Your first application—a brief tutorial

Topic groups

The quickest way to introduce yourself to Delphi is to write an application. This tutorial guides you through the creation of a program that navigates a marine-life database. After setting up access to the database, you'll write an event handler that opens a standard Save As dialog box, allowing you to write information from the database to a file.

Note: This tutorial is only for the Professional and Enterprise versions. It sets up database access which requires features not available on the Standard version of Delphi.

Starting a new application

Topic groups

Before beginning a new application, create a folder to hold the source files.

1. Create a folder called Marine in the Projects directory off the main Delphi directory.
2. Open a new project.

   Each application is represented by a *project*. When you start Delphi, it opens a blank project by default. If another project is already open, choose File|New Application to create a new project.

   When you open a new project, Delphi automatically creates the following files.

- *Project1.DPR*: a source-code file associated with the project. This is called a *project file*.
- *Unit1.PAS*: a source-code file associated with the main project form. This is called a *unit file*.
- *Unit1.DFM*: a resource file that stores information about the main project form. This is called a *form file*.

   Each form has its own unit and form files.
3. Choose File|Save All to save your files to disk. When the Save dialog appears, navigate to your Marine folder and save each file using its default name.

   Later on, you can save your work at any time by choosing File|Save All.

   When you save your project, Delphi creates additional files in your project directory. You don't need to worry about them but don't delete them.

When you open a new project, Delphi displays the project's main form, named *Form1* by default. You'll create the user interface and other parts of your application by placing components on this form.

The default form has Maximize and Minimize buttons, a Close button, and a Control menu. If you run the form now by pressing F9, you'll see that these buttons all work. (To return to design mode, click the X to close the form.)

Next to the form, you'll see the Object Inspector, which you can use to set property values for the form and components you place on it.

Setting property values

Topic groups

When you use the Object Inspector to set properties, Delphi maintains your source code for you. The values you set in the Object Inspector are called *design-time* settings.

- Set the background color of *Form1* to Aqua.

  Find the form's *Color* property in the Object Inspector and click the drop-down list displayed to the right of the property. Choose *clAqua* from the list.

Adding objects to the form

Topic groups

The Component palette represents components by icons grouped onto tabbed pages. Add a component to a form by selecting the component on the palette, then clicking on the form where you want to place it. You can also double-click a component to place it in the middle of the form.

Add a *Table* and a *StatusBar* to the form:

1. Drop a *Table* component onto the form.

   Click the Data Access tab on the Component palette. To find the *Table* component, point at an icon on the palette for a moment; Delphi displays a Help hint showing the name of the component.

   When you find the *Table* component, click it once to select it, then click on the form to place the component. The *Table* component is nonvisual, so it doesn't matter where you put it. Delphi names the object *Table1* by default. (When you point to the component on the form, Delphi displays its name—*Table1*—and the type of object it is—*TTable*.)

   Each Delphi component is a *class*; placing a component on a form creates an *instance* of that class. Once the component is on the form, Delphi generates the code necessary to construct an instance object when your application is running.
2. Set the *DatabaseName* property of *Table1* to DBDEMOS. (DBDEMOS is an alias to the sample database that you're going to use.)

   Select *Table1* on the form, then choose the *DatabaseName* property in the Object Inspector. Select DBDEMOS from the drop-down list.
3. Double-click the *StatusBar* component on the Win32 page of the Component palette. This adds a status bar to the bottom of the application.
4. Set the *AutoHint* property of the status bar to *True*. The easiest way to do this is to double-click on *False* next to *AutoHint* in the Object Inspector. (Setting *AutoHint* to *True* allows Help hints to appear in the status bar at runtime.)

Connecting to a database

Topic groups

The next step is to add database controls and a *DataSource* to your form.

1. From the Data Access page of the Component palette, drop a *DataSource* component onto the form. The *DataSource* component is nonvisual, so it doesn't matter where you put it on the form. Set its *DataSet* property to *Table1*.
2. From the Data Controls page, choose the *DBGrid* component and drop it onto your form. Position it in the lower left corner of the form above the status bar, then expand it by dragging its upper right corner.

