, including the indicator column. This means that the first data column is Column[1] when the DBGrid indicator is enabled, and Column[0] when the indicator is disabled.

Note: This may change before the retail release of this product.

The Columns editor

DBGrid columns can be configured at design time using the DBGrid Columns Editor. Select the DBGrid in the form, then double click on the Columns property in the Object Inspector to bring up the DBGrid Columns Editor dialog box. From this dialog box, you can add, delete, reorder (by dragging items in the listbox), and set the properties of the grid's columns. Deleting all columns forces the DBGrid to revert to 16-bit compatibility mode.

The TDBGrid.Columns property and every individual column object are assignment compatible. For example,

```
DBGrid1.Columns := DBGrid2.Columns
```

will clear all column attributes in DBGrid1 and copy all column information from DBGrid2. Column field names are copied, but the grid's data set is not changed. Similarly,

```
DBGrid1.Columns[5] := DBGrids.Columns[2]
```

will clear and copy column information between these two individual columns.

See also
DBGrid operating modes

TField support for lookup fields

Example

Delphi 2.0 adds a new kind of field support. In addition to data fields and calculated fields, you can now have lookup fields. Lookup fields are much like calculated fields, but their "calculation" is a lookup in another table. Lookup fields are always read-only.

To define a lookup field, define a field in the Fields editor, then set its Lookup property to True with the Object Inspector.

Note: The Calculated and Lookup properties of a field component are mutually exclusive. Setting one to True will automatically set the other to False.

Once you create a lookup field, you need to specify values for four of its properties to indicate where and how it looks up its values, as described in the following table. As the names of the properties reflect, the relationship of the lookup field to the lookup table is much like a master-detail relationship.

Property	Meaning
LookupDataSet	Specifies the data set to look for the values in.
LookupKeyFields	Specifies the field or fields in the data set used to key the lookup.
KeyFields	Specifies the field or fields in the lookup table corresponding to the master fields in this data set. If the lookup table is a local (Paradox or dBASE) table, it must have and index that maps to these fields.
LookupResultField	Name of the field in the lookup table returned. The type of the result field must be the same as the type of the lookup field, since its value will become the value of the lookup field. If the lookup field is a string field, make sure that the Size property of the field matches that of the result field; otherwise, you will only see blanks.

The last three properties in the table are all strings. The master field and detail field strings can be semicolon-separated lists of fields.

Note: Field lookup happens before field calculations are performed. You can therefore base calculated fields on lookup fields, but you cannot base lookup fields on calculated fields.

Example

A simple but clear example of using a lookup field is a database that contains two tables, one for customers and one for orders. If the order table contains fields such as order number and customer number, you can define a lookup field in the order table that displays the customer name. Such a lookup field would use the customer number in the order table, match it to a customer number in the customer table, and return the name of the company from the customer table.

Using the orders and customer tables in the DBDEMOS database that comes with Delphi, you can define a lookup field in the orders table (ORDERS.DB) that displays a company name from the customer table (CUSTOMER.DB).

1 Create a new application.

2 Place two table components, a data source, and a data grid on the form.

3 Connect Table1 to ORDERS.DB, DataSource1 to Table1, and Table2 to CUSTOMER.DB.

4 Create a new string field called CustName in Table1 and set its Lookup property to True.

By default, a defined field is calculated. Setting Lookup to True will set Calculated to False.

5 Set the other lookup properties as shown in the following table:

Property	Value
LookupDataSet	Table2
LookupKeyFields	CustNo
KeyFields	CustNo
LookupResultField	Company

Local filtering on tables

Delphi allows you to locally filter records from a data set for display, much as you would use a query to filter records from a database server into a local data set. Using filtering, you can hide or show records locally from any table or some queries.

- In the shipping version of Delphi, you will be able to filter from all queries, as well.

Delphi data-set filters are written as Object Pascal code. That is, you can perform any calculations or manipulations in your filters that you can write code for.

There are two ways to use data-set filtering:

- Filtering records in a data set
- Finding records that match the filter

Whichever method you choose, you also need to understand writing a data filter.

Filtering records in a data set

See also Example Local filtering on tables

The simplest and most common way to use filtering is to turn on filtering for the entire data set. With filtering on, the data set generates an event for each record in the data set. In handling that event, you can determine whether to accept or "filter out" each record.

To turn on filtering in a data set, set the Filtered property to True.

With the Filtered property set to True, the data set generates an OnFilterRecord event for each record it retrieves. The handler for OnFilterRecord tests the record to determine whether it passes the filter, and only those that pass are visible in the data set.

You can change the filtering of records at run time by assigning a different handler to the OnFilterRecord event. However, you must then call Refresh to actually cause the new filter to take effect.

- Changing the filter or altering the current record can result in making the current record disappear if it no longer passes the filter. If this occurs, the next record that does pass the filter becomes the current record.

See also

Finding records that match the filter

Writing a data filter

Finding records that match the filter

See also Example Local filtering on tables

You can use filtering to locate individual records within an unfiltered data set. You can have Delphi iterate through the data set and locate records that match the filter by calling methods, collectively known as "find" methods.

To locate records that match the filter, call the FindFirst, FindNext, FindPrior, or FindLast methods.

All the find methods are Boolean functions, returning True if they locate a matching record. For instance, if you call FindFirst, it iterates through the data set, generating OnFilterRecord events until a record passes the filter (as explained in Writing a data filter). If it finds such a record, it makes that the current record and returns True. Otherwise, the current record stays the same and FindFirst returns False.

You can always check the result of the most recent call to any of the find methods by checking the value of the Found property.

See also

Filtering records in a data set

Writing a data filter

Writing a data filter

See also Example Local filtering on tables

A filter for a Delphi data set is a handler for the OnFilterRecord event. The data set generates OnFilterRecord events when it is filtering all records in a local data set or when finding individual records with the find methods. In either case, the way you write the filter is the same.

As with any event handler, the data filter is Object Pascal code called in response to a specific event. Your filter can be as simple or as complex as you need. Keep in mind, however, that the filter must be applied to all the records in the data set, and in some cases this could be a performance penalty. If you need to filter large amounts of data, you should probably use an index with a range instead of a local filter.

To filter data, write and handler for the OnFilterRecord event, setting the Accept parameter to indicate which records to filter out.

Accept is True by default, meaning that all records are included in the filter. Your data filter must set Accept to False to exclude specific records.

Calculated and lookup fields are not available when filtering. If you need to use calculated fields, your filter will have to perform the calculations itself.

Example

Here is a simple filter for a customer database that hides all customer records from the state of California by checking the value of the field called State:

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
  Accept := DataSet['State'] <> 'CA';     { True if not from CA }
end;
```

See also

Filtering records in a data set

Finding records that match the filter

Delphi 2.0 documentation team

The printed and online documentation for Delphi 2.0 was produced by the Borland Technical Publications department. The Delphi Tech Pubs group is managed by Jim Regan. The team includes:

- Richard Nelson, Lead Writer
- James Von Hendy, Senior Technical Writer
- Bob Simpson, Senior Technical Writer
- Chloe Redon, Technical Writer
- Susanne Edgerton, Technical Writer
- Bob Arnson, Technical Writer
- Frances Buran, Indexer
- Geta Carlsen, Editor
- Kimberly Haeger, Intern

with production assistance from Merry Bilgere, Bette Huntalas, and David Yeater.

Special thanks to Mike "F1 Boy" Satenberg.

About data modules

A TDataModule is a specialized Delphi class for centralized location and handling of any nonvisual component in an application. Typically these are data access components (TSession, TDatabase, TTable, TQuery, TStoredProc, TBatchMove, and TReport), but they can also be other nonvisual components (TTimer, TOpenDialog, TSaveDialog, TImageList, and TDdeClientConv, for example). A data module enables you to:

- Place all your data access components in a single visual container at design time instead of duplicating them on each application form.
- Design tables and queries once for use with many forms instead of recreating them separately for each form.
- Create business rules—using component events, and additional methods you add to the unit code for a data module—that can be shared across an entire application.
- Separate business logic and data access from user interface code for easier maintenance.
- Standardize common dialogs, timers, DDE client, DDE client items, DDE server, DDE server items, and image lists across an application.
- Store well-designed data access modules in the Object Repository to share with other projects and developers.

To use a component from a data module in a form, you add the data module to the form's uses clause, then use a component's methods or properties in the form. For data access components, set the DataSource property of each data-aware control in the form to point to the data source in the module.

Note: Delphi still supports putting nonvisual components directly on individual forms for simple applications and for compatibility with older applications. If you have older applications, consider converting them to use data modules. Using data modules produces simpler applications that are easier to maintain.

At run time a data module is not visible. Your application code can still change or read the properties of the components in the data module, and it can call the methods belonging to those components.

Creating a new data module

To create a new, empty data module for a project, choose File|New Data Module. Delphi opens a data module container on the Integrated Development Environment (IDE) desktop and adds the data module to the project file.

At design time a data module looks like a standard Delphi form with a white background and no alignment grid. As with forms, you can place nonvisual components on a module from the Components palette, and you can resize a data module to accommodate the components you add to it. You can also right-click a module to display a SpeedMenu for it. The following table summarizes the SpeedMenu options for a module.

Menu item	Purpose
Align To Grid	Aligns data access components to the module's invisible grid.
Align	Aligns data access components according to criteria you supply in the Alignment dialog box.
Revert to inherited	Discards changes made to a module inherited from another module in the Object Repository, and reverts to the originally inherited module.
Creation Order	Enables you to change the order in which data access components are created at start-up.
Add to Repository	Stores a link to the data module in the Object Repository.
View as Text	Displays the text representation of the data module's properties.

Behind the data module container you see in the IDE, there is a corresponding unit file containing source code for the data module.

Naming a data module and its unit file

The title bar of a data module displays the module's name. The default name for a data module is DataModuleN where N is a number corresponding to the number of forms and modules in an application plus one. For example, if you start a new project, and add a module to it before doing any other application building, the name of the module defaults to DataModule2. The corresponding unit file for DataModule2 default to Unit2.

You should rename your data modules and their corresponding unit files at design time to make them more descriptive. You should especially rename data modules you add to the Object Repository to avoid name conflicts with other data modules in the Repository or in applications that use your modules.

To rename a data module,

1. Select the module.
2. Edit the Name property for the module in the Object Inspector.

   The new name for the module appears in the title bar when the Name property in the Object Inspector no longer has focus.

When you change the name of a data module at design time, Delphi changes references to the module throughout your source code. References are not automatically changed in event handlers you write without using the Object Inspector.

To rename a unit file for a data module,

1. Select the unit file.
2. Choose File | SaveAs.

   In the Save As dialog box, enter a file name that clearly identifies the unit with the renamed data module.

Placing and naming components

You place non-visual components, such as TTable and TQuery, in a data module just as you place visual components on a form. Click the desired component on the appropriate page of the Component palette, then click in the data module to place the component. You cannot place visible controls, such as grids, on a data module. If you attempt it, you receive an error message.

For ease of use, components are displayed with their names in a data module. When you first place a component, Delphi assigns it a generic name that identifies what kind of component it is, such as DataSource1 and Table1. This makes it easy to select specific components whose properties and methods you want to work with. To make it even easier, you should give your components more descriptive names (e.g., CustSource and CustTable).

To change the name of a component in a data module,

1. Select the component.
2. Edit the component's Name property in the Object Inspector.

   The new name for the component appears under its icon in the data module as soon as the Name property in the Object Inspector no longer has focus.

When you name a component, the name you give it should reflect the type of component and what it is used for. For example, for database components, the name should reflect the type of component, and the database it accesses. For example, suppose your application uses the CUSTOMER table. To access CUSTOMER you need a minimum of two data access components: a data source component, and a table component. When you place these components in your data module, Delphi assigns them the names DataSource1 and Table1. To reflect that these components use CUSTOMER, and to relate the components to one another, you could change these names to CustSource and CustTable.

Using component properties and methods in a data module

Placing components in a data module centralizes their behavior for your entire application. For example, you can use the properties of dataset components, such as TTable and TQuery, to control the data available to the data source components that use those datasets. Setting the ReadOnly property to True for a dataset prevents users from editing the data they see in a data-aware visual control on a form. You can also invoke the Fields editor for a dataset to restrict the fields within a table or query that are available to a data source and therefore to the data-aware controls on forms.

In any case, the properties you set for components in a data module apply consistently to all forms in your application.

In addition to properties, you can write event handlers for components. For example, a TDataSource component has three possible events, OnDataChange, OnStateChange, and OnUpdateData. A TTable component has over twenty potential events. You can use these events to create a consistent set of business rules that govern data manipulation throughout your application.

For a complete list of the properties and events available for components, see the individual component entries in the online VCL Reference.

Note: Ensure that a data module is the first auto-created form in your project. Select Project|Options to bring up the Project Options dialog box. Choose the Forms page, and specify that the data module should be the first auto-created form in the project.

Creating business rules in a data module

Besides writing event handlers for the components in a data module, you can code methods directly in the unit file for a data module. These methods can be applied to the forms that use the data module as business rules. For example, you might write a procedure to perform month-, quarter-, or year-end bookkeeping. You might call the procedure from an event handler for a component in the data module.

The prototypes for the procedures and functions you write for a data module should appear in the module's type declaration:

```
  type
TCustomerData = class(TDataModule)
  Customers: TTable;
  Orders: TTable;
  .
  .
  .
  procedure LineItemsCalcFields(DataSet: TDataSet); { A procedure you add }
private
  { Private declarations }
public
  { Public declarations }
end;