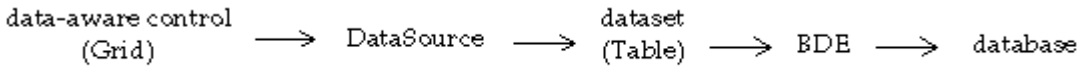
   If necessary, you can enlarge the form by dragging its lower right corner.
3. Set *DBGrid* properties to align the grid with the form. Double-click *Anchors* in the Object Inspector to display *akLeft*, *akTop*, *akRight*, and *akBottom*; set them all to *True*.
4. Set the *DataSource* property of *DBGrid* to *DataSource1* (the default name of the *DataSource* component you just added to the form).

   Now you can finish setting up the *Table1* object you placed on the form earlier.
5. Select the *Table1* object on the form, then set its *TableName* property to BIOLIFE.DB. (*Name* is still *Table1*.) Next, set the *Active* property to *True*.

   When you set *Active* to *True*, the grid fills with data from the BIOLIFE.DB database table. If the grid doesn't display data, make sure you've correctly set the properties of all the objects on the form, as explained in the instructions above. (Also verify that you copied the sample database files into your ...\Borland Shared\Data directory when you installed Delphi.)

   The *DBGrid* control displays data at design time, while you are working in the IDE. This allows you to verify that you've connected to the database correctly. You cannot, however, edit the data at design time; to edit the data in the table, you'll have to run the application.
6. Press F9 to compile and run the project. (You can also run the project by clicking the Run button on the Debug toolbar, or by choosing Run from the Run menu.)

In connecting our application to a database, we've used three components and several levels of indirection. A data-aware control (in this case, a *DBGrid*) points to a *DataSource* object, which in turn points to a dataset object (in this case, a *Table*). Finally, the dataset (*Table1*) points to an actual database table (BIOLIFE), which is accessed through the BDE alias DBDEMOS. (BDE aliases are configured through the BDE Administrator.)



This architecture may seem complicated at first, but in the long run it simplifies development and maintenance. For more information, see Developing database applications.

Adding support for a menu and a toolbar

Topic groups

When you run your project, Delphi opens the program in a window like the one you designed on the form. The program is a full-fledged Windows application, complete with Minimize, Maximize, and Close buttons and a Control menu. You can scroll through the BIOLIFE data in the grid.

Though your program already has a great deal of functionality, it still lacks many features usually found in Windows applications. For example, most Windows applications implement menus and toolbars to make them easy to use.

In this section, you'll prepare your application for additional graphical-interface elements by setting up an *ActionList* component. While you can create menus, toolbars, and buttons without using action lists, action lists simplify development and maintenance by centralizing responses to user commands.

1. Click the X in the upper right corner to close the application and return to the design-time view of the form.
2. From the Win32 page of the Component palette, drop an *ImageList* onto the form. This is a nonvisual component, so it doesn't matter where you place it. The *ImageList* will contain icons that represent standard actions like *Cut* and *Paste*.
3. From the Standard page of the Component palette, drop an *ActionList* onto the form. This is another nonvisual component.
4. Set the action list's *Images* property to *ImageList1*.
5. Double-click the action list to display the Action List editor.
6. Right-click on the Action List editor and choose New Standard Action. The Standard Actions list box is displayed.
7. Select the following actions: *TDataSetFirst*, *TDataSetLast*, *TDataSetNext*, *TDataSetPrior*, *TEditCopy*, *TEditCut*, and *TEditPaste*. (Use the *Ctrl* key to select multiple items.) Then click OK.
8. Click on the X to close the Action List editor.

You've added standard actions. Now you're ready to add the menu and toolbar.

Adding a menu

Topic groups

In this section, you'll add a main menu bar with three drop-down menus—File, Edit, and Record—and you'll add menu items to each one using the standard actions in the action list.

1. From the Standard page of the Component palette, drop a *MainMenu* component onto the form. It doesn't matter where you place it.
2. Set the main menu's *Images* property to *ImageList1*.
3. Double-click the menu component to display the Menu Designer.
4. Type &File to set the *Caption* property of the first top-level menu item and press *Enter*.
5. Type &Save and press *Enter* to create a Save menu item under File.
6. Type a hyphen in the next item under the File menu and press *Enter* to create a separator bar on the menu.
7. Type E&xit and press *Enter* to create an Exit menu item under File.
8. Click on the second top-level menu item (to the right of File), type &Edit, and press *Enter*. The first menu item under Edit is selected.
   - In the Object Inspector, set its *Action* to *EditCut1* and press *Enter*. The item's caption is automatically set to *Cut*.
   - Select the next menu item (under *Cut*) and set its *Action* to *EditCopy1*.
   - Select the next menu item and set its *Action* to *EditPaste1*.
9. Click on the third top-level menu item (to the right of Edit), type &Record as its caption, and press *Enter*. The menu item under Record is selected.
   - In the Object Inspector, set its *Action* to *DataSetFirst1*.
   - Select the next menu item and set its *Action* to *DataSetPrior1*.
   - Select the next menu item and set its *Action* to *DataSetNext1*.
   - Select the next menu item and set its *Action* to *DataSetLast1*.
10. Click on the X to close the Menu Designer.

Press F9 to run your program and see how it looks.

Close the application when you're ready to continue.

Adding a toolbar

Topic groups

1. On the Win32 page of the Component palette, double-click the *ToolBar* to add it to the form.
   - Set the toolbar's *Indent* property to 4.
   - Set its *Images* property to *ImageList1*.
   - Set *ShowHint* to *True*.
2. Add buttons to the toolbar.
   - With the toolbar selected, right-click and choose New Button three times.
   - Right-click and choose New Separator.
   - Right-click and choose New Button four more times.
3. Assign actions to the first set of buttons.
   - Select the first button and set its *Action* to *EditCut1*.
   - Select the second button and set its *Action* to *EditCopy1*.
   - Select the third button and set its *Action* to *EditPaste1*.
4. Assign actions to the second set of buttons.
   - Select the first button and set its *Action* to *DataSetFirst1*.
   - Select the second button and set its *Action* to *DataSetPrior1*.
   - Select the third button and set its *Action* to *DataSetNext1*.
   - Select the last button and set its *Action* to *DataSetLast1*.
5. Press F9 to compile and run the project.

Check out the toolbar. The First, Prior, Next, and Last buttons work. Select text within a cell in the grid; the Cut, Copy, and Paste buttons work as well.