var
CustomerData: TCustomerData;
```

The procedures and functions you write should follow in the implementation section of the code for the module.

Reusing data modules in the Object Repository

Instead of creating a data module from scratch, you may be able to reuse an existing data module from the Object Repository. Data modules in the Object Repository are modules that you or other developers created that are generic enough to be of use in different projects.

You can reuse an existing data module from the Object Repository at any time. To see the data modules available in the Object Repository:

1. Choose File|New.
2. Click the Data Modules page in the New Items dialog box.

The Data Modules page displays the data modules you can use. Radio buttons at the bottom of the page enable you to specify the method of reuse.You can borrow a module in one of three ways: Copy, Inherit, and Use. Sometimes one or more of these options may be unavailable. Unavailable options are dimmed.

Copying a data module

Copying a data module from the Object Repository puts an exact duplicate of the data module and its code in your project. Your copy of a data module is like a snapshot of the data module in the repository at the time of the copy operation. The data module copied into your project exists independently of its ancestor data module in the repository.

Copy a data module from the repository when

- The module already provides a fundamental level of access to data your application needs,
- The demands of your application require substantial changes and additions to the components in the module, and
- You do not want changes made to the ancestor data module in the repository to affect the module in your application.

Note: If you want future changes made to ancestor data modules in the repository to ripple into your project, you should inherit the data module instead of copying it.

Inheriting a data module

Inheriting a data module from the Object Repository creates a duplicate of the data module in your project, and creates a link to the ancestor module in the repository. Your copy of the module inherits any subsequent changes made to the components, properties, and methods of the repository's data module. Changes made to the data module in the repository are applied to the inherited data module in your project the next time you recompile. These changes apply in addition to any changes or additions you make to the data module in your project.

If you add components and event handlers to your copy of an inherited data module, you only generate new code in your application for the added components and event handlers.

Inherit a data module from the repository when

- The module provides a fundamental level of access to data your application needs, and
- You want to propagate changes made to the ancestor data module in the repository to the module in your application (for example, because you want to maintain a consistent corporate view of data in all your applications).

Note: Do not inherit data modules that contain TSession or TDatabase components. You should copy these modules instead of inheriting them. A second option is to put TSession and TDatabase components in a separate data module that you can copy, and to put all other components in another data module that you can inherit.

Inheriting event handlers

If you override an event handler for a component inherited from a data module (for example, to provide functionality specific to your application), you can still invoke the inherited handler from your new handler. When you override the handler, the new handler is created with a single statement: inherited. This call invokes the event handler from the ancestor data module if it exists. If it does not exist, the inherited statement is ignored. If you do not want to call an inherited handler from your new handler, delete the inherited statement. For more information about using inherited handlers, see the User's Guide.

Using a data module

You can both copy a data module from the Object Repository into your application and propagate any changes you make to it back into the original module in the repository by selecting the Use radio button in the New Items dialog box. Using a data module is like reverse inheritance. Instead of inheriting changes others make to a data module, they inherit your changes when they use the data module in the repository.

Use a data module in the repository when

- The module already provides a fundamental level of access to data your application needs,
- The demands of your application require substantial changes and additions to the components in the module, and
- The changes you need to make in your application are also fundamentally applicable to other applications.

Note: Be careful with the Use radio button option. Make sure the changes you make to a data module are robust and thoroughly tested before letting others copy it into their applications from the repository.

Accessing a data module from a form

To associate visual controls on a form with a a data module, you must first add the data module to the form's uses clause. To add a module to a form's uses clause, you can:

- Choose File | Use Unit and enter the name of the module or pick it from the list box in the Use Unit dialog box,

or

- Drag selected field from the Fields editor of a TTable or TQuery component in the data module to a form. In this case

Delphi prompts you to confirm that you want to add the module to the form's uses clause, then creates database edit boxes for each field you dragged into the form.

Note: If you employ this last method, Delphi also adds a new data source component to the form and sets the DataSource properties of the edit controls it creates to point to this data source. The data source itself is associated with a table or query component in the data module. You can keep this arrangement, or, to keep your data access model cleaner, can change it. Delete the data source component on the form, and set the DataSource properties of the control on the form to point directly to the appropriate data source in the data module.

Adding a data module to the Object Repository

You can save your data modules and add them to those already available in the Object Repository. This is helpful when you want to develop standard data access models throughout an organization.

Before adding a data module to the Object Repository, you should develop and test it as fully as possible to minimize additional work for future users of the module.

To store a data module in the Object Repository,

1. Save the data module.
2. Right-click the module to bring up the SpeedMenu.
3. Choose Add to Repository.
4. Enter a title and description for the data module in the Add to Repository dialog box.
5. Select Data Modules from the Pages combo box, and click OK.

Using the Object Repository

See also

Delphi's Object Repository provides a versatile mechanism for sharing forms, dialog boxes, and data modules across projects. It can also help with reusing similar forms in a single project, and provides project templates as starting points for new projects.

The following topics focuse on how to use the Object Repository in general as a project management tools and discusses some of the mechanics of using project templates.

See also

Object Repository dialog box

About the Object Repository

See also

Delphi provides the Object Repository as a means for sharing and reusing forms and projects. The repository itself is really just a text file that contains references to forms, projects, and experts. Details of the file format are in online Help. The Object Repository has replaced the Gallery in this version of Delphi.

Sharing across projects

By adding forms, dialog boxes, and data modules to the Object Repository, you make them available to other projects. For example, in a simple case, you could have all your projects use the same About box, copied from the Object Repository. A more advanced use of the Object Repository would be to have a standard empty dialog box with the company or product logo and standard button placement, from which all your projects derive standard-looking dialog boxes.

These sharing options are described in detail in Object Repository usage options.

Sharing within projects

The Object Repository can also help you to share items within a project, by allowing you to inherit from forms already in the project. When you open the New Items dialog box (by choosing File|New), you'll see a page tab with the name of your project. If you click that page tab, you'll see all the forms, dialog boxes, and data modules in your project. You can then derive a new item from the existing item, and customize it as needed.

For example, in a database application you might need several forms that display the same data, but which provide different command buttons. Instead of creating and maintaining several nearly-identical forms, you could lay out a generic form that contains all the data-display controls, then create separate forms that inherit the data-display layout, but have different command buttons.

By carefully planning your project forms, you can save tremendous amounts of time and effort by sharing forms within projects.

Sharing entire projects

You can also add an entire project to the Object Repository as a template for future projects. If you have a number of similar applications, for example, you can base them all on a single, standardized model.

Using experts

The Object Repository also contains references to experts, which are small applications that lead the user through a series of dialog boxes to create a form or project. Delphi provides a number of experts, and you can also add your own.

Object Repository usage options

See also

When you use an item from the Object Repository in a project you have as many as three options on how to include that item. Keep in mind that items in the Object Repository are there to be shared, and that you want to use them in ways that help, rather than hinder, reuse.

In general, you these are the three options for using Object Repository items:

- Copy the item
- Inherit from the item
- Use the item directly

Copying items from the Object Repository

The simplest sharing option is to copy an item from the Object Repository into your project. Copying makes an exact duplicate of the item as it stands and adds the copy to your project. Future changes to the item in the Object Repository will not be reflected in your copy, and alterations made to your copy will not affect the original Object Repository item.

Note: Copying is the only option available for using project templates.

Inherit from Object Repository items

The most flexible and powerful sharing option is to inherit from an item in the Object Repository. Inheriting derives a new class from the item and adds the new class to your project. When you recompile your project, any changes made to the item in the Object Repository will be reflected in your derived class, unless you have changed a particular aspect. Changes made to your derived class do not affect the shared item in the Object Repository.

Note: Inheriting is available as an option for forms, dialog boxes, and data modules, but not for project templates. It is the only option available for reusing items from within the same project.

Using Object Repository items directly

The least flexible sharing option is using an item from the Object Repository directly in your project. Using the item adds the item itself to your project, just as if you had created it as part of that project. Design-time changes made to the item therefore appear in all projects that directly use the item, as well as affecting any projects that inherit from the item.

Note: Using items directly is an available option for forms, dialog boxes, and data modules.

Items shared this way should generally be modified only at run time, to avoid making changes that affect other projects.

The Use option is the only option available for experts, whether form experts or project experts. Using an expert doesn't actually add shared code, but rather runs a process that generates its own code.

Using project templates

See also

Delphi provides project templates, pre-designed projects you can use as starting points for your own projects. Project templates are part of the Object Repository (located in the OBJREPOS subdirectory), which also provides form objects and experts.

When you start a project from a project template (other than the blank project template), Delphi prompts you for a project directory, a subdirectory in which to store the new project's files. If you specify a directory that doesn't currently exist, Delphi creates it for you. Delphi copies the template files to the project directory. You can then modify it, adding new forms and units, or use it unmodified, adding only your event-handler code. In any case, your changes affect only the open project. The original project template is unaffected and can be used again.

To start a new project from a project template,

1. Choose File|New to display the New Items dialog box.
2. Choose the Projects tab.
3. Select the project template you want and choose OK.
4. In the Select Directory dialog box, specify a directory for the new project's files.

   A copy of the project template opens in the specified directory.

Adding projects to the Object Repository

You can add your own projects and forms to those already available in the Object Repository. This is helpful in situations where you want to enforce a standard framework for programming projects throughout an organization.

For example, suppose you develop custom billing applications. You might have a generic billing application project that contains the forms and features common to all billing systems. Your business centers around adding and modifying features in this application to meet specific client requirements. In such a case, you might want to save the project containing your Generic Billing application as a project template and perhaps specify it as the default new project on your Delphi development system. Likewise, you'll probably have a particular form within this project that you want to appear as the default main or new form.

To add a project to the Object Repository,

1. If necessary, open the project you want added to the Object Repository.
2. Choose Tools|Repository.
3. On the Project Templates page, choose Add to display the Save Project Template dialog box.
4. In the Title edit box, enter a project title.

   The title for the template will appear in the Object Repository window.
5. In the Description field, enter text that describes the template.

   This text will appear in the Object Repository window's status bar.
6. In the Page field, choose the name of the page in the New Items dialog box (probably Projects) you want the template to appear on.
7. In the Author field, enter text identifying the author of the application.

   Author information appears only when the user views the repository items with full details.
8. Choose Browse to select an icon to represent this template in the Object Repository.
9. Choose OK to save the current project as a project template.

Note: If you later make changes to a project template, those changes automatically appear in new projects created from that template. They will not, however, affect projects already created from that template.

You can also save your own forms as form templates and add them to those already available in the Object Repository. This is helpful in situations where you want to develop standard forms for an organization's software, as in the earlier example.

To add a form to the Object Repository as a template, follow the preceding steps for adding projects as templates, using the Form Templates page of the Object Repository Options dialog box instead of the Project Templates page.

Customizing the Object Repository

See also

The settings in the Object Repository Options dialog box affect the behavior of Delphi when you begin a new project or create a new form in an open project. This is where you specify

- Default project
- Default new form
- Default main form

You always have to option to override these defaults by choosing File|New and selecting from the New Items dialog box.

By default, opening a new project displays a blank form. You can change this default behavior by changing Object Repository options.

Specifying the default new project

The default new project opens whenever you choose New Application from the File menu on the Delphi menu bar. If you haven't specified a default project, Delphi creates a blank project with an empty form. You can specify a project template (including a project you have created and saved as a template) as the default new project.

You can also designate a project expert to run by default when you start a new project. A project expert is a program that enables you to build a project based on your responses to a series of dialog boxes.

To specify the default new project,

1. Choose Tools|Repository to display the Object Repository dialog box.
2. Choose Projects in the Pages list.
3. Select the project object you want as the default new project from the Objects list.
4. With the object you want selected, check New Project.
5. Choose OK to register the new default setting.

Specifying the default new form

The default new form opens whenever you choose File|New Form or use the Project Manager to add a new form to an open project. If you haven't specified a default form, Delphi uses a blank form. You can specify any form template, including a form you have created and saved as a template, as the default new form. Or you can designate a form expert to run by default when a new form is added to a project.

To specify the default new form for new projects,

1. Choose Tools|Repository to display the Object Repository dialog box.
2. Choose Forms in the Pages list.
3. Select the form object you want as the default new form.
4. With the object you want selected, check New Form.
5. Choose OK to register the new default setting.

Specifying the default main form

Just as you can specify a form template or expert to be used whenever a new form is added to a project, you can also specify a form template or expert that should be used as the default main form whenever you begin a new project.

To specify the default main form for open projects,

1. Choose Tools|Repository to display the Object Repository dialog box.
2. Choose Forms in the Pages list.
3. Select the form object you want as the default main form.
4. With the object you want selected, check Main Form.
5. Choose OK to register the new default setting.

Changing defaults for new projects

See also

The Project Options dialog box contains a check box labeled Default. This control enables you to modify some of Delphi's default project configuration properties. Checking this control writes the current settings from the Compiler, Linker, and Directories/Conditionals pages of the Project Options dialog to a file called DEFPROJ.DOF. Delphi creates this file when you check the Default box and choose OK in the Project Options dialog box. Delphi then uses the project options settings stored in this file as the default for any new projects you create.

If you create a project from a template in the Object Repository that has its own options file, those settings will override the default settings in DEFPROJ.DOF.

To restore Delphi's original default settings, delete or rename the DEFPROJ.DOF file.

Note: Project options you set for an open project override the current Delphi defaults, whether those defaults are as originally shipped or as modified by you or another user.

Using the Object Repository in a shared environment

To change the location where Delphi looks for the Object Repository File, DELPHI32.DRO, create a new String Value called "BaseDir" in the “HKEY_CURRENT_USER | Software | Borland | Delphi | 2.0 | Repository” key in the Windows Registry Editor and set its Value data to the directory where this file is to be located. It is recommended that the location uses the UNC name when the DELPHI32.DRO file is to be shared.

It is suggested that Forms and Projects be saved using UNC names when they will be added to a shared Repository.

While someone is modifying the Object Repository (DELPHI32.DRO file), anyone attempting to open or add to the repository will get a dialog box with the user name of the person that has the repository open. If you are attempting to open the repository from Tools | Object Repository, the dialog box will ask if you wish to view the repository. If you choose to view the repository, you will not be allowed to save any changes.

Lock information is stored in the DELPHI32.DRL file. If this lock file cannot be opened, an exception is raised. This can mean the file is read-only or the user doesn’t have write rights for the directory. Also, if someone exits from Delphi abnormally while modifying the repository, the lock file may still contain information that the user is editing the repository and you will not be allowed to modify the DELPHI32.DRO file. In this case the lock file should be deleted.

<Library Name>is already loaded, probably as a result of an incorrect program termination. Your system may be unstable and you should exit and restart Windows now.

An error occurred while atempting to initialize Delphi's component library. One or more DLLs are already in memory, probably as a result of an incorrect program termination in a previous Delphi or BDE session.

You should exit and then restart Windows.

<IDname> is not a valid identifier

The identifier name is invalid. Ensure that the first character is a letter or an underscore (_). The characters that follow must be letters, digits, or underscores, and there cannot be any spaces in the identifier.

A field or method named <Name> already exists

The name you have specified is already being used by an existing method or field.
For a complete list of all fields and methods defined, check the form declaration at the top of the unit source file.

A component class named <Name> already exists

The component library already contains a component with the same class name you have specified.

Breakpoint is set on line that contains no code or debug information. Run anyway?

A breakpoint is set on a line that does not generate code or in a module which is not part of the project. If you choose to run anyway, invalid breakpoints will be disabled (ignored).

Could not stop due to hard mode

The integrated debugger has detected that Windows is in a modal state and will not allow the debugger to stop your application. Windows enters "hard mode" whenever processing an inter-task SendMessage, when there is no task queue, or when the menu system is active. You will not generally encounter hard mode unless you are debugging DDE or OLE processes within Delphi.

A standalone debugger such as the Turbo Debugger for Windows can be used to debug applications even when Windows is in hard mode.

Another file named <FileName> is already on the search path

A file with the same name as the one you just specified is already in another directory on the search path.

Cannot find <FileName.PAS> or <FileName.DCU> on the current search path

The .PAS or .DCU file you just specified cannot be found on the search path.
You can modify the search path, copy the file to a directory along the path, or remove the file from the list of installed units.

Cannot find implementation of method <MethodName>

The indicated method is declared in the form's class declaration but cannot be located in the implementation section of the unit. It probably has been deleted, commented out, renamed, or incorrectly modified.

Use UNDO to reverse your changes, or correct the procedure declaration manually. Be sure the declaration in the class is identical to the one in the implementation section. (This is done automatically if you use the Object Inspector to create and rename event handlers.)

For more information about the syntax of procedure declarations, see Handling events.

Debug session in progress. Terminate?

Your application is running and will be terminated if you proceed. When possible, you should cancel this dialog and terminate your application normally (for example, by selecting Close on the System Menu).

Declaration of class <ClassName> is missing or incorrect

Delphi is unable to locate the form's class declaration in the interface section of the unit. This is probably because the type declaration containing the class has been deleted, commented out, or incorrectly modified. This error will occur if Delphi cannot locate a class declaration equivalent to the following:

```
type
  ...
  TForm1 = class(TForm)
  ...
```

Use UNDO to reverse your edits, or correct the declaration manually. For more information about class declaration syntax, see Object Types.

Error address not found

The address you have specified cannot be mapped to a source code position. This error usually occurs for one of the following reasons:

- The address you entered is invalid or is not an address in your application.
- The module containing the specified address was not compiled with debug information.
- The address specified does not correspond to a program statement.

Note that the runtime and visual component libraries are compiled without debug information.

Error creating process: <Process> (<ErrorCode>)

Delphi was unable to start your application for the reason specified.
For more information about "Insufficient memory to run" errors, see README.TXT.

Field <Field Name> does not have a corresponding component. Remove the declaration?

The first section of your form's class declaration defines a field for which there is no corresponding component on the form. Note that this section is reserved for use by the form designer.

To declare your own fields and methods, place them in a separate public, private, or porotected section.

This error will also occur if you load the binary form file (.DFM) into the Code Editor and delete or rename one or more components.

Field <Field Name> should be of type <Type1> but is declared as <Type2>. Correct the declaration?

The type of specified field does not match its corresponding component on the form. This error will occur if you change the field declaration in the Code Editor or load the binary form file (.DFM) into the Code Editor and modify the type of a component.

If you select No and run your application, an error will occur when the form is loaded.

IMPLEMENTATION part is missing or incorrect

In order to keep your form and source code synchronized, Delphi must be able to find the unit's implementation section. This reserved word has been deleted, commented out, or misspelled.

Use UNDO to reverse your changes or correct the reserved word manually. For more information about unit syntax, see the reserved word Unit.

Incorrect field declaration in class <ClassName>

In order to keep your form and source code synchronized, Delphi must be able to find and maintain the declaration of each field in the first section of the form's class definition. Though the compiler allows more complex syntax, the form designer will report an error unless each field that is declared in this section is equivalent to the following:

```
type
  ...
  TForm1 = class(TForm)
  Field1:FieldType;
  Field2:FieldType;
  ...
```

This error has occurred because one or more declarations in this section have been deleted, commented out, or incorrectly modified. Use UNDO to reverse your changes or correct the declaration manually.

Note that this first section of the form's class declaration is reserved for use by the form designer. To declare your own fields and methods, place them in a separate public, private, or protected section.

Incorrect method declaration in class <ClassName>

In order to keep your form and source code synchronized, Delphi must be able to find and maintain the declaration of each method in the first section of the form's class definition. The form designer will report an error unless the field and method declarations in this section are equivalent to the following:

```
type
  ...
  TForm1 = class(TForm)
  Field1:FieldType;
  Field2:FieldType;
  ...
  <Method1 Declaration>;
  <Method2 Declaration>;
  ...
...
```

This error has occurred because one or more method declarations in this section have been deleted, commented out, or incorrectly modified. Use UNDO to reserve your changes or correct the declaration manually.

Note that this first section of the form's class declaration is reserved for use by the form designer. To declare your own fields and methods, place them in a separate public, private, or protected section.

Insufficient memory to run

Delphi was unable to run your application due to insufficient memory or Windows resources. Close other Windows applications and try again.

This error sometimes occurs because of insufficient low (conventional) memory. For further information, see README.TXT.

Invalid event profile <Name>

The VBX control you are installing is invalid.

Module header is missing or incorrect

The module header has been deleted, commented out, or otherwise incorrectly modified. Use UNDO to reverse your changes, or correct the declaration manually.

In order to keep your form and source code synchronized, Delphi must be able to find a valid module header at the beginning of the source file. A valid module header consists of the reserved word unit, program or library, followed by an identifier (for example, Unit1, Project1), followed by a semi-colon. The file name must match the identifier.

For example, Delphi will look for a unit named Unit1 in UNIT1.PAS, a project named Project1 in PROJECT1.DPR, and a library (.DLL) named MyDLL in MYDLL.DPR.

Note that module identifiers cannot exceed eight characters in length.

No code was generated for the current line

You are attempting to run to the cursor position, but you have specified a line that did not generate code, or is in a module which is not part of the project.

Specify another line and try again.

Note that the smart linker will remove procedures that are declared but not called by the program (unless they are virtual method of an object that is linked in).

Property and method <MethodName> are not compatible

You are assigning a method to an event property even though they have incompatible parameter lists. Parameter lists are incompatible if the number of types of parameters are not identical. For a list of compatible methods in this form, see the dropdown list on the Object Inspector Events page.

Source has been modified. Rebuild?

You have made changes to one or more source or form modules while your application is running. When possible, you should terminate your application normally (select No, switch to your running application, and select Close on the System Menu), and then run or compile again.

If you select Yes, your application will be terminated and then recompiled.

Symbol <BrowseSymbol> not found.

The browser cannot find the specified symbol. This error will occur if you enter an invalid symbol name or if debug information is not available for the module that contains the specified symbol.

The <Method Name> method referenced by <Form Name> does not exist. Remove the reference?

The indicated method is no longer present in the class declaration of the form. This error occurs when you manually delete or rename a method in the form's class declaration that is assigned to an event property.

If you select No and run this application, an error will occur when the form is loaded.

The <Method Name> method referenced by <Form Name>.<Event Name> has an incompatible parameter list. Remove the reference?

A form has been loaded that contains an event property mapped to a method with an incompatible parameter list. Parameter lists are incompatible if the number or types of parameters are not identical.

For a list of methods declared in this form which are compatible for this event property, use the dropdown list on the Object Inspector's Events page.

This error occurs when hyou manually modify a method declaration that is referenced by an event property.

Note that it is unsafe to run this program without removing the reference or correcting the error.

The project already contains a form or module named <Name>

Every module name (program or library, form and unit) in a project must be unique.

USES clause is missing or incorrect

In order to keep your forms and source code synchronized, Delphi must be able to find and maintain the USES clause of each module.

In a unit, a valid USES clause must be present immediately following the interface reserved word. In a program or library, a valid USES clause must be present immediately following the program or library header.

This error occurs because the USES clause has been deleted, commented out, or incorrectly modified. Use UNDO to reverse your changes or correct thte declaration manually. For more information about the USES clause syntax, see the reserved word USES.

Features chart

The following table summarizes the tools and features in Delphi 2.0. Available tools and features differ depending on Delphi 2.0 version. All features are available in the Delphi Client/Server Suite.

CS=Client/Server Suite DB=Developer DT=Desktop

Tool/Feature	Description	C/S	DB	DT
SQL Explorer	SQL-enabled integrated tool for browsing databases, managing BDE aliases, and creating data dictionaries.	3		
Database Explorer	Integrated tool for browsing databases, managing BDE aliases, and creating data dictionaries.		3	3
SQL Monitor	Integrated tool for tracing and examining SQL query performance.	3		
Visual Query Builder	Integrated tool for visual building of SQL queries.	3		
Data Pump Expert	Standalone tool for moving metadata and data between databases	3		
Data Dictionary	Stores extended field attributes apart from application code; shares attributes across fields, datasets, and applications.	3	3	
Object Repository	Stores data modules and forms for sharing among applications.	3	3	3
Data modules	Non-visual component containers for centralized data-access across forms and applications.	3	3	3
Data-access components	Non-visual components for encapsulating database connections.	3	3	3
Data-aware controls	Visual components for providing a user interface to data.	3	3	3
Database Desktop (DBD)	Tool for browsing, creating, and changing desktop databases.	3	3	3
Borland Database Engine (BDE)	Borland's core database engine and connectivity software.	3	3	3
SQL Links	BDE SQL drivers for Sybase, Microsoft SQL-Server, Oracle, and InterBase	3		
ODBC socket	BDE support socket for third-party ODBC drivers.	3	3	
BDE API	BDE application programming interface and online help files.	3	3	
ReportSmith SQL Edition	Borland's SQL-enabled, live-link database report-writing application.	3		
ReportSmith	Borland's standard desktop database report-writing application.		3	
Quick Reports	Delphi components for creating pre-defined reports in an application.	3	3	3
InterBase NT	Borland InterBase Workgroup Server for NT, two-user license.	3		
Local InterBase Server	32-bit Local InterBase Server.	3	3	

Using the object repository with data modules

The new Object Repository stores links to data modules, forms, and projects for reuse and reference. When you create a new application, you can:

- *Copy* an existing data module, form, or project, ensuring that your copy is completely independent of the repository.
- *Inherit* an existing data module, form, or project, ensuring that changes to the linked module, form, or project in the repository are replicated to your application when you recompile.
- *Use* an existing data module, form, or project, ensuring that changes you make to the module, form, or project are available for use in other applications.

The Object Repository supports team development practices. It uses a referencing mechanism to data modules, forms, and projects where they exist on a network server or shared machine. Every developer in your organization can save objects to a shared location, and then set Delphi's Object Repository reference to point to that location.

The Object Repository is also customizable through the Tools|Repository menu in the IDE.

Using the data dictionary

The new Data Dictionary provides a customizable storage area, independent of your applications, where you can create extended field attribute sets that describe the content and appearance of data.

For example, if you frequently develop financial applications, you may create a number of specialized field attribute sets describing different display formats for currency. When you create datasets for your application at design time, rather than using the Object Inspector to set the currency fields in each dataset by hand, you can associate those fields with extended fields attribute set in the Data Dictionary. Using the Data Dictionary also ensures a consistent data appearance within and across the applications you create.

In a client/server environment the Data Dictionary can reside on a remote server for additional sharing of information.

To learn how to create extended field attribute sets from the Fields editor at design time, and how to associate them with fields throughout the datasets in your application, see Using the Fields Editor. To learn more about creating a Data Dictionary and extended field attributes with the SQL and Database Explorers, see their respective online help files.

Using the SQL/Database Explorer

In Delphi Client/Server Suite you browse databases and populate tables with data using the new SQL Explorer from the IDE. In Delphi Developer you use the new Database Explorer. Both versions of the Explorer enable you to:

- Examine existing database tables and structures. The SQL Explorer enables you to examine remote SQL databases, and to query them.
- Populate tables with data.
- Create extended field attribute sets in a Data Dictionary for later retrieval and reuse. Extended field attributes describe how values in a field are formatted and displayed.
- Associate extended field attributes with fields in your application.
- Create and manage BDE aliases, used by your application to connect to databases.

To learn more about the SQL Explorer, and the Database Explorer see their respective online help files.

Using the SQL Monitor

In Delphi Client/Server Suite you can use the SQL Monitor to trace and time calls between your client application and a remote SQL database server. Timing information is useful for optimizing SQL statements and transactions. A series of options enables you to trace:

- Prepared query statements
- Executed query statements
- Statement operations
- Connect/Disconnect
- Transactions
- Blob I/O
- Vendor errors
- Vendor API calls

You can use the SQL Monitor to examine how optimally SQL statements in an application are performed, to see the SQL statements generated by the Borland Database Engine (BDE), to see if the database client libraries are functioning properly, and to see if the database server is executing a run-away query. The SQL Monitor also enables you to save and print session logs for further reference and comparison.

Using the Visual Query Builder

In Delphi Client/Server Suite you can enter SQL statements directly in SQL and Update SQL property editors, or you can invoke the Visual Query Builder to construct a query based on a visible representation of tables and fields in a database.

To use the Visual Query Builder, select a query component, right-click it to invoke the SpeedMenu, and choose Query Builder.

Data access component summary

Delphi provides non-visual data-access components that encapsulate your client's communication with a database. Data-access components only deal with database connectivity, which enables you to focus your attention on your application's data needs without worrying about user interface. Typically these components are placed in a data module container in an application.

The following table summarizes the components that appear on the Data Access page of the Component palette, and where in the *Database Application Developer's Guide* you can get more information about them:

Component	Purpose
TDataSource	Acts as a conduit between other data access components and data-aware visual controls. For more information about TDataSource, see Chapter 6, "Working with data sources."
TTable	Represents a dataset that retrieves all columns and records from a database table. For more information about TTable, see Chapter 8, "Working with tables."
TQuery	Represents a dataset that retrieves a subset of columns and records from one or more local or remote database tables based on an SQL query. For more information about TQuery, see Chapter 9, "Working with queries."
TStoredProc	Represents a dataset that retrieves one or more records from a database table based on a stored procedure defined for a database server. For more information about TStoredProc, see Chapter 10, "Working with stored procedures."
TDatabase	Encapsulates a client/server connection to a single database in one session. For more information about TDatabase, see Chapter 3, "Connecting to databases."
TSession	Represents a single session in a multi-threaded database application. Each session can have multiple database connections as long as each thread associated with a particular database has its own session. For more information about TSession, see Chapter 3, "Connecting to databases."
TBatchMove	Encapsulates a dataset used to move data from one table to another en masse. For more information about TBatchMove, see Chapter 16, "Working with TBatchMove."
TReport	Encapsulates a ReportSmith report in an application. For more information about TReport, see Chapter 14, "Using reports."
TUpdateSQL	Represents SQL INSERT, UPDATE, and DELETE statements that can be used to update the read-only result sets of some queries. For more information about TUpdateSQL, see Chapter 15, "Working with cached updates."

Data controls summary

Delphi provides data-aware visual components, called visual controls, that encapsulate user interaction with your application's data sources. You design a user interface with these controls. You place these controls on the forms in your application. Each control is linked to one or more fields or records, and controls how a user sees a field or record in your application.

You link visual controls to data-access components through a data source component. A data-source component acts as a conduit between an application's low-level data-access interactions and the high-level view of data its users are provided.

The following table summarizes the data-aware visual controls that appear on the Data Controls page:

Component	Purpose
TDBGrid	Display and edit dataset records in tabular format.
TDBNavigator	Cursor through dataset records; enable Edit and Insert states; post new or modified records; cancel edit mode; refresh data display.
TDBText	Display a field as a label.
TDBEdit	Display and edit a field in an edit box.
TDBMemo	Display and edit multi-line or blob text in a scrollable, multi-line edit box.
TDBImage	Display and edit a graphics image or binary blob data.
TDBListBox	Display a list of choices for entry in a field.
TDBComboBox	Display an edit box and drop-down list of choices for edit and entry in a field.
TDBCheckBox	Display and set a Boolean field condition in a check box.
TDBRadioGroup	Display and set exclusive choices for a field in a radio button group.
TDBLookupListBox	Display a list of choices derived from a field in another dataset for entry into a field.
TDBLookupComboBox	Display an edit box and drop-down list of choices derived from a field in another dataset for edit and entry in a field.
TDBCtrlGrid	(Client/Server Suite and Developer only). Display and edit records in a tabular grid, where each cell in the grid contains a repeating set of data-aware components and one record.

Using TTable

A table component is the most fundamental and flexible dataset component class in Delphi. It gives you access to every row and column in an underlying database table, whether it is from Paradox, dBASE, an ODBC-compliant database such as Microsoft Access, or an SQL database on a remote server, such as InterBase, Sybase, or SQL Server.

You can view and edit data in every column and row of a table, you can work with a range of rows in a table, and you can filter records to retrieve a subset of all records in a table based on filter criteria you specify.

Creating a table component

To create a table component:

1. Place a table component from the Data Access page of the Component palette in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
2. Set the *DatabaseName* of the component to the name of the database to access.
3. Set the *TableName* property to the name of the table in the database. You can select tables from the drop-down list if the *DatabaseName* property is already specified.
4. Place a data source component in the data module or on the form, and set its *DataSet* property to the name of the table component. The data source component is used to pass a result set from the table to data-aware components for display.

Note: These are general steps for creating a table component. There are additional properties you may need to set because of application requirements.

Searching on a table

You can search a table for specific records using the generic search methods Locate and Lookup. These methods enable you to search on any type of columns in any table, whether or not they are indexed or keyed.

Table components also continue to support Goto and Find methods for compatibility with previous versions of Delphi. While these methods are documented here to allow you to work with legacy applications, you should always use Lookup and Locate in your new applications. Moreover, you may see performance gains in existing applications if you convert them to use the new method.

Using Locate

Locate moves the cursor to the first row matching a specified set of search criteria. In its simplest form, you pass Locate the name of a column to search, a field value to match, and an options flag specifying whether the search is case-insensitive or if it can use partial-key matching. For example, the following code moves the cursor to the first row in the CustTable where the value in the Company column is "Professional Divers, Ltd.":

```
var
  LocateSuccess: Boolean;
  SearchOptions: TLocateOptions;
begin
  SearchOptions := [loPartialKey];
  LocateSuccess := CustTable.Locate('Company', 'Professional Divers, Ltd.',
      SearchOptions);
end;
```

If Locate finds a match, the first record containing the match becomes the current record. Locate returns True if it finds a matching record, False if it does not. If a search fails, the current record does not change.

The real power of Locate comes into play when you want to search on multiple columns and specify multiple values to search for. Search values are variants, which enables you to specify different data types in your search criteria. To specify multiple columns in a search string, separate individual items in the string with semi-colons.

Because search values are variants, if you pass multiple values, you must either pass a variant array type as an argument (for example, the return values from the Lookup method), or you must construct the variant array on the fly using the VarArrayOf function. The following code illustrates a search on multiple columns using multiple search values and partial-key matching:

```
with CustTable do
  Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver','P']), loPartialKey);
```

Locate uses the fastest possible method to locate matching records. If the columns to search are indexed and the index is compatible with the search options you specify, Locate uses the index. Otherwise, Locate creates a BDE filter for the search.

Using Lookup

Lookup searches for the first row that matches specified search criteria. If it finds a matching row, it forces the recalculation of any calculated fields and lookup fields associated with the dataset, then returns one or more fields from the matching row. Lookup does not move the cursor to the matching row; it only returns values from it.

In its simplest form, you pass Lookup the name of column to search, the field value to match, and the field or fields to return. For example, the following code looks for the first row in the CustTable where the value in the Company column is "Professional Divers, Ltd.", and returns the company name, a contact person, and a phone number for the company:

```
var
  LookupResults: Variant;
begin
with CustTable do
  LookupResults := Lookup('Company', 'Professional Divers, Ltd.', 'Company;
      Contact; Phone');
end;
```

Lookup returns values for the specified fields from the first matching record it finds. Values are returned as variants. If more than one return value is requested, Lookup returns a variant array. If there are no matching records, Lookup returns a Null variant. For more information about variant arrays, see the Object Pascal Language Guide.

The real power of Lookup comes into play when you want to search on multiple columns and specify multiple values to search for. To specify strings containing multiple columns or result fields, separate individual in the string items with semi-colons.

Because search values are variants, if you pass multiple values, you must either pass a variant array type as an argument (for example, the return values from the Lookup method), or you must construct the variant array on the fly using the VarArrayOf function. The following code illustrates a lookup search on multiple columns:

```
var
  LookupResults: Variant;
begin
with CustTable do
  LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver', 'Christiansted']),
      'Company; Addr1; Addr2; State; Zip');
end;
```

Lookup uses the fastest possible method to locate matching records. If the columns to search are indexed Lookup uses the index. Otherwise, Lookup creates a BDE filter for the search.

Using TQuery

A query component encapsulates an SQL statement that returns a set of related rows and columns from one or more database tables. SQL (pronounced "ess-cue-ell") is an industry-standard relational database language that is used by most remote server-based database vendors, such as Sybase, Oracle, InterBase, and SQL Server.

Delphi query components can be used with remote database servers (Delphi Client/ Server Suite only), with Paradox and dBASE, and with OBDC-compliant databases such as Microsoft Access and FoxPro.

A query component differs from a table component in two significant ways. It can:

- Access more than one table at a time (called a "join" in SQL).
- Automatically access a subset of rows and columns in its underlying table(s), rather than always returning all rows and columns and requiring you to set ranges and filters to restrict row access.

The data returned by a query component depends on the selection criteria you specify in the component's SQL property at design time or at run time. The selection criteria is in the form of an SQL statement, and can be

- *Static*, where all parameters in the statement are specified at design time, or
- *Dynamic*, where some or all of a statement's parameters are set at run time.

In addition to encapsulating standard SQL features, query components also offer a powerful ability not found on most remote database servers: the ability to create "heterogeneous joins," queries against more than one server or table type.

Choosing between query and table components

Table components are full-featured, flexible, and easily used data access components that are sufficient for many database applications.

If you are a desktop client/server developer, you are familiar with the table component paradigm, and know its strengths. Query components offer different capabilities from table components, and are useful when you need to:

- Query data in multiple tables.
- Restrict data access to a subset of rows and columns across tables.
- Write applications that require SQL syntax for compatibility with other SQL applications.

If you are a client/server developer used to working with remote databases, you are familiar with SQL, and know the strength in query-based data access. On the other hand, table components may occasionally offer you data access alternatives, such as ranges, that make them attractive.

Creating a query component

To create a query component:

1. Place a query component from the Data Access page of the Component palette in a data module, and set its *Name* property appropriately for your application.
2. Set the *DatabaseName* property of the component to the name of the database to query. *DatabaseName* can be a Borland Database Engine (BDE) alias, or an explicit directory path and database name. If your application uses database components, *DatabaseName* can be set to a local alias defined for the database component instead.
3. Specify an SQL statement in the *SQL* property of the component, and optionally specify any parameters for the statement in the *Params* property. For more information, see "Setting the SQL property at design time" on page 111.
4. Place a data source component in the data module, and set its *DataSet* property to the name of the query component. The data source module is used to return a result set from the query to data-aware components for display.

Note: These are the general steps for creating a query component. Depending on the type of query you create (static or dynamic), there may be additional properties you need to set before executing a query.

Using TStoredProc

A stored procedure component encapsulates a stored procedure in a database on a remote server. A stored procedure is a set of statements, stored as part of a remote server's database metadata (like tables, indexes, and domains). A stored procedure performs a frequently-repeated database-related task on the server and passes results to a client application, such as a Delphi database application. The stored procedure component enables Delphi applications to execute server stored procedures.

Often, operations that act upon large numbers of rows in database tables—or that use aggregate or mathematical functions—are candidates for stored procedures. By moving these repetitive and calculation-intensive tasks to the server, you improve the performance of your database application by:

- Taking advantage of the server's usually greater processing power and speed.
- Reducing the amount of network traffic since the processing takes place on the server where the data resides.

For example, consider an application that needs to compute a single value: the standard deviation of values over a large number of records. To perform this function in your Delphi application, all the values used in the computation must be fetched from the server, resulting in increased network traffic. Then your application must perform the computation. Because all you want in your application is the end result—a single value representing the standard deviation—it would be far more efficient for a stored procedure on the server to read the data stored there, perform the calculation, and pass your application the single value it requires.

See your server's database documentation for more information about its support for stored procedures.

Creating a stored procedure component

To create a stored procedure component for a stored procedure on a database server:

1. Place a stored procedure component from the Data Access page of the Component palette in a data module.
2. Set the *DatabaseName* property of the stored procedure component to the name of the database in which the stored procedure is defined. *DatabaseName* must be a BDE alias.
3. Set the *StoredProcName* property to the name of the stored procedure to use, or select its name from the drop-down list for the property.
4. Specify input parameters, if any, in the *Params* property. You can use the Parameters editor to set input parameters.

Note: The Parameters editor also lets you see the output parameters used by the procedure to return results to your application.

Setting input parameters and viewing output and result parameters

Many stored procedures require you to pass them a series of input arguments, or parameters, to specify what and how to process. You specify input parameters in the Params property. The order in which you specify input parameters is significant, and is determined by the stored procedure definition on the server.

At design time, the easiest and safest way to enter input parameters is to invoke the StoredProc Parameters editor. The StoreProc Parameters editor lists input parameters in the correct order, and lets you assign values to them.

To invoke the StoredProc Parameters editor:

1. Select the stored procedure component.
2. Right-click the component to invoke the SpeedMenu.
3. Choose Define Parameters.

The Parameter name list box displays all input, output, and result parameters for the procedure. Information on input and output parameters is retrieved from the server. For some servers (such as Sybase), parameter information may not be accessible. In this case, the Parameter name list box is empty, and you must add the names and order of input and output parameters in order to use the procedure.

The Parameter type combo box describes whether a parameter selected in the list box is an input, output, or results parameter. If a server's stored procedure allows it, a single parameter may be both an input and output. If you add a parameter to the list, you must set the parameter type for it.

The Data type combo box lists the data type for a parameter selected in the list box. If you add a parameter to the list, you must set its data type.

The Value edit box enables you to enter a value for a selected input parameter, and the Null Value check box enables you set a Null value for the selected input parameter if its data type supports Null values.

The Add button enables you to add parameters to a stored procedure definition when your server does not pass the information to Delphi. The Delete button enables you to remove parameters you have added, and the Clear button removes all parameters from the list. Do not add, delete, or clear parameters for servers that pass parameter information to Delphi except if you are working with Oracle overloaded stored procedures.

Note: Defining parameters at design time also prepares the stored procedure for execution. A stored procedure must be prepared before it can be executed at run time.

To signal the end of parameter definition, choose OK.

Creating parameters and parameter values at run time

To create parameters at run time, access the Params property directly. Params is an array of parameter strings. For example, the following code assigns the text of an edit box to the first string in the array:

```
StoredProc1.Params[0].AsString:= Edit1.Text;
```

You can also access parameters by name using the ParamsByName method:

```
StoredProc1.ParamsByName('Company').AsString := Edit1.Text;
```

Preparing and executing a stored procedure

To use a stored procedure, you must prepare it and execute it. You can prepare a stored procedure at:

- Design time, by choosing OK in the Parameters editor.
- Run time, by calling the *Prepare* method of the stored procedure component.

  For example, the following code prepares a stored procedure for execution:

  ```
  StoredProc1.Prepare;
  ```

Note: You can prepare a stored procedure both at run time and at design time. In fact, if your application changes parameter information at run time, such as when using Oracle overloaded procedures, you should prepare the procedure again.

To execute a prepared stored procedure, call the ExecProc method for the stored procedure component. The following code illustrates code that prepares and executes a stored procedure:

```
StoredProc1.Params[0].AsString := Edit1.Text;
StoredProc1.Prepare;
StoredProc1.ExecProc;
```

When you execute a stored procedure, it returns either output parameters or a result set. There are two possible return types: singleton returns, which return a single value or set of values, and result sets, which return many values, much like a query.

Understanding output parameters and result sets

A stored procedure returns values in an array of output parameters. Return values can be either asingleton result or a result set with a cursor, if the server supports it.

To access a stored procedure's output parameters at run time, you can index into the Params string list, or you can use the ParamByName method to access the values. The following statement both set the value of a edit box based on output parameters:

```
Edit1.Text := StoredProc1.Params[6].AsString;
Edit1.Text : = StoredProc1.ParamByName('Contact').AsString;
```

Note: If a stored procedure returns a result set, you may find it more useful to access and display return values in standard data-aware controls.

Working with result sets

On some servers, such as Sybase, stored procedures can return a result set similar to a query. Applications can use data aware controls to display the output of such stored procedures.

To display the results from a stored procedure in data-aware controls:

1. Place a datasource component on the data module.
2. Set the *DataSet* property of the datasource to the name of the stored procedure component from which to receive data.
3. Set the *DataSource* properties of the data-aware controls to the name of the datasource component.

The data-aware controls can now display the results from a stored procedure when the Active property for the stored procedure component is True.

Using TBatchMove

The TBatchMove component enables you to copy, append, delete, and update tables and data, and is most often used to:

- Download data from a server to a local data source for analysis or other operations.
- Move a desktop database into tables on a remote server as part of an upsizing operation.

A batch move component can create tables on the destination that correspond to the source tables, automatically mapping the column names and data types as appropriate. It inherits many of its fundamental properties and methods from TDataSet.

Creating a batch move component

To create a batch move component:

1. Place a *TBatchMove* component from the Data Access page of the Component palette in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
2. Set the *Source* property of the component to the name of the table to copy, append, or update from. You can select tables from the drop-down list of available table components.
3. Set the *Destination* property to the name of the table to create, append to, or update. If you are appending, updating, or deleting, *Destination* must be the name of an existing table. You can select tables from the drop-down list of available table components in these cases.
4. Set the *Mode* property to indicate the type of operation to perform. Valid operations are *batAppend* (the default), *batUpdate*, *batAppendUpdate*, *batCopy*, and *batDelete*. For more information about these modes, see "Specifying a batch move mode" on page 178.
5. Optionally set column mappings using the *Mappings* property. You need not set this property if you want batch move to match column based on their position in the source and destination tables.
6. Optionally specify the *ChangedTableName*, *KeyViolTableName*, and *ProblemTableName* properties. Batch move stores problem records it encounters during the batch move operation in the table specified by *ProblemTableName*. If you are updating a Paradox table through a batch move, key violations can be reported in the table you specify in *KeyViolTableName*. *ChangedTableName* lists all records that changed in the destination table as a result of the batch move operation. If you do not specify these properties, these error tables are not created or used.

Using Filters

An application is frequently interested in only a subset of records within a dataset. For example, you may be interested in retrieving or viewing only those records for companies based in California in your customer database, or you may want to find a record that contains a particular set of field values. Delphi supports filtering of a table or query to handle both of these requirements. Using filters you can temporarily restrict an application's view of data.

There are two ways to filter a dataset:

- Restricting record visibility at the time of record retrieval using an *OnFilterRecord* event handler.
- Finding a record in a dataset that matches search values using the *Locate* method for the dataset. If you use *Locate*,

Delphi automatically generates a filter for you at run time if it needs to.

Note: Filters are applied to every record retrieved in a dataset. When you want to filter large volumes of data, it may be more efficient to use a query to restrict record retrieval, or to set a range on an indexed table rather than using filters.

Using the OnFilterRecord event handler

To restrict visible records in a dataset to a subset of those that match a certain set of criteria, follow these steps:

1. Write an *OnFilterRecord* event handler for the dataset.
2. Set the *Filtered* property for the dataset to *True*.

When Filtered is True, a dataset generates an OnFilterRecord event for each record it fetches. The event handler for the OnFilterRecord tests each record, and only those that meet the Filter's conditions are visible in the application.

Note: When filtering is on, user edits to a record may mean that the record no longer meets a filter's test conditions. The next time the record is retrieved from the dataset, it may therefore "disappear." If that happens, the next record that passes the filter condition becomes the current record.

Turning filters on and off at run time

At run time you can turn filtering off by setting Filtered to False, and you can turn it on by setting Filtered to True again. If you turn off filtering, all records in a dataset are available to your application, but in order for the application to see all records, you must also call the dataset's Refresh method. For example:

```
with Table1 do
begin
  Filtered := False;
  Refresh;
end;
```

If you turn on filtering at run time, you must also call Refresh again to make the filter take effect. The current record may no longer pass the filter condition, and may disappear. If that happens, the next record that passes the filter condition becomes the current record.

Switching filter event handlers at run time

You can change how records are filtered at run time by assigning a different handler to the OnFilterRecord event for a dataset. To apply the new filter to the dataset after changing it, call the dataset's Refresh method:

```
Table1.OnFilterRecord := Query2FilterRecord;
Refresh;
```

Writing an OnFilterRecord event handler

A filter for a dataset is an event handler that responds to OnFilterRecord events generated by the dataset for each record it retrieves. At the heart of every filter handler is a test that determines if a record should be included in those that are visible to the application.

To indicate whether a record passes the filter condition, your filter handler must set an Accept parameter to True to include a record, or False to exclude it. For example, the following filter displays only those records with the State field set to “CA”:

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
  Accept := DataSet['State'] = 'CA';
end;
```

Using TDatabase

A database component encapsulates the connection to a single database within an application. If you do not need to control database connections (such as specifying a transaction isolation level), do not create database components. Temporary database components are created automatically at run time when an application attempts to open a table for which there is not already a database component. But if you want to control the persistence of database connections, logins to a database server, property values of database aliases, or transactions, then you must create a database component for each desired connection.

Creating a database component

The Data Access page of the Component palette contains a database component you can place in a data module or form. The main advantages to creating a database component at design time are that you can set its initial properties and write OnLogin events for it. OnLogin offers you a chance to customize the handling of security on a database server when a database component first attaches the server.

Logging into a server

You can control server login with a database component's LoginPrompt property and the its OnLogin method. Controlling login is essential to server security, and may be required by your remote server.

LoginPrompt specifies whether your users are prompted to log in to a database server the first time your application attempts to connect to a database requiring a login. If True (the default), your application displays a standard Login dialog box. The Login dialog box prompts for a user name and password. A mask symbol (an asterisk by default) displays for each character entered in the Password edit box.

If you set LoginPrompt to False, the standard Login dialog box is not displayed at run time when an application attempts to connect to a database server that requires a login. Instead, your application must provide the user name and password, either through the Params property or the OnLogin event for the database component. At design time, the standard Login dialog box appears when you first connect to a remote database server if LoginPrompt is False and the Params property does not contain user name and password entries.

Note: Storing hard-coded user name and password entries in the *Params* property or in code for an *OnLogin* event can compromise server security.

The Params property is a string list containing the database connection parameters for the BDE alias associated with a database component. Some typical connection parameters include path statement, server name, schema caching size, language driver, and SQL query mode.

At design time you can create or edit connection parameters in three ways:

- Use the Database Explorer or BDE Configuration utility to create or modify BDE aliases, including parameters. For more information about these utilities, see their online Help files.
- Double-click the *Params* property in the Object Inspector to invoke the String List editor. To learn more about the String List editor, see the Delphi User' s Guide.
- Double-click a database component in a data module or form to invoke the Database Properties editor.

When you first invoke the Database Properties editor, the parameters for the BDE alias are not visible. To see the current settings, click Defaults. The current parameters are displayed in the Parameter overrides memo box. You can edit existing entries or add new ones. To clear existing parameters, click Clear. Changes you make take effect only when you click OK.

Handling transactions

There are two ways to manage transactions in a Delphi database application using implicit and explicit control. It especially focuses on:

- Using the methods and properties of the *TDatabase* component that enable explicit transaction control.
- Using passthrough SQL with *TQuery* components to control transactions.

Delphi also supports local transactions, transactions made against local Paradox and dBASE tables.

Understanding transactions

When you create a database application using Delphi, Delphi provides transaction control for all database access, even against local Paradox and dBASE tables. A transaction is a group of actions that must all be carried out successfully on one or more tables in a database before they are committed (made permanent). If one of the actions in the group fails, then all the actions fail.

Transactions ensure database consistency even if there are hardware failures. It also maintains the integrity of data while permitting concurrent multiuser access.

For example, an application might update the ORDERS table to indicate that an order for a certain item was taken, and then update the INVENTORY table to reflect the reduction in inventory available. If there were a hardware failure after the first update but before the second, the database would be in an inconsistent state because the inventory would not reflect the order entered. Under transaction control, both statements would be committed at the same time. If one statement failed, then both would be undone (rolled back).

Using implicit transactions

By default, Delphi provides implicit transaction control for your applications through the Borland Database Engine (BDE). When an application is under implicit transaction control, Delphi uses a separate transaction for each record in a dataset that is written to the underlying database. It commits each individual write operation, such as Post and AppendRecord.

Using implicit transaction control is easy. It guarantees both a minimum of record update conflicts and a consistent view of the database. On the other hand, because each row of data written to a database takes place in its own transaction, implicit transaction control can lead to excessive network traffic and slower application performance.

If you explicitly control transactions, you can choose the most effective times to start, commit, and roll back your transactions. When you develop client applications in a multi-user environment, particularly when your applications run against a remote SQL server, such as Sybase, Oracle, Microsoft SQL, or InterBase,® or remote ODBC-compliant databases such as Access and FoxPro, you should control transactions explicitly.

Note: If you used cached updating, you may be able to minimize the number of transactions you need to use in your applications. For more information about cached updates, see Chapter 15, "Working with cached updates," in the *Database Application Developer's Guide*.

Working with explicit transaction control

There are two mutually exclusive ways to control transactions explicitly in a Delphi database application:

- Use the methods and properties of the *TDatabase* component.
- Use passthrough SQL in a *TQuery* component. Passthrough SQL is only meaningful in the Delphi Client/Server Suite,

where you use SQL Links to pass SQL statements directly to remote SQL or ODBC servers.

The main advantage to using the methods and properties of a database component to control transactions is that it provides a clean, portable application that is not dependent on a particular database or server.

The main advantage to passthrough SQL is that you can use the advanced transaction management capabilities of a particular database server, such as schema caching. To understand the advantages of your server's transaction management model, see your database server documentation.

Using TDatabase methods and properties

The following table lists the methods and properties of TDatabase that are specific to transaction management, and describes how they are used:

Method or property	Purpose
Commit	Commits data changes and ends the transaction.
Rollback	Undoes data changes and ends the transaction.
StartTransaction	Starts a transaction.
TransIsolation	Specifies the transaction isolation level for a transaction.