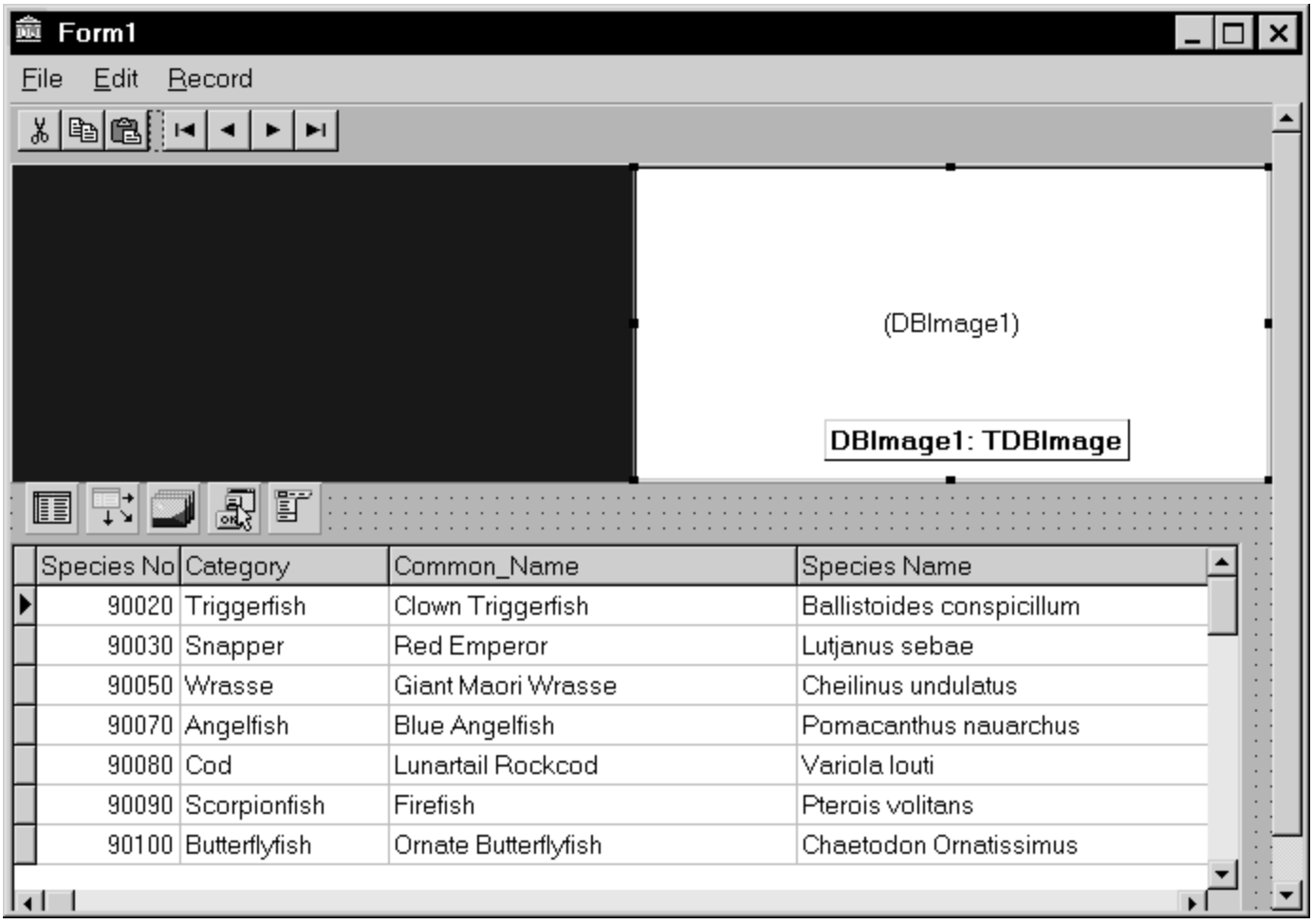
Close the application when you're ready to continue.

Displaying images

Topic groups

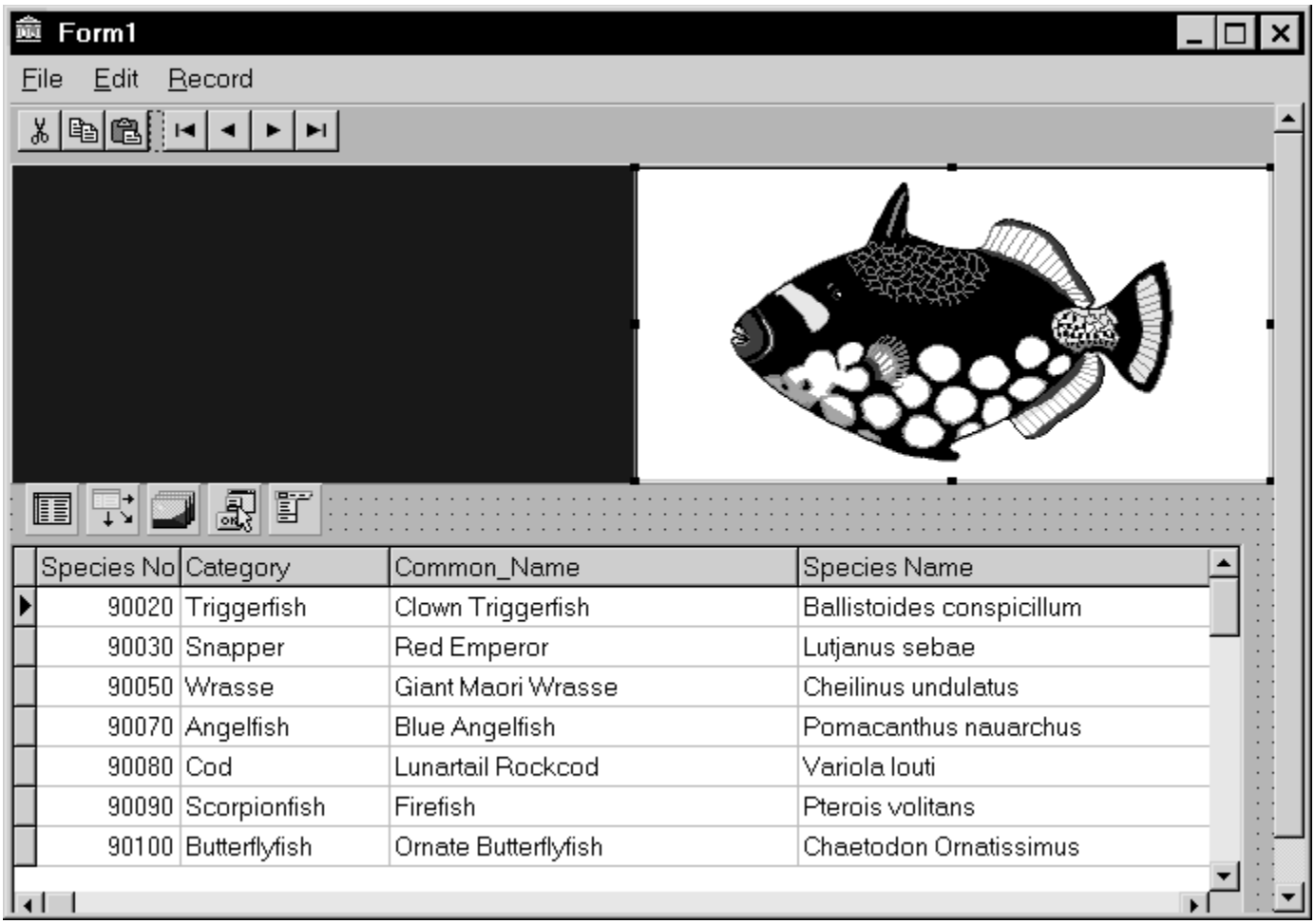
Each record in the BIOLIFE database has a picture associated with it. In this section, we'll expand our application to display pictures.

1. From the Standard page of the Component palette, drop a *Panel* component onto the form below the toolbar. Delphi names this *Panel1* by default.
2. In the Object Inspector, delete the *Panel1* string from the panel's *Caption* property. Leave the *Caption* property blank.
3. Align *Panel1* to the top of the form by setting its *Align* property to *alTop*. Next, drag the bottom of the panel down so it fills the portion of the form between the toolbar and the grid.
4. Set the panel's color to *clBlue*.
5. From the Data Controls palette page, drop a *DBImage* component on top of *Panel1* and set its *Align* property to *alRight*. Size the *DBImage* by dragging out its left side so your form resembles the one shown in the following figure.



6. Set the *DataSource* property of *DBImage* to *DataSource1*. Then set its *DataField* property to *Graphic*. (In the Object Inspector, the drop-down list next to *DataField* shows the fields in the BIOLIFE table. *Graphic* is one of the field names.)

   As soon as you set *DataField* to *Graphic*, the *DBImage* component displays the image of a fish corresponding to the first record of the table. This shows that you have correctly hooked up to the database.



7 Press F9 to compile and run your application.

Close the application when you're ready to continue.

Adding text and memo objects

Topic groups

In this section, you'll add two components that display individual text fields from the database.

1. Select *Panel1*.
2. From the Data Controls page of the Component palette, drop a *DBMemo* component onto *Panel1* and position it so it occupies the upper left corner of the panel (below the menus and toolbar).
3. Resize the *DBMemo* by dragging its lower right corner. Extend the right edge of the *DBMemo* until it touches the left edge of the *DBImage*. Extend the bottom of the *DBMemo* to within a half inch or so of the bottom of *Panel1*.
4. In the Object Inspector, set the following properties for the *DBMemo*.
   - Set *DataSource* to *DataSource1*.
   - Set *DataField* to *Notes* (information about the fish appears).
   - Set *ScrollBars* to *ssVertical*.
5. Drop a *DBText* component on *Panel1* under the *DBMemo* object. Enlarge the *DBText* so it fills the area under the *DBMemo*, then set its properties as follows.
   - Set *DataSource* to *DataSource1*.
   - Set *DataField* to *Common_Name*.
   - Set *Alignment* to *taCenter*.
6. Customize the *Font* property of the *DBText* component using the Font editor.