StartTransaction, Commit, and Rollback are methods your application can call at run time to start transactions, control the duration of transactions, and save or discard changes made to the database.

TransIsolation is a database component property that enables you to control how a transaction interacts with other simultaneous transactions when they work with the same tables. In particular, it affects how much a transaction "sees" of other transactions' changes to a table.

Starting a transaction

When you start a transaction, all subsequent statements that read from and write to the database occur in the context of that transaction. Each statement is considered part of a group. Write changes made by any statement in the group must be

successfully committed to the database, or every change made by every write statement made in the group are undone.

Grouping statements is useful when statements are dependent upon one another. Consider a bank transaction at an Automated Teller Machine (ATM). When a customer decides to transfer money from a savings account to a checking account, two changes must take place in the bank's database records:

- The savings account must be debited.
- The checking account must be credited.

If, for any reason, one of these actions cannot be completed, then neither action should take place. Because these actions are so closely related, they should take place within a single transaction.

To start a transaction in a Delphi application, call a database component's StartTransaction method:

```
DatabaseInterBase.StartTransaction;
```

All subsequent database actions take place in the context of the newly started transaction until the transaction is explicitly terminated by a subsequent call to Commit or Rollback.

How long should you keep a transaction going? Ideally, only as long as necessary. The longer a transaction is active, the more simultaneous users that access the database, and the more concurrent, simultaneous transactions that start and end during the lifetime of your transaction, the greater the likelihood that your transaction will conflict with another when you attempt to commit your changes.

Committing a transaction

To make database changes permanent, a transaction must be committed using a database component's Commit method. Executing a commit statement saves database changes and ends the transaction. For example, the following statement ends the transaction started in the previous code example:

```
DatabaseInterBase.Commit;
```

Note: *Commit* should always be attempted in a **try...except** statement. If a transaction cannot commit successfully, you can attempt to handle the error, and perhaps retry the operation.

Rolling back a transaction

To discard database changes, a transaction must roll back its changes using Rollback. Rollback undoes a transaction's changes and ends the transaction. For example, the following statement rolls back a transaction:

```
DatabaseInterBase.Rollback;
```

Rollback usually occurs in:

- Exception handling code when you cannot recover from a database error.
- Button or menu event code, such as when a user clicks a Cancel button.

Using the TransIsolation property

TransIsolation specifies the transaction isolation level for a database component's transactions. Transaction isolation level determines how a transaction interacts with other simultaneous transactions when they work with the same tables. In particular, it affects how much a transaction "sees" of other transaction's changes to a table. Before changing or setting TransIsolation, you should be familiar with transactions and transactions management in Delphi.

The default setting for TransIsolation is tiReadCommitted. The following table summarizes possible values for TransIsolation and describes what they mean:

Isolation level	Meaning
tiDirtyRead	Permit reading of uncommitted changes made to the database by other simultaneous transactions. Uncommitted changes are not permanent, and might be rolled back (undone) at any time. At this level your transaction is least isolated from the changes made by other transactions.
tiReadCommitted	Permit reading only of committed (permanent) changes made to the database by other simultaneous transactions. This is the default isolation level.
tiRepeatableRead	Permit a single, one time reading of the database. Your transaction cannot see any subsequent changes to data by other simultaneous transactions. This isolation level guarantees that once your transaction reads a record, its view of that record will not change. At this level your transaction is most isolated from changes made by other transactions.

Database servers may support these isolation levels differently or not at all. If the requested isolation level is not supported by the server, then Delphi will use the next highest isolation level. The actual isolation level used by each type of server is listed in the following table. For a detailed description of how each isolation level is implemented, see your server documentation.

TransIsolation setting	Paradox and dBASE	Oracle	Sybase and Microsoft SQL servers	InterBase
Dirty read	Dirty read	Read committed	Read committed	Read committed
Read committed (Default)	Not supported	Read committed	Read committed	Read committed
Repeatable read	Not supported	Repeatable read (READ ONLY)	Not supported	Repeatable Read

Note: When using transactions with local Paradox and dBASE tables, set TransIsolation to tiDirtyRead instead of using the

default value of tiReadCommitted. A BDE error is returned if TransIsolation is set to anything but tiDirtyRead for local tables

If an application is using ODBC to interface with a server, the ODBC driver must also support the isolation level. For more information, see your ODBC driver documentation.

Using passthrough SQL

To be able to use passthrough SQL to control a transaction you must:

- Use the Delphi Client/Server Suite.
- Install the proper SQL Links drivers. If you chose the Standard installation when installing Delphi, all SQL Links drivers are already properly installed.
- Configure your network protocol correctly. See your network administrator for more information.
- Have access to a database on a remote server.
- Use the BDE Configuration utility to set SQLPASSTHROUGHMODE to NOT SHARED.

With passthrough SQL, you use a TQuery, TStoredProc, or TUpdateSQL component to send an SQL transaction control statement directly to a remote database server. The BDE does not process the SQL statement. Using passthrough SQL enables you to take direct advantage of the transaction controls offered by your server, especially when those controls are non-standard.

Setting SQLPASSTHROUGHMODE

SQLPASSTHROUGHMODE specifies whether the BDE and passthrough SQL statements can share the same database connections. In most cases, SQLPASSTHROUGHMODE is set to SHARED AUTOCOMMIT. If however, you want to pass SQL transaction control statements to your server, you must use the BDE Configuration utility to set the BDE SQLPASSTHROUGHMODE to NOT SHARED. For more information about SQLPASSTHROUGH modes, see the online help for the BDE Configuration utility.

Note: When SQLPASSTHROUGHMODE is NOT SHARED, you must use separate database components for *TQuery* components that pass SQL transaction statements to the server and those other dataset components that do not.

Using local transactions

The BDE supports local transactions against Paradox and dBASE tables. From a coding perspective, there is no difference to you between a local transaction and a transaction against a remote database server.

When a transaction is started against a local table, updates performed against the table are logged. Each log record contains the old record buffer for a record. When a transaction is active, records that are updated are locked until the transaction is committed or rolled back. On rollback, old record buffers are applied against updated records to restore them to their pre-update states.

Understanding the limitations of local transactions

The following limitations apply to local transactions

- Automatic crash recovery is not provided.
- Data definition statements are not supported.
- For Paradox, local transactions can only be performed on tables with valid indexes. Data cannot be rolled back on Paradox tables that do not have indexes.
- Transactions cannot be run against temporary tables.
- Transactions cannot be run against the BDE ASCII driver.
- Closing a cursor on a table during a transaction rolls back the transaction unless:

  Several tables are open.

  The cursor is closed on a table to which no changes were made.

Using TSession

Each time an application runs, Delphi creates a default TSession component for it. A session component provides global control over all database connections in an application. If you do not create multi-threaded database applications, you need only concern yourself with the default session in your application.

A multi-threaded database application is a single application that attempts to run two or more simultaneous operations, such as SQL queries, against the same database. If you create a multi-threaded database application, then each additional thread after the first requires its own session component.

Creating a session component

If you are creating a multi-threaded database application, then you need to create additional session components. To create a session component, place a session component from the Data Access page onto a data module used in your project.

The following table lists the properties of TSession, what they are used for, their default values if any, and whether they are available at design time:

Property	Purpose	Default	Design time access
Active	StartEasterTrue, starts the BDE session. False, disconnects datasets and stops the session.EndEaster	False	Yes
Databases	Specifies an array of all active databases in the session.		No
DatabaseCount	Provides an integer value specifying the number of currently active databases in the session.		No
KeepConnections	True, maintain database connection(s) even if there are no open datasets. False, close database connection(s) when there are no open datasets.	True	Yes
Name	Names a session component StartEaster(for example, Session1EndEaster).		Yes
NetFileDir	Specifies the directory path of the Paradox network control file, which enables sharing of Paradox tables on network drives.		Yes
PrivateDir	Specifies the path in which to store temporary StartEasterEndEasterfiles (for example, files used to process local SQL statements).		Yes
SessionName	Specifies the name of the session that must be used by database components to link themselves to a particular session.		Yes

Using TSessionList

Multi-threaded applications need to manage sessions through a TSessionList component. You never need to create your own session list component, but you may certainly need to manipulate the properties and use the methods of the default session list.

A default session list component, called Sessions, is created for you whenever you start a database application.

Sessions is a component of type TSessionList. You use the properties and methods of Sessions to keep track of multiple sessions in a multi-threaded database application. The following table summarizes the properties and methods of the TSessionList component:

Property or Method	Purpose
Count	Returns the number of sessions, both active and inactive, in the sessions list.
FindSession	Searches the session list for a session with a specified name, and returns a pointer to the session component, or nil if there is no session with the specified name. If passed a blank session name, FindSession returns a pointer to the default session, Session.
GetSessionNames	Returns a string list containing the names of all currently instantiated session components. This procedure always returns at least one string, 'Default' for the default session (note that the default session's name is actually a blank string).
List	Returns the session component for a specified session name. If there is no session with the specified name, an exception is raised.
OpenSession	Creates and activates a new session or reactivates an existing session for a specified session name.
Sessions	Accesses the session list by ordinal value. Sessions is the default property, so you can call it like so Sessions[0], rather than Sessions.Sessions[0].

Specfiying a data source

A TDataSource component is a nonvisual database component that acts as a conduit between a dataset and data-aware components on a form that enable the display, navigation, and editing of the data underlying the dataset. All datasets must be associated with a data source component in order for their data to be displayed and manipulated in data-aware controls on a form. Similarly, each data-aware control needs to be associated with a data source component in order to have data to display and manipulate. You also use data source components to link datasets in master-detail relationships.

You place a data source component in a data module or form just as you place other nonvisual database components. You should place at least one data source component for each dataset component in a data module or form.

Using TDataSource properties

TDataSource has only a few published properties. The following sections discuss these key properties and how to set them at design time and run time.

Setting the DataSet property

The DataSet property specifies the name of the dataset from which the a data source component gets its data. At design time you can select a dataset from the drop down list in the Object Inspector. At run time you can switch the dataset for a data source component as needed. For example, the following code swaps the dataset for the CustSource data source component between Customers and Orders:

```
with CustSource do
begin
  if DataSet = 'Customers' then
    DataSet := 'Orders'
  else
    DataSet := 'Customers';
end;
```

You can also set the DataSet property to a dataset on another form to synchronize the data controls on the two forms. For example:

```
procedure TForm2.FormCreate (Sender : TObject);
begin
  DataSource1.Dataset := Form1.Table1;
end;
```

Setting the Name property

Name enables you to specify a meaningful name for a data source component that distinguishes it from all other data sources in your application. The name you supply for a data source component is displayed below the component's icon in a data module.

Generally, you should provide a name for a data source component that indicates the dataset with which it is associated. For example, suppose you have a dataset called Customers, and that you link a data source component to it by setting the data source component's DataSet property to "Customers." To make the connection between the dataset and data source obvious in a data module, you should set the Name property for the data source component to something like "CustomersSource".

Setting the Enabled property

The Enabled property determines if a data source component is connected to its dataset. When Enabled is True, the data source is connected to a dataset.

You can temporarily disconnect a single data source from its dataset by setting Enabled to False. When Enabled is False, all data controls attached to the data source component go blank and become inactive until Enabled is set to True. It is recommended, however, to control access to a dataset through a dataset component's DisableControls and EnableControls methods because they affect all attached data sources.

Setting the AutoEdit property

The *AutoEdit* property of *TDataSource* specifies whether datasets connected to the data source automatically enter Edit state when the user starts typing in data-aware controls linked to the dataset. If *AutoEdit* is True (the default), Delphi automatically puts the dataset in Edit state when a user types in a linked data-aware control. Otherwise, a dataset enters Edit state only when the application explicitly calls its *Edit* method.

Using TDataSource methods

TDataSource has three event handlers associated with it:

- OnDataChange
- OnStateChange
- OnUpdateData

Using the OnDataChange event

OnDataChange is called whenever the cursor moves to a new record. When an application calls Next, Previous, Insert, or any method that leads to a change in the cursor position, then an OnDataChange is triggered.

This event is useful if an application is keeping components synchronized manually.

Using the OnUpdateData event

OnUpdateData is called whenever the data in the current record is about to be updated. For instance, an OnUpdateData event

occurs after Post is called, but before the data is actually posted to the database.

This event is useful if an application uses a standard (non-data aware) control and needs to keep it synchronized with a dataset.

Using the OnStateChange event

OnStateChange is called whenever the state for a data source's dataset changes. A dataset's State property records its current state. OnStateChange is useful for performing actions as a TDataSource's state changes.

For example, during the course of a normal database session, a dataset's state changes frequently. To track these changes, you could write an OnStateChange event handler that displays the current dataset state in a label on a form. The following code illustrates one way you might code such a routine. At run time, this code displays the current setting of the dataset's State property and updates it every time it changes:

```
procedure TForm1.DataSource1.StateChange(Sender:TObject);
var
  S:String;
begin
  case CustTable.State of
    dsInactive: S := 'Inactive';
    dsBrowse: S := 'Browse';
    dsEdit: S := 'Edit';
    dsInsert: S := 'Insert';
    dsSetKey: S := 'SetKey';
  end;
  CustTableStateLabel.Caption := S;
end;
```

Similarly, *OnStateChange* can be used to enable or disable buttons or menu items based on the current state:

```
procedure Form1.DataSource1.StateChange(Sender: TObject);
begin
  CustTableEditBtn.Enabled := (CustTable.State = dsBrowse);
  CustTableCancelBtn.Enabled := CustTable.State in [dsInsert, dsEdit, dsSetKey];
  ...
end;
```

Displaying data in a data control

To display data in a data control:

1. Place a data control from the Data Access page of the Component palette onto a form.
2. Set the *DataSource* property of the control to the name of a data source component from which to get data. A data source component acts as a conduit between the control and a dataset containing data.
3. Set the *DataField* property of the control to the name of a field to display, or select a field name from the drop-down list for the property.

If the *Enabled* property of the data control's data source is *True* (the default), and the *Active* property of the dataset attached to the data source is also *True*, data is now displayed in the data control.

Note: Two data controls, TDBGrid and TDBNavigator, access all available field components within a dataset, and therefore do not have DataField properties. For these controls, omit Step 3.

Enabling mouse, keyboard, and timer events

The Enabled property of a data control determines whether it responds to mouse, keyboard, or timer events, and passes information to its data source. The default setting for this property is True.

To prevent mouse, keyboard, or timer events from accessing a data control set its Enabled property to False. When Enabled is False, a data source does not receive information from the data control. The data control continues to display data, but the text displayed in the control is dimmed.

Enabling editing in controls on user entry

A dataset must be in dsEdit state to permit editing to its data. The AutoEdit property of the data source to which a control is attached determines if the underlying dataset enters dsEdit mode when data in a control is modified in response to keyboard or mouse events. When AutoEdit is True (the default), dsEdit mode is set as soon as editing commences. If AutoEdit is False, you must provide a TDBNavigator control with an Edit button (or some other method) to permit users to set dsEdit state at run time.

Updating fields

The ReadOnly property of a data control determines if a user can edit the data displayed by the control. If False (the default), users can edit data. To prevent users from editing data in a control, set ReadOnly to True.

Properties of the data source and dataset underlying a control also determine if the user can successfully edit data with a control and can post changes to the dataset.

The Enabled property of a data source determines if controls attached to a data source are able to display fields values from the dataset, and therefore also determine if a user can edit and post values. If Enabled is True (the default), controls can display field values.

The ReadOnly property of the dataset determines if user edits can be posted to the dataset. If False (the default), changes are posted to the dataset. If False, the dataset is read-only.

Note: Table components have an additional, read-only run-time property CanModify that determines if a dataset can be modified. CanModify is set to True if a database permits write access. If CanModify is False, a dataset is read-only. Query components that perform inserts and updates are, by definition, able to write to an underlying database, provided that your application and user have sufficient write privileges to the database itself.

The following table summarizes the factors that determine if a user can edit data in a control and post changes to the database:

Data control ReadOnly property	Data source Enabled property	Dataset ReadOnly property	Dataset CanModify property (tables only)	Database write access	Can write to database?
False	True	False	True	Read/Write	Yes
False	True	False	False	Read-only	No
False	False	—	—	—	No
True	—	—	—	—	No

In all data controls except TDBGrid, when you modify a field, the modification is copied to the underlying field component in a dataset when you Tab from the control. If you press Esc before you Tab from a field, Delphi abandons the modifications, and the value of the field reverts to the value it held before any modifications were made.

In TDBGrid, modifications are copied only when you move to a different record; you can press Esc in any record of a field before moving to another record to cancel all changes to the record.

When a record is posted, Delphi checks all data-aware components associated with the dataset for a change in status. If there is a problem updating any fields that contain modified data, Delphi raises an exception, and no modifications are made to the record.