   The Font editor is a property editor that you can access through the Object Inspector. Select the *Font* property in the Object Inspector; an ellipsis button appears on the right side of the property setting. Click the ellipsis button to display the Font editor.

   Modify the following *DBText* settings using the Font editor, then click OK.
   - Set the *Font Style* to *Bold*.
   - Set the *Color* to *Silver*.
   - Set the *Size* to *12*.
7. Compile and run your application by pressing F9.

You can view and edit the data in the *DBMemo* component. The *DBText* component displays data for reading only.

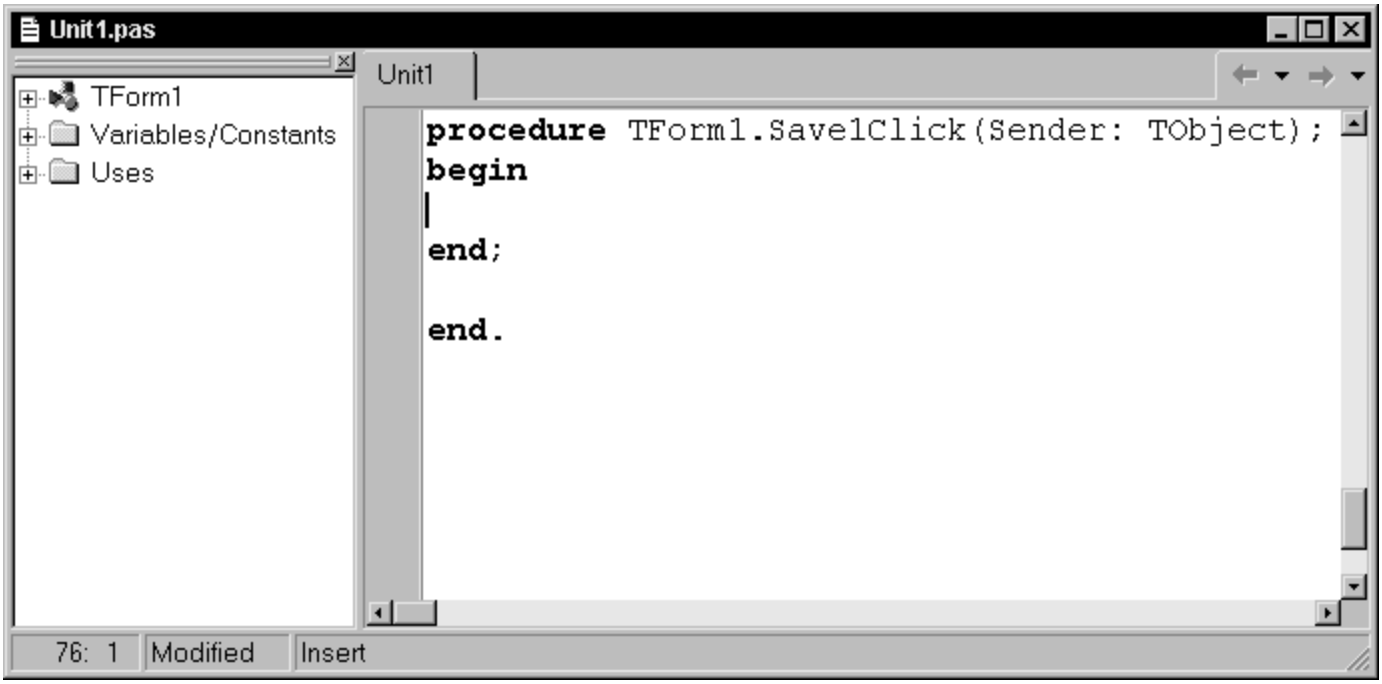
Close the application when you're ready to continue.

Writing an event handler

Topic groups

Up to this point, you've developed your application without writing a single line of code. By using the Object Inspector to set property values at design time, you've taken full advantage of Delphi's RAD environment. In this section, however, you'll write procedures called *event handlers* that respond to user input while the application is running. You'll connect the event handlers to menu items, so that when a menu item is selected your application executes the code in the handler.