Disabling and enabling data display

When your application iterates through a dataset or performs a search, you should temporarily prevent refreshing of the values displayed in data-aware controls each time the current record changes. Preventing refreshing of values speeds the iteration or search and prevents annoying screen-flicker.

DisableControls is a dataset method that disables display for all data-aware controls linked to a dataset. As soon as the iteration or search is over, your application should immediately call the dataset's EnableControls method to re-enable display for the controls.

Usually you disable controls before entering an iterative process. The iterative process itself should take place inside a try...finally statement so that you can re-enable controls even if an exception occurs during processing. The finally clause should call EnableControls. The following code illustrates how you might use DisableControls and EnableControls in this manner:

```
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets EOF False }
  while not CustTable.EOF do { Cycle until EOF is True }
  begin
    { Process each record here }
    ƒ
    CustTable.Next; { EOF False on success; EOF True when Next fails on last record }
  end;
finally
  CustTable.EnableControls;
end;
```

Refreshing data

The Refresh method for a dataset flushes local buffers and refetches data for an open dataset. You can use this method to update the display in data-aware controls if you think that the underlying data has changed because other applications have simultaneous access to the data used in your application.

Important

Refreshing can sometimes lead to unexpected results. For example, if a user is viewing a record deleted by another application, then the record disappears the moment your application calls *Refresh*. Data can also appear to change if another user changes a record after you originally fetched the data and before you call *Refresh*.

Opening and closing datasets

To read or write data in a table or through a query, an application must first open a dataset. There are two ways to open a dataset:

- Set the *Active* property of the dataset to *True*, either at design time in the Object Inspector, or in code at run time:

```
CustTable.Active := True;
```

- Call the *Open* method for the dataset at run time:

```
CustQuery.Open;
```

There are also two ways to close a dataset:

- Set the Active property of the dataset to False, either at design time in the Object Inspector, or in code at run time:

```
CustQuery.Active := False;
```

- Call the Close method for the dataset at run time:

```
CustTable.Close;
```

You need to close a dataset when you want to change any of its properties that affect the query, such as the DataSource property. At run time you may also want to close a dataset for other reasons specific to your application.

Setting dataset states

The state—or mode—of a dataset determines what can be done to its data. For example, when a dataset is closed, its state is dsInactive, meaning that nothing can be done to its data. A dataset is always in one state or another. At run time you can examine a dataset's read-only State property to determine its current state. The following table summarizes possible values for the State property and what they mean:

Value	State	Meaning
dsInactive	Inactive	Dataset closed. Its data is unavailable.
DsBrowse	Browse	Dataset open. Its data can be viewed, but not changed. This is the default state of an open dataset.
DsEdit	Edit	Dataset open. The current row can be modified.
DsInsert	Insert	Dataset open. A new row can be inserted.
DsSetKey	SetKey	TTable only. Dataset open. Enables searching for rows based on indexed fields, or indicates that a SetRange operation is under way. A restricted set of data can be viewed, and no data can be changed.
DsCalcFields	CalcFields	Dataset open. Indicates that an OnCalcFields event is under way. Prevents changes to fields that are not calculated.
DsUpdateNew	UpdateNew	Internal use only.
DsUpdateOld	UpdateOld	Internal use only.
DsFilter	Filter	Dataset open. Indicates that a filter operation in under way. A restricted set of data can be viewed, and no data can be changed.

When an application opens a dataset, Delphi automatically puts the dataset into dsBrowse mode. The state of a dataset changes as an application processes data. An open dataset changes from one state to another based on either the

- Code in your application, or
- Delphi's built-in behavior.

To put a dataset into dsBrowse, dsEdit, dsInsert, or dsSetKey states, call the method corresponding to the name of the state. For example, the following code puts CustTable into dsInsert state, accepts user input for a new record, and writes the new record to the database:

```
CustTable.Insert; { Your application explicitly sets dataset state to Insert }
AddressPromptDialog.ShowModal;
if AddressPromptDialog.ModalResult := mrOK then
  CustTable.Post; { Delphi sets dataset state to Browse on successful completion }
else
  CustTable.Cancel; {Delphi sets dataset state to Browse on cancel }
```

This example also illustrates that Delphi automatically sets the state of a dataset to dsBrowse when

- The *Post* method successfully writes a record to the database. (If *Post* fails, the dataset state remains unchanged.)
- The *Cancel* method is called.

Some states cannot be set directly. For example, to put a dataset into dsInactive state, set its Active property to False, or call the Close method for the dataset. The following statements are equivalent:

```
CustTable.Active := False;
CustTable.Close;
```

The remaining states (dsCalcFields, dsUpdateNew, dsUpdateOld, and dsFilter) cannot be set by your application. Instead, Delphi sets them as necessary For example, dsCalcFields is set when a dataset's OnCalcFields event is called. When the OnCalcFields event finishes, the dataset is restored to its previous state.

Note: Whenever a dataset's state changes, the *OnStateChange* event is called for any data source components associated with the dataset.

dsInactive

A dataset is inactive when it is closed. You cannot access records in a closed dataset. At design time a dataset is closed until you set its Active property to True. At run time, a dataset is closed until it is opened either by calling the Open method, or by setting the Active property to True. When you open an inactive dataset, Delphi automatically puts it into dsBrowse state.

To make a dataset inactive, call its Close method. You can write BeforeClose and AfterClose event handlers that respond to the Close method for a dataset. For example, if a dataset is in dsEdit or dsInsert modes when an application calls Close, you should prompt the user to post pending changes or cancel them before closing the dataset. The following code illustrates such a handler:

```
procedure CustTable.VerifyBeforeClose(DataSet: TDataSet)
begin
  if (CustTable.State = dsEdit) or (CustTable.State = dsInsert) then
  begin
    if MessageDlg('Post changes before closing?', mtConfirmation, mbYesNo, 0) = mrYes then
      CustTable.Post;
    else
      CustTable.Cancel;
```

```
  end;
end;
```

To associate a procedure with the BeforeClose event for a dataset at design time:

1. Select the table in the data module (or form).
2. Click the Events page in the Object Inspector.
3. Enter the name of the procedure for the *BeforeClose* event (or choose it from the drop-down list).

dsBrowse

When an application opens a dataset, Delphi automatically puts the dataset into dsBrowse state. Browsing enables you to view records in a dataset, but you cannot edit records or insert new records. You mainly use dsBrowse to scroll from record to record in a dataset.

From dsBrowse all other dataset states can be set. For example, calling the Insert or Append methods for a dataset changes its state from dsBrowse to dsInsert (note that other factors and dataset properties such as CanModify, may prevent this change). Calling SetKey to search for records puts a dataset in dsSetKey mode.

Two methods associated with all datasets can return a dataset to dsBrowse state. Cancel ends the current edit, insert, or search task, and always returns a dataset to dsBrowse state. Post attempts to write changes to the database, and if successful, also returns a dataset to dsBrowse state. If Post fails, the current state remains unchanged.

dsEdit

A dataset must be in dsEdit mode before an application can modify records. In your code you can use the Edit method to put a dataset into dsEdit mode if the read-only CanModify property for the dataset is True. CanModify is True if the database underlying a dataset permits read and write privileges.

On forms in your application, some data-aware controls can automatically put a dataset into dsEdit state if:

- The control's *ReadOnly* property is *False* (the default),
- The *AutoEdit* property of the data source for the control is *True*, and
- *CanModify* is *True* for the dataset.

Important

For TTable components only, if the ReadOnly property is True, CanModify is False, preventing editing of records.

Note: Even if a dataset is in *dsEdit* state, editing records may not succeed for SQL-based databases if your application user does not have proper SQL access privileges.

You can return a dataset from dsEdit state to dsBrowse state in code by calling the Cancel, Post, or Delete methods. Cancel discards edits to the current field or record. Post attempts to write a modified record to the dataset, and if it succeeds, returns the dataset to dsBrowse. If Post cannot write changes, the dataset remains in dsEdit state. Delete attempts to remove the current record from the dataset, and if it succeeds, returns the dataset to dsBrowse state. If Delete fails, the dataset remains in dsEdit state.

Data-aware controls for which editing is enabled automatically call Post when a user executes any action that changes the current record (such as moving to a different record in a grid) or that causes the control to lose focus (such as moving to a different control on the form).

dsInsert

A dataset must be in dsInsert mode before an application can add new records. In your code you can use the Insert or Append methods to put a dataset into dsInsert mode if the read-only CanModify property for the dataset is True. CanModify is True if the database underlying a dataset permits read and write privileges.

On forms in your application, the data-aware grid and navigator controls can put a dataset into dsInsert state if

- The control's *ReadOnly* property is *False* (the default),
- The *AutoEdit* property of the data source for the control is *True*, and
- *CanModify* is *True* for the dataset.

Important

For *TTable* components only, if the *ReadOnly* property is *True*, *CanModify* is *False*, preventing editing of records.

Note

Even if a dataset is in *dsInsert* state, inserting records may not succeed for SQL-based databases if your application user does not have proper SQL access privileges.

You can return a dataset from dsInsert state to dsBrowse state in code by calling the Cancel, Post, or Delete methods. Delete and Cancel discard the new record. Post attempts to write the new record to the dataset, and if it succeeds, returns the dataset to dsBrowse. If Post cannot write the record, the dataset remains in dsInsert state.

Data-aware controls for which inserting is enabled automatically call Post when a user executes any action that changes the current record (such as moving to a different record in a grid).

dsSetKey

dsSetKey mode applies only to TTable components. To search for records in a TTable dataset, the dataset must be in dsSetKey mode. You put a dataset into dsSetKey mode with the SetKey method at run time. The GotoKey method, which carries out the actual search, returns the dataset to dsBrowse state upon completion of the search.

Note: Other search methods, including *FindKey*, and *FindNearest* automatically put a dataset into *dsSetKey* state during a search, and return the dataset to *dsBrowse* state upon completion of the search.

dsCalcFields

Delphi puts a dataset into dsCalcFields mode whenever an application calls the dataset's OnCalcFields event handler. This state prevents modifications or additions to the records in a dataset except for the calculated fields the handler modifies itself. The reason all other modifications are prevented is because OnCalcFields uses the values in other fields to derive values for calculated fields. Changes to those other fields might otherwise invalidate the values assigned to calculated fields.

When the OnCalcFields handler finishes, the dataset is returned to dsBrowse state.

dsFilter

Delphi puts a dataset into dsFilter mode whenever an application calls the dataset's OnFilterRecord event handler. This state prevents modifications or additions to the records in a dataset during the filtering process so that the filter request is not invalidated.

When the OnFilterRecord handler finishes, the dataset is returned to dsBrowse state.

Using the Fields editor

The Fields editor enables you to create, delete, arrange, and define persistent field components associated with a dataset. Persistent field components guarantee your application receives a consistent view of the data underlying a dataset.

To start the Fields editor:

- Double-click the dataset component, or
- Select the component, right-click to invoke the SpeedMenu, then choose Fields editor.

The Fields editor contains a title bar, navigator buttons, and a list box.

The title bar of the Fields editor displays both the name of the data module or form containing the dataset, and the name of the dataset itself. So, if you open the Customers dataset in the CustomerData data module, the title bar displays 'CustomerData.Customers', or as much of the name as fits.

Below the title bar is a set of navigation buttons that enable you to scroll one-by-one through the records in an active dataset at design time, and to jump to the first or last record. The navigation buttons are dimmed if the dataset is not active or if the dataset is empty.

The list box displays the names of persistent field components for the dataset. The first time you invoke the Fields editor for a new dataset, the list is empty because the field components for the dataset are dynamic, not persistent. If you invoke the Fields editor for a dataset that already has persistent field components, you see the field component names in the list box.

Creating persistent field components

To create a persistent field component for a dataset:

- Right-click the Fields editor list box.
- Choose Add fields from the SpeedMenu. The Add Fields dialog box appears.

The Available fields list box displays all fields in the dataset which do not have persistent field components. Select the fields for which to create persistent field components, and click OK.

The Add Fields dialog box closes, and the fields you selected appear in the Fields editor list box. Fields in the Fields editor list box are persistent. If the dataset is active, note, too, that the Next and Last navigation buttons above the list box are enabled.

From now on, each time you open the dataset, Delphi no longer creates dynamic field components for every column in the underlying database. Instead it only creates persistent components for the fields you specified.

Each time you open the dataset, Delphi verifies that each non-calculated persistent field exists or can be created from data in the database. If it cannot, Delphi raises an exception warning you that the field is not valid, and does not open the dataset.

Deleting persistent field components

Deleting a persistent field component is useful for accessing a subset of available columns in a table, and for defining your own persistent fields to replace a column in a table.To remove one or more persistent field components for a dataset:

1. Select the field(s) to remove in the Fields editor list box.
2. Press *Del*.

Note: You can also delete selected fields by invoking the SpeedMenu and choosing Delete.

Fields you remove are no longer available to the dataset and cannot be displayed by data-aware controls. You can always re-create persistent field components that you delete by accident, but any changes previously made to its properties or events is lost.

Note: If you remove all persistent field components for a dataset, then Delphi again generates dynamic field components for every column in the database table underlying the dataset.

Arranging the order of persistent field components

The order in which persistent field components are listed in the Fields editor list box is the default order in which the fields appear in a data-aware grid component. You can change field order by dragging and dropping fields in the list box.

To change the order of a single field:

1. Select the field.
2. Drag it to a new location.

Alternatively, you can select the field, and use Ctrl-Up and Ctrl-Dn to move the field to a new location in the list.

To change the order for a block of fields:

1. Select the fields.
2. Drag them to their new location.

If you select a non-contiguous set of fields and drag them to a new location, they are inserted as a contiguous block. Within the block the order of fields to one another does not change.

Defining new persistent field components

Besides selecting which dynamic field components to make into persistent field components for a dataset, you can also create new persistent fields as additions to or replacements of the other persistent fields in a dataset. There are three types of persistent fields you can create:

- *Data fields*, which usually replace existing fields (for example to change the data type of a field), are based on columns in

the table or query underlying a dataset.

- *Calculated fields*, which displays values calculated at run time by a dataset's OnCalcFields event handler.
- *Lookup fields*, which retrieve values from a specified dataset at run time based on search criteria you specify.

These types of persistent fields are only for display purposes. The data they contain at run time are not retained either because they already exist elsewhere in your database, or because they are temporary. The physical structure of the table and data underlying the dataset is not changed in any way.

To create a new persistent field component:

1. Right-click the Fields editor list box.
2. Choose New field from the SpeedMenu.

   The New Field dialog box appears.

Dialog box options

New Field dialog box
The New Field dialog box contains three group boxes: Field properties, Field type, and Lookup definition.

Field type radio group
The Field type radio group box enables you to specify the type of new field component to create. The default type the first time you open the New Field dialog box in a session is Data. If you choose Lookup, the Dataset and Source Fields edit boxes in the Lookup definition group box are enabled. The Lookup definition group box is only used to create lookup fields.

Field properties group box
The Field properties group box enables you to enter general field component information. Enter the component's field name in the Name edit box. The name you enter here corresponds to the field component's FieldName property. Delphi uses this name to build a component name in the Component edit box. The name that appears in the Component edit box corresponds to the field component's Name property and is only provided for informational purposes (Name contains the identifier by which you refer to the field component in your source code). Delphi discards anything you enter directly in the Component edit box.

Type combo box
The Type combo box in the Field properties group enables you to specify the field component's data type. You must supply a data type for any new field component you create. For example, to display floating point currency values in a field, select Currency from the drop-down list. The Size edit box enables you to specify the maximum number of characters that can be displayed or entered in a string-based field or the size of Bytes and VarBytes fields. For all other data types, Size is meaningless.

Defining a data field

A data field replaces an existing field in a dataset. For example, for programmatic reasons you might want to replace a TSmallIntField with a TIntegerField. Because you cannot change a field's data type directly, you must define a new field to replace it.

Important

Even though you define a new field to replace an existing field, the field you define must derive its data values from an existing column in a table underlying a dataset.

To create a data field in theNew Field dialog box:

1. Enter the name of an existing persistent field in the Name edit box. Do not enter a new field name.
2. Choose a new data type for the field from the Type combo box. The data type you choose should be different from the data type of the field you are replacing.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.
4. Select Data in the Field type radio group if it is not already selected.
5. Choose OK. The New Field dialog box closes, the newly defined data field replaces the existing field you specified in Step 1, and the component declaration in the data module or form's **type** declaration is updated.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector.

Defining and programming a calculated field

A calculated field displays values calculated at run time by a dataset's OnCalcFields event handler. For example, you might create a string field that displays concatenated values from other fields.

To create a calculated field in the New Field dialog box:

1. Enter a name for the calculated field in the Name edit box. Do not enter the name of an existing field.
2. Choose a data type for the field from the Type combo box.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.
4. Select Calculated in the Field type radio group.
5. Choose OK. The newly defined calculated field is automatically added to end of the list of persistent fields in the Field editor list box, and the component declaration is automatically added to the form's **type** declaration in the source code.
6. Place code that calculates values for the field in the *OnCalcFields* event handler for the dataset. For more information about writing code to calculate field values, see "Programming a calculated field," below.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector.