1. From the Dialogs page of the Component palette, drop a *SaveDialog* component onto the form. This is a nonvisual component, so it doesn't matter where you place it. Delphi names it *SaveDialog1* by default. (When *SaveDialog*'s *Execute* method is called, it invokes a standard Windows dialog for saving files.)
2. From the menu on your form, choose File|Save. Delphi creates a skeleton event handler for the event that occurs at runtime when the user selects Save from the File menu. The Code editor opens with the cursor inside the event handler.



This event handler is attached to the *OnClick* event of the main menu's first menu item. The menu item is an instance of the class *TMenuItem*, and *OnClick* is its *default event*. Hence the *Save1Click* method is a *default event handler*.

3. Complete the event handler by adding the code shown below in the **var** section and between the outermost **begin** and **end**.

```
procedure TForm1.Save1Click(Sender: TObject);
var
  i: integer;
begin
  SaveDialog1.Title := Format('Save info for %s', [DBText1.Field.AsString]);
  if SaveDialog1.Execute then
  begin
    with TStringList.Create do
    try
      Add(Format('Facts on the %s', [DBText1.Field.AsString]));
      Add(#13#10);
      for i := 1 to DBGrid1.FieldCount-3 do
      Add(Format('%s : %s',
        [DBGrid1.Fields[i].FieldName,
        DBGrid1.Fields[i].AsString]));
```

```
      Add(Format(#13#10+'%s'+#13#10,[DBMemo1.Text]));
      SaveToFile(SaveDialog1.FileName);
    finally
      Free;
    end;
  end;
end;
```

This event handler calls the *Execute* method in the *SaveDialog* component. When the dialog box opens and the user specifies a file name, it saves fields from the current database record into a file.

4 To add code for the Exit command, choose File|Exit. Delphi generates another skeleton event handler and displays it in the editor.

```
procedure TForm1.Exit1Click(Sender: TObject);
begin

end;
```

Right where the cursor is positioned (between **begin** and **end**), type

```
Close;
```

5 Choose File|Save All to save your work. Then press F9 to run the application.

You can exit the program using the now functional File|Exit command.

Most components on the Component palette have events, and most components have a default event. A common default event is *OnClick*, which gets called whenever the component is clicked; for example, if you placed a *Button* component (*TButton*) on a form, you would almost certainly write an *OnClick* event handler for it. When you double-click certain objects on a form, Delphi creates a skeleton handler for the default event.

You can also access all of a component's events through the Object Inspector. Select an object on a form, then click the Events tab on the object Inspector; you'll see a list of the object's events. To create a skeleton handler for any event, double-click in the space to its right.

For more information about events and event handlers, see Developing the application user interface.

Related topic groups

Quick Start Tutorial

- Your first application: a brief tutorial

Your first application: a brief tutorial

Related topic groups

- Your first application--a brief tutorial
- Starting a new application
- Setting property values
- Adding objects to the form
- Connecting to a database
- Adding support for a menu and a toolbar
- Adding a menu
- Adding a toolbar
- Displaying images
- Adding text and memo objects
- Writing an event handler

Connection String dialog box

The Connection String dialog box lets you specify the connection string used to connect an ADO data component with an ADO data store. You can type the connection string, build it using an ADO-supplied dialog box, or place the string in a file.

Choosing between using a data link file and a connection string

To use a data link file to establish the connection to the data store, click the **Use Data Link File** checkbox. Select or Enter the name of a data link file, or click the Browse button to use a File dialog box to locate the file.

To use a string to establish the connection to the data store, click the **Use Connection String** checkbox. Enter the connection string with the connection information into the Connection String edit box. Or, click the Build button to invoke an ADO dialog box that takes you through setting up and testing the connection.

Connection string information

A connection string consists of one or multiple connection parameters that define a connection. When multiple parameters are specified, separate individual parameters in the list using semicolons.

ADO supports the following four arguments for connection strings. Any other arguments (such as a user ID and password) are not processed by ADO and are passed on to the provider.