Programming a calculated field

After you define a calculated field, you must write code to calculate its value. Otherwise it always has a null value. Code for a calculated field is placed in the OnCalcFields event for its dataset.

To program a value for a calculated field:

1. Select the dataset component from the Object Inspector drop-down list.
2. Choose the Object Inspector Events page.
3. Double-click the *OnCalcFields* property to bring up or create a *CalcFields* procedure for the dataset component.
4. Write the code that sets the values and other properties of the calculated field as desired.

For example, suppose you have created a CityStateZip calculated field for the Customers table on the CustomerData data module. CityStateZip should display a company's city, state, and zip code on a single line in a data-aware control.

To add code to the CalcFields procedure for the Customers table, select the Customers table from the Object Inspector drop-down list, switch to the Events page, and double-click the OnCalcFields property.

The TCustomerData.CustomersCalcFields procedure appears in the unit's source code window. Add the following code to the procedure to calculate the field:

```
CustomersCityStateZip.Value := CustomersCity.Value + ', ' + CustomersState.Value
    + ' ' + CustomersZip.Value;
```

Defining a lookup field

A lookup field displays values it searches for at run time based on search criteria. In its simplest form, a lookup field is passed the name of a field to search, a field value to search for, and the field in the lookup dataset whose value it should display.

For example, consider a mail-order application that enables an operator use a lookup field to determine automatically the city and state that correspond to a zip code a customer provides. In that case, the column to search on might be called ZipTable.Zip, the value to search for is the customer's zip code as entered in Order.CustZip, and the values to return would be those in the ZipTable.City and ZipTable.State columns for the record where ZipTable.Zip matches the current value in the Order.CustZip field.

To create a lookup field in the New Field dialog box:

1. Enter a name for the lookup field in the Name edit box. Do not enter the name of an existing field.
2. Choose a data type for the field from the Type combo box.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.
4. Select Lookup in the Field type radio group. Selecting Lookup enables the Dataset and Key Fields combo boxes.
5. Choose from the Dataset combo box drop-down list the dataset in which to look up field values. The lookup dataset must be different from the dataset for the field component itself, or a circular reference exception is raised at run time. Specifying a lookup dataset enables the Lookup Keys and Result Field combo boxes.
6. Choose from the Key Fields drop-down list a field in the current dataset for which to match values. To match more than one field, enter field names directly instead of choosing from the drop-down list. Separate multiple field names with semicolons.
7. Choose from the Lookup Keys drop-down list a field in the lookup dataset to match against the Source Fields field you specified in step 6. To specify more than one field, enter field names directly instead. Separate multiple field names with semicolons.
8. Choose from the Result Field drop-down list a field in the lookup dataset to return as the value of the lookup field you are creating. To return values from more than one field in the lookup dataset, enter field names directly instead. Separate multiple field names with semicolons.

Creating Data Dictionary attribute sets for field components

When several fields in the datasets used by your application share common formatting properties (such as Alignment, DisplayWidth, DisplayFormat, EditFormat, MaxValue, MinValue, and so on), it is more convenient to set the properties for a single field, then store those properties as an attribute set in the Data Dictionary. Attribute sets stored in the data dictionary can be easily applied to other fields.

To create an attribute set based on a field component in a dataset:

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to set properties.
3. Set the desired properties for the field in the Object Inspector.
4. Right-click the Fields editor list box to invoke the SpeedMenu.
5. Choose Save Attributes to save the current field's property settings as an attribute set in the Data Dictionary.

The name for the attribute set defaults to the name of the current field. You can specify a different name for the attribute set by choosing Save attributes as instead of Save attributes from the SpeedMenu.

Note: You can also create attribute sets directly from the Database Explorer. When you create an attribute set from the data dictionary, the set is not applied to any fields, but you can specify two additional attributes in the set: a field type (such as *TFloatField*, *TStringField*, and so on) and a data-aware control (such as *TDBEdit*, *TDBCheckBox*, and so on) that is automatically placed on a form when a field based on the attribute set is dragged onto the form.

Associating Data Dictionary attribute sets with field components

When several fields in the datasets used by your application share common formatting properties (such as Alignment, DisplayWidth, DisplayFormat, EditFormat, MaxValue, MinValue, and so on), and you have saved those property settings as attribute sets in the Data Dictionary, you can easily apply the attribute sets to fields without having to recreate the settings manually for each field. In addition, if you later change the attribute settings in the Data Dictionary, those changes are automatically applied to every field associated with the set the next time field components are added to the dataset.

To apply an attribute set to a field component:

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to apply an attribute set.
3. Right-click the Fields editor list box to invoke the SpeedMenu.
4. Choose Associate attributes.
5. Select or enter the attribute set to apply from the Attribute set name dialog box. If there is an attribute set in the Data Dictionary that has the same name as the current field, that set name appears in the edit box.

Unassociating a Data Dictionary attribute set from a field component

If you change your mind about associating an attribute set with a field component, you can easily remove the attribute set from the field component:

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to remove an attribute set.
3. Right-click the Fields editor list box to invoke the SpeedMenu.
4. Choose Unassociate attributes.

After you remove an attribute set from a field component, you can either use the Object Inspector to set its properties, or you can associate a different attribute set with the component.

Accessing field values with the default dataset method

The preferred method for accessing a field's value is to use variants with the default dataset method, FieldValues. For example, the following statement puts the value of an edit box into the CustNo field in the Customers table:

```
Customers['CustNo'] := Edit2.Text;
```

Because FieldValues is the default method for a dataset, you do not need to specify its method name explicitly. The following statement, however, is identical to the previous one:

```
Customers.FieldValues['CustNo'] := Edit2.Text;
```

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

Accessing field values with the Fields property

You can access the value of a field with the Fields property of the dataset component to which the field belongs. Accessing field values with a dataset's Fields property is useful when you need to iterate over a number of columns, or if your application works with tables that are not available to you at design time.

To use the Fields property you must know the order of and data types of fields in the dataset. You use an ordinal number to specify the field to access. The first field in a dataset is numbered 0. Field values must be converted as appropriate using the field component's conversion routine. For more information about field component conversion functions, see "Using field component conversion functions."

For example, the following statement assigns the current value of the seventh column (Country) in the Customers table to an edit control:

```
Edit1.Text := CustTable.Fields[6].AsString;
```

Conversely, you can assign a value to a field by setting the Fields property of the dataset to the desired field. For example:

```
begin
  Customers.Edit;
  Customers.Fields[6].AsString := Edit1.Text;
  Customers.Post;
end;
```

Accessing field values with the FieldByName method

You can also access the value of a field with a dataset's FieldByName method. This method is useful when you know the name of the field you want to access, but do not have access to the underlying table at design time.

To use FieldByName, you must know the dataset and name of the field you want to access. You pass the field's name as an argument to the method. To access or change the field's value, convert the result with the appropriate field component conversion function, such as AsString or AsInteger. For example, the following statement assigns the value of the CustNo field in the Customers dataset to an edit control:

```
Edit2.Text := Customers.FieldByName('CustNo').AsString;
```

Conversely, you can assign a value to a field:

```
begin
  Customers.Edit;
  Customers.FieldByName('CustNo').AsString := Edit2.Text;
  Customers.Post;
end;
```

Using conversion functions

Conversion functions attempt to convert one data type to another. For example, the AsString function converts numeric and Boolean values to string representations. The following table lists field component conversion functions, and which functions are recommended for field components by field-component type:

Function	TStringField	TIntegerField	TSmallintField	TWordField	TFloatField	TCurrencyField	TBCDField	TDateTimeField	TDateField	TTimeField	TBooleanField	TBytesField	TVarBytesField	TBlobField	TMemoField	TGraphicField
AsVariant	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
AsString		3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
AsInteger	3				3	3	3									
AsFloat	3	3	3	3				3	3	3						
AsCurrency	3	3	3	3				3	3	3						
AsDateTime	3															
AsBoolean	3															

Note that the AsVariant method is recommended to translate among all data types. When in doubt, use AsVariant.

In some cases, conversions are not always possible. For example, AsDateTime can be used to convert a string to a date, time, or datetime format only if the string value is in a recognizable datetime format. A failed conversion attempt raises an exception.

In some other cases, conversion is possible, but the results of the conversion are not always intuitive. For example, what does it mean to convert a TDateTimeField value into a float format? AsFloat converts the date portion of the field to the number of days since 12/31/1899, and it converts the time portion of the field to a fraction of 24 hours. The following table lists permissible conversions that produce special results:

Conversion	Result
String to Boolean	Converts "True," "False," "Yes," and "No" to Boolean. Other values raise exceptions.
Float to Integer	Rounds float value to nearest integer value.
DateTime to Float	Converts date to number of days since 12/31/1899, time to a fraction of 24 hours.
Boolean to String	Converts any Boolean value to "True" or "False."

In other cases, conversions are not possible at all. In these cases, attempting a conversion also raises an exception.

You use a conversion function as you would use any method belonging to a component: append the function name to the end of the component name wherever it occurs in an assignment statement. Conversion always occurs before an actual assignment is made. For example, the following statement converts the value of CustomersCustNo to a string and assigns the string to the text of an edit control:

```
Edit1.Text := CustomersCustNo.AsString;
```

Conversely, the next statement assigns the text of an edit control to the CustomersCustNo field as an integer:

```
MyTableMyField.AsInteger := StrToInt(Edit1.Text);
```

An exception occurs if an unsupported conversion is performed at run time.

Deploying an application

To make an application available to end users, you deploy it. Deployment involves preparing a package for your end users that enables them to install and run your application and access data with it. Delphi ships with InstallShield Express, a product that helps you deploy your applications.

A deployment package usually includes your application executable file, and support files, such as DLLs, and data files. Particularly when you are upsizing an application to use a remote server, there are additional files and DLLs you need to deploy. For example, to access remote database servers, your application must use the appropriate Borland SQL Links drivers and support files.

Before you deploy any application you build with Delphi, see the online text file DEPLOY.TXT for the latest information about deployment and licensing.

Overview

This Help system describes how to install and configure the InterBase Server for Windows NT provided with Delphi Client/Server. It provides information on the following topics:

Important: DO NOT install InterBase Server V4.0 for NT and Local InterBase Server V4.1.0.116 on the same PC. If you do, the NT server configuration will be invalidated and the server will not run.

System Requirements

The InterBase server for Windows NT requires:

- An IBM-compatible 80486 PC with a hard disk and a floppy drive or CD-ROM drive.
- 16MB RAM minimum; 24MB recommended.
- Microsoft Windows NT 3.1 or higher.
- At least 6MB of free hard disk space.

Installation

To install InterBase from within Windows NT, follow these steps:

1. Start Windows NT and log in as Administrator or as a user in the Administrators user group.
2. Put the InterBase for Windows NT CD-ROM in the CD drive.
3. Choose File | Run... from the Program Manager Main menu.
4. In the Command Line text box, if the InterBase CD is in drive E, type:

   ```
   E:\nt_ibsvr\disk1\install
   ```

   then choose OK. If the CD drive designation is other than E, then type the appropriate drive letter. This starts the installation program.

A dialog box will appear asking for the directory in which to install InterBase. The default is C:\IBSERVER.

To change the directory to which InterBase is installed, type the directory path in the text field. A directory on a disk with an NTFS file partition is recommended, but not required. Choose Continue. The installation program will then create a directory with the specified name.

Then a dialog box will open stating the total disk space required to install InterBase and the available space on the specified disk.

If there is not sufficient space, then choose Cancel and remove files until there is enough space for installation. If there is sufficient space, choose Continue.

The installation program will look for the SERVICES file in the %SystemRoot%\SYSTEM32\DRIVERS\ETC directory (%SystemRoot% specifies the drive and directory where Windows NT is installed, usually C:\WINNT). If it finds the SERVICES file, it will check for the appropriate InterBase entry in the file and then edit it if necessary.

If the installation program does not find the SERVICES file, then a dialog box will appear asking you to enter its location.

If you know where the SERVICES file is, enter the drive and directory path in the text field and choose Continue. the Installation program will then edit the SERVICES file for you.

If you do not know the location of the SERVICES file, choose Skip.

After installation you can find the SERVICES file by using File | Search in the File Manager or asking your system administrator. If you want to use TCP/IP, you must edit the SERVICES file manually and add the following line to the file:

```
gds_db 3050/tcp
```

This line specifies the name of the InterBase port and the port ID used by TCP/IP.

You can choose to exit installation, edit SERVICES, and then install InterBase. If you elect to use this process, manual edit of the SERVICES file will not be required.

After you choose Continue or Skip, the installation program will then install all the InterBase files.

Then a dialog box will open that enables you to choose between automatic or manual restart of the InterBase service.

If you select Automatic, the NT service for the InterBase server starts automatically when NT is started. If you select Manual, InterBase must be started manually each time NT is restarted. Select the configuration you want, and choose Continue to complete the installation.

The README file will then be opened in the Notepad editor. Choose File | Exit to exit the editor.

Testing Your InterBase Installation

To test you InterBase installation:

1. Start the InterBase Server.
2. Run ISQL from /IBSERVER/BIN
3. At the ISQL prompt type:

```
SQL> connect/iberserver/examples/employee.gdb:
SQL> SELECT * FROM EMPLOYEE;
```

   You may elect to use other queries.

4. Exit ISQL and logoff the InterBase Server.

Environment Variables

The PATH, INCLUDE, and LIB environment variables must be set, either system-wide by the system administrator or by each user individually. They should be set up as follows (for the default installation directory):

- PATH should contain \IBSERVER\BIN.
- INCLUDE should contain \IBSERVER\INCLUDE.
- LIB should contain \IBSERVER\LIB.

If InterBase is installed to a directory other than the default IBSERVER, then substitute that directory for IBSERVER above.

Stopping the InterBase Service

There are two ways to stop the InterBase service: through the Control Panel Services application and with a command entered at the NT command prompt.

To use the Services application, double-click the Control Panel icon in the Main program group in the Program Manager. Then double-click the Services icon. A dialog box will appear showing all the services and their status. Click the InterBase Remote Service to highlight it and then click the Stop button.

To use the command prompt method, double-click the Command Prompt icon in the Main program group in the Program Manager. Then type:

```
instsvc stop -z
```

This command will stop the InterBase remote service.

Restarting the InterBase Service

There are two ways to restart the InterBase service: through the Control Panel Services application and with a command entered at the NT command prompt.

To use the Services application, invoke the Control Panel application in the Main program group. Then double-click the Services icon. A dialog box will open showing all the services and their status. Click the InterBase Remote Service to highlight it and then click the Start button. This will start the InterBase Service.

To start the InterBase service from an NT Command Prompt window, type:

```
instsvc start -z
```

This command will restart the InterBase remote service with the previous configuration and the default priority. For information about altering the process priority, see Changing Process Priority.

Re-configuring the InterBase Service

To change whether the InterBase remote service is configured to start automatically or manually, invoke the Services application in the Control Panel program group (in Main). A dialog box will open showing all the services on the system.

If the status of the InterBase Remote Service is listed as "Started", then stop it as described in the previous section. Once the service has been stopped, you can change the Startup parameter to "Automatic" or "Manual" as desired. Click on Startup and choose the desired startup configuration. For more information, see the Microsoft Windows NT documentation.

Changing Process Priority

The InterBase process is a background process that by default runs at normal priority. If there are other processes running at high priority, you may wish to increase the priority of the InterBase process. You can increase InterBase's priority with the command:

```
instsvc configure -b[oostpriority]
```

You must then stop and restart the InterBase service as described previously. Thereafter, all subsequent server processes (database connections) will be started with higher priority.

To set the default priority of the InterBase process back to its regular level, use the command:

```
instsvc configure -r[egularpriority]
```

Then stop and restart the InterBase service. All subsequent server processes will then be started with regular priority.

You can override the default process priority by starting the service with:

```
instsvc start -b[oostpriority]
```

to start the process at high priority or:

```
instsvc start -r[egularpriority]
```

to start the process at regular priority.

If InterBase is not getting the CPU share that it needs, and for some reason you do not want to boost the priority of the InterBase process, and you can change the default priority of foreground and background processes. Invoke the System application in the Control Panel program group.

In the System dialog box that opens, choose the Tasking... button, then select the option "Foreground and Background Applications Equally Responsive". This will ensure that the InterBase process (that runs in the background) will receive the same priority as foreground processes. Note that all other background processes will have their priority boosted as well.

For more information on foreground and background tasking, see the Microsoft Windows NT documentation.

Installing a New Version

Installation of a new version of the server will not overwrite the security database, ISC4.GDB, so your existing user configurations will be preserved. However, it is recommended you back up ISC4.GDB before installing in case there is a disk crash or other disaster.