Argument	Meaning
Provider	The name of the provider to use for the connection.
File name	The name of a file containing connection information.
Remote Provider	The name of the provider to use for a client-side connection.
Remote Server	The path name of the server to use for a client-side connection.

A connection string can contain parameters other than those listed above, parameters not directly supported by ADO, but which are involved in accessing a server or provider. Such parameters might include user ID, login password, the name of a default database, persistent security information, ODBC data source names, connection timeout values, and locale identifiers. These parameters and their values are specific to particular providers, servers, and ODBC drivers and not to either ADO or Delphi. For specific information on them, consult the documentation for the provider, server, or ODBC driver.

OK button

Click the OK button to accept the current connection information and return to program design in the IDE.

Cancel button

Click the Cancel button to abort the connection string construction process. Any changes made to the connection string since the dialog was invoked are lost. The connection string is returned to the state it was in prior to invoking the dialog (empty if there were no previous contents).

Help button

Click the Help button to open the Delphi help at this topic. This topic can also be viewed by pressing F1 when the dialog is open.

CommandText editor

The CommandText editor lets you construct the command for an ADO dataset component. A multi-line editing control in the dialog lets you manually edit the command or to watch as the command is built by the other controls in the dialog. Lists of available tables and table columns are provided and, when an item is used, the name of the metadata object is automatically inserted into the command.

SQL edit control

Displays the command (SQL statement) for the CommandText property of the ADO dataset or command component. The statement displayed in this editing control may be edited manually or the other lists and buttons (see below) may be used to build the statement. You can edit the statement after it is automatically built.

Tables list

This listbox displays the names of tables available in the current database. Select one for use by double-clicking its name in the list or by highlighting the name and clicking the **Add Table to SQL** button. When a table is selected, it is added to the command (SQL statement) displayed in the SQL edit control in the right half of the dialog.

If no SQL statement is in the SQL edit control when a table is selected, a framework of a statement is added using the selected table name. This statement template includes a SELECT clause (with no columns) and a FROM clause (using the name of the selected table). If a statement was already in the SQL edit control, the table name is inserted at the insertion point marked by the arrow.

Fields list

This listbox displays the names of the columns available in the table currently highlighted in the Tables list. Select one for use by double-clicking its name in the list or by highlighting the name and clicking the **Add Field to SQL** button. When a column is selected for use in this manner, it is added to the command (SQL statement) displayed in the SQL edit control in the right half of the dialog.

Multiple columns may be selected at the same time using the Windows convention of Shift-clicking on each item to select. After all of the multiple columns have been selected in the list, clicking the **Add Field to SQL** button adds all of them to the SQL statement. The multiple column names are added to the statement as a comma-separated list.

If no SQL statement is in the SQL edit control when a column is selected or used, a statement framework based on the selected column name is added. This statement template includes a SELECT clause (with the selected column). It does not add a FROM clause, even if there currently is none. If a statement was already in the SQL edit control, the column name is inserted at the insertion point marked by the arrow. Commas are automatically added as needed in a columns list.

Constructing SQL statements

This dialog allows the construction of only rudimentary SQL statements. The only clauses added are the SELECT and FROM clauses and then they are only added if needed. Other clauses (WHERE, GROUP BY, HAVING, ORDER BY, and so on) must be manually added in the SQL edit control.

Once any of these other clauses have been added to the statement, tables, and columns can be added using the dialog lists. Place the caret at the point the table or column name is to be added and then add the metadata object name as described above.

Table and column correlation names are not automatically added. If column names are duplicated across multiple tables in the statement, manually add the correlation names to differentiate between the instances of the same column names.

OK button

Click the OK button to accept the current command text information and return to program design in the IDE.

Cancel button

Click Cancel to abort the command text construction process. Any changes made to the command text since the dialog was invoked are lost. The command text is returned to the state it was in prior to invoking the dialog (or empty if there were no previous contents).

Help button

Click the Help button to open the Delphi help at this topic. This topic can also be viewed by pressing F1 when the dialog box is open.

ActiveX Control wizard

See also

Use the ActiveX Control Wizard to add an ActiveX control or Active Form to an ActiveX Library project. The wizard creates an ActiveX Library project (if needed), a type library, a form, an implementation unit, and a unit containing corresponding