To install a new version of InterBase on the server (when a previous version has been installed):

1. Stop the current InterBase service, as described previously.
2. Remove the service from the services directory by typing:

   ```
   instsvc remove -z
   ```
3. Remove the InterBase sub-key from the NT Registry. At the command prompt, type:

   ```
   instreg remove -z
   ```

   This command removes InterBase from the NT Registry.
4. Install the new version as described previously.

Entering Commands

Enter InterBase commands in an NT Command Prompt window. Do not use quoted strings in commands entered on the NT command line.

You cannot issue InterBase commands from the Windows NT POSIX sub-system or OS/2 sub-system.

Interactive SQL

With InterBase for Windows NT, you can issue interactive SQL commands with:

- Windows ISQL, a Windows application running on any client PC connected to the server over a network (or running on the server machine). For more information about using Windows ISQL, see the Windows ISQL overview in Help.
- Command-line ISQL, running in an NT Command Prompt window. For information on using command-line ISQL, see the ISQL command reference in Help.

Database Administration

You can perform database administration (DBA) for InterBase for Windows NT servers and databases with:

- The Server Manager, a Windows application running on a client PC connected to the server over a network (or running on the server). For more information about using the Server Manager, see the Server Manager overview in Help.
- The command-line DBA utilities on the server. For information on using the command-line DBA utilities, see Using the Command-line DBA Utilities in Help.

Working With Databases

You cannot connect to a database on an InterBase for Windows NT server using Novell SPX. You can only connect using TCP/IP or NetBEUI/Named Pipes.

If there is a disk drive shared between a Windows NT machine and a Novell NetWare server, you cannot connect to a database on the shared disk using the shared drive name. You can connect to the database on the shared drive using TCP/IP.

Database Names

Databases on FAT file partitions must conform to the DOS FAT file name restrictions: an eight letter base name with a maximum three letter extension. Database names on NTFS file partitions are unrestricted.

User Authorization

To connect to an InterBase database on an Windows NT server, a user must be authorized by an entry in the security database, ISC4.GDB, on that server. To add users to the security database, use Server Manager from a Windows Client or GSEC on the server. For more information about Server Manager, see the Windows Client User's Guide. For more information about GSEC, see the GSEC Help.

The NT Administrator user may connect to a database, and the security database will not be checked if the user is logged in directly to the server platform (the connection is local) or if the user is connecting from a client using NetBEUI/Named Pipes. This can be used as a "back door" if the security database becomes corrupted.

Forced Database Writes

By default, the InterBase Workgroup Server for Windows NT performs forced writes (also referred to as synchronous writes). When InterBase performs forced writes, it physically writes data to disk whenever it performs an (internal) write operation.

If forced writes are not enabled, then even though InterBase performs an internal write, the data may not be physically written to disk, since the operating system buffers disk writes. If there is a system failure before the data is written to disk, then information might be lost.

Performing forced writes ensures data integrity and safety, but will slow performance. In particular, operations which involve data modification will be slower.

You can manually enable and disable forced writes with Server Manager. For more information about enabling and disabling forced writes using Server Manager, see Enabling Forced Writes in Help. You can also enable and disable forced writes with the GFIX command-line DBA utility. For more information about GFIX, see the GFIX Help.

Shutdown and Logoff

If the Windows NT console is shut down or logged off while an application is connected to an InterBase database, a dialog box will appear warning that data could be lost. If you choose to shut down or log off anyway, then the active database can potentially have orphan pages.

You can mend orphan pages with Server Manager or the GFIX command-line DBA utility. For more information about Server Manager, see the Windows Client User's Guide. For more information about GFIX, see "Validating and a Database" in Help.

InterBase Information Roadmap

The InterBase documentation is an integrated system designed for all levels of users of InterBase Server and Local InterBase. Information about using InterBase is provided in three forms: print documentation, online Help on Windows 95 and Windows NT, and Borland OnLine (http://www.borland.com).

Information about...	Source
Installing InterBase	Online Help during install (PC platforms)
	Print: Installing and Running on... (Unix)
	WWW: Borland OnLine (Unix) (Platform-specific information)
Remote configuration	Online Help (PC platforms)
Conceptual overview (white papers)	Borland OnLine
Supported features	Online Help (PC platforms)
	Print: *Getting Started...*
Learning to use SQL	Online Help tutorial (PC platforms)
Database Design	Online Help (PC platforms)
	Print: *Data Definition Guide*
SQL Syntax	Online Help (PC platforms)
	Print: *Language Reference*
Windows ISQL	Online Help (PC platforms)
	Print: *Getting Started...*
Command-Line ISQL	Online Help (PC platforms)
	Print: *Installing and Running on... (Unix)*
Server Manager	Online Help (PC platforms)
	Print: *Getting Started....*
Command-Line DBA utilities	Online Help (PC platforms)
	Print: *Installing and Running on...(Unix)*
Application development overview	Print: *Getting Started...*
Embedded SQL/DSQL applications	Online Help (PC platforms)
	Print: *Programmer's Guide*
API applications	Online Help (PC platforms)
	Print: *API Guide*
Supported compilers	online readme (PC platforms)
	Borland OnLine (Unix)
Example database	/examples (described in online Help)
Example programs	/examples
Borland client development tools	Print and online Help for Borland products, such as Delphi, Visual dBASE, Borland C++, and Paradox

Requesting a Live Result Set

A TTable component always returns a live result set to an application. That is, the dataset is connected to the database, and a user can change it directly with any data controls.

A TQuery can return two kinds of result sets:

- Live result sets

  Users can edit data in the result set with data controls, as with TTable components. Any changes are sent to the database when a post occurs.

- Read-only result sets

  Users cannot edit data in the result set with data controls.

By default, a TQuery always returns a read-only result set.

To receive a live result set from TQuery:

1. Set the TQuery RequestLive property to True.
2. Use SELECT syntax that conforms to the guidelines listed in Local SQL Syntax Requirements for a Live Result Set. and Remote Server SQL Syntax Requirements for a Live Result Set

If you request a live result set, but the syntax does not conform to the requirements, the BDE returns a read-only result set (for local SQL) or an error return code (for passthrough SQL). When a TQuery returns a live result set, Delphi sets the CanModify property to True.

RequestLive	CanModify	Type of result set
False	False	Read-only result set
True, and SELECT syntax meets requirements	True	Live result set
True, and SELECT syntax does not meet requirements	False	Read-only result set

If you need to update the data in a read-only result set, use a separate TQuery to construct an UPDATE statement. By setting the parameters of the update query based on the data retrieved in the first query, you can perform the desired update operation.

Local SQL Syntax Requirements for a Live Result Set

For queries using the local SQL engine, the Borland Database Engine offers expanded support for both single table and multi-table updateable queries. The local SQL engine is used when a query contains one or more local table (dBASE and Paradox), or one or more remote server tables prefaced by its alias. A local table has an alias of type STANDARD. For more details on the local SQL engine, see LOCALSQL.HLP which is installed by default to \Borland\Common Files\BDE.
These restrictions apply to updates:

- Linking fields cannot be updated
- Index switching will cause an error

Restrictions on live queries

Single table queries or views are updateable provided that:

- There are no JOINs, UNIONs, INTERSECTs, or MINUS operations.
- There is no DISTINCT key word in the SELECT.
- There are no aggregate operations.
- The table referenced in the FROM clause is either an updateable base table or an updateable view.
- There is no GROUP BY or HAVING clause.
- There are no subqueries.
- Any ORDER BY clause can be satisfied with an index.

Restrictions on live joins

Live joins depend upon composite cursors. Currently, only 2-table live joins are supported. Live joins may be used only if:

- All joins are left-to-right outer joins.
- All join are equi-joins.
- All join conditions can be satisfied by indexes (for Paradox and dBASE)
- Output ordering is not defined.
- Each table in the join is a base table.
- The query contains no elements listed above that would prevent single-table updatability.

Remote Server SQL Syntax Requirements for a Live Result Set

A query of a server table using passthrough SQL is restricted according to the SQL-92 standard and any further server-specific limitations. In some cases, a server may “enhance” the SQL-92 standards. Consult your server documentation for syntax requirements specific to your server.

These restrictions apply to updates:

- Linking fields cannot be updated
- Index switching will cause an error

Restrictions on live queries

A query of a server table using passthrough SQL can return a live result set if:

- There is no DISTINCT key word in the SELECT.
- Everything in the SELECT clause is a simple column reference or a calculated field, no aggregation is allowed.
- The FROM clause has only one table reference, and is either an updateable base table or an updateable view.
- There is no GROUP BY or HAVING clause.
- There are no subqueries that reference the table in the FROM clause and no correlated subqueries.

A unique index is required, except when the table is from an Oracle server.

SQLSpecifying Parameters for Dynamic SQL

See Also

A dynamic SQL statement (also called a parameterized query) contains parameters that can vary at run time.

To specify parameters at design time,

1. Open the Parameters editor by right-clicking a TQuery component and selecting Parameters editor.
2. Select the desired data type for the parameter in the Data type combo box.
3. Enter a value in the Value text field, or select Null Value to set the parameter's value to Null.
4. Choose OK to prepare the query and bind values to the parameters.

When you set the Active property of the TQuery to True, the results of the SQL query with the specified parameter values appear in any data controls connected to the TQuery.

To specify parameter values at run time, do any of the following:

- Set the TQuery Params property, using the order that the parameters appear in the SQL statement.

  See Assigning Parameter Values With the Params Property for more information.
- Use the ParamByName method, using the parameter names specified in the SQL statement.

  See Assigning Parameter Values With the ParamByName Function for more information.
- Set the DataSource property to assign values from another dataset for columns that match the names of parameters that have no values.

  See Assigning Parameter Values With the Data Source Property for more information.

See Also

Optimizing a Parameterized Query
Assigning Parameter Values With the Params Property
Assigning Parameter Values With the ParamByName Function
Assigning Parameter Values With the Data Source Property
Example of Dynamic SQL

Optimizing a Parameterized Query

See Also

Use the Prepare method to optimize a parameterized query that executes more than once.

Using Prepare is not required; however, it improves performance for dynamic queries that execute multiple times. If you do not explicitly prepare a query, Delphi automatically prepares it each time it executes.

The boolean Prepared property of TQuery indicates if a Query has been prepared. The Parameters editor automatically prepares a Query if you use it to set parameter values at design time.

After a query executes, you must call Close before calling Prepare again.

You should call Prepare once only, for example, in the OnCreate event of a form. After you call Prepare, set parameters using the Params property, then call Open or ExecSQL to execute the query. Before the query executes with different parameter values, call Close, set the parameter values, then execute the query again.

Preparing a query consumes some database resources, so you should unprepare it with the UnPrepare method when you are done using it. When you change the text of a query at run time, Delphi automatically closes and unprepares it.

See Also

Specifying Parameters for Dynamic SQL
Assigning Parameter Values With the Params Property
Assigning Parameter Values With the ParamByName Function
Assigning Parameter Values With the Data Source Property
Example of Dynamic SQL

Assigning Parameter Values With the Params Property

See Also

Delphi creates a Params array to contain the parameters of a dynamic SQL statement when a query is prepared.
Params is a zero-based array of TParam objects with an element for each parameter in the query. That is, the first parameter is Parms[0], the second Params[1], and so on.
For example, if a TQuery component named Query2 has the following statement for its SQL property:

```
INSERT
  INTO COUNTRY (NAME, CAPITAL, POPULATION)
  VALUES (:Name, :Capital, :Population)
```

An application can use Params to specify the values of the parameters as follows:

```
Query2.Params[0].AsString := 'Lichtenstein';
Query2.Params[1].AsString := 'Vaduz';
Query2.Params[2].AsInteger := 420000;
```

These statements bind the following values:

- "Lichtenstein" is bound to the :Name parameter
- "Vaduz" is bound to the :Capital parameter
- 42000 is bound to the :Population parameter.

See Also

Specifying Parameters for Dynamic SQL
Optimizing a Parameterized Query
Assigning Parameter Values With the ParamByName Function
Assigning Parameter Values With the Data Source Property
Example of Dynamic SQL

Assigning Parameter Values With the ParamByName Function

See Also

The ParamByName function enables you to assign values to parameters based on their names. Instead of providing the ordinal location of the parameter, as you do when you use the Params property, you supply its name.

For example, if a TQuery component named Query2 has the following statement for its SQL property:

```
INSERT
  INTO COUNTRY (NAME, CAPITAL, POPULATION)
  VALUES (:Name, :Capital, :Population)
```

You can use ParamByName to specify values for the parameters as follows:

```
Query2.ParamByName('Name').AsString := 'Lichtenstein';
Query2.ParamByName('Capital').AsString := 'Vaduz';
Query2.ParamByName('Population').AsInteger := 42000;
```

These statements have the same effect as the example in Assigning Parameter Values With the Params Property that uses the Params array directly.

See Also

Specifying Parameters for Dynamic SQL
Optimizing a Parameterized Query
Assigning Parameter Values With the Params Property
Assigning Parameter Values With the Data Source Property
Example of Dynamic SQL

Assigning Parameter Values With the Data Source Property

See Also

Delphi checks a query's DataSource property for parameters that are not bound to values at design time. DataSource specifies the name of a TDataSource component. If DataSource is set, and the unbound parameter names match any column names in the specified DataSource, Delphi binds the current values of those fields to the corresponding parameters. This capability enables you to link queries in an application.

The LINKQRY example application illustrates the use of the DataSource property to link a query in a master-detail form. The form contains a TQuery component (named Orders) with the following statement in its SQL property:

```
SELECT Orders.CustNo, Orders.OrderNo, Orders.SaleDate
    FROM Orders
    WHERE  Orders.CustNo = :CustNo
```

The form also contains:

- A TDataSource named OrdersSource, linked to Orders by its DataSet property.
- A TTable component (named Cust).
- A TDataSource named CustSource linked to Cust.
- Two data grids; one linked to CustSource and the other to OrdersSource.

The DataSource property of Orders is set to CustSource. Because the :CustNo parameter does not have any value assigned to it, Delphi tries to match it with a column name in CustSource, which gets its data from the Customer table through Cust, at run time. Because Cust contains a CustNo column, the current value of CustNo in the Cust table is assigned to the parameter, and the two data grids are linked in a master-detail relationship. Each time the Cust table moves to a different row, the Orders query automatically re-executes to retrieve all the orders for the current customer.

See Also

Specifying Parameters for Dynamic SQL
Optimizing a Parameterized Query
Assigning Parameter Values With the Params Property
Assigning Parameter Values With the ParamByName Function
Example of Dynamic SQL

Example of Dynamic SQL

See Also

The following example uses a dynamic query and provides substitution parameters programmatically:

- Start a new project and place the required controls on the form. For this example, place an edit control, Edit1, a button, Button1, and a data-aware grid control, DBGrid1, on the form. Now create a TQuery component, Query1, and a data source, DataSource1, and set the DataSource property of DBGrid1 to "DataSource1" and the Dataset property of DataSource1 to "Query1."

  Set the DatabaseName property of Query1 to DBDEMOS. Double-click on its SQL property in the Object Inspector and enter the following SQL statement. Remember to precede the substitution parameter with a colon:

```
SELECT * FROM country WHERE name LIKE :CountryName
```

- Prepare the query in the OnCreate event of the form:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   Query1.Prepare;
end;
```

- Provide parameters in response to some event. In this example, double-click on Button1 to edit the OnClick event and use the contents of Edit1.Text as a substitution parameter:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Query1.Close;
   Query1.Params[0].AsString := Edit1.Text;
   Query1.Open;
end;
```

See Also

Specifying Parameters for Dynamic SQL
Optimizing a Parameterized Query
Assigning Parameter Values With the Params Property
Assigning Parameter Values With the ParamByName Function
Assigning Parameter Values With the Data Source Property

Creating Heterogeneous Queries

A query that accesses tables in more than one database is called a heterogeneous query. Although heterogeneous queries are not supported by standard SQL, Delphi supports them if the query syntax conforms to the requirements of local SQL.

A heterogeneous query may join tables on different servers, and even different types of servers. For example, a heterogeneous query might involve a table in an Oracle database, a table in a Sybase database, and a local dBASE table.

To perform a heterogeneous query, you must define a BDE standard alias that references a local directory, and use the alias for the DatabaseName of the query component. You must also define a BDE alias for each of the databases you want to query. In the query text, precede each table name with the alias for its database. See the BDE Configuration Utility help for information on defining aliases.

For example, suppose you define an alias called Oracle1 for an Oracle database that has a CUSTOMER table, and Sybase1 for a Sybase database that has an ORDERS table. You can create a simple heterogeneous query against these two tables as follows:

```
SELECT CUSTOMER.CUSTNO, ORDERS. ORDERNO
FROM :Oracle1:CUSTOMER, :Sybase1:ORDERS
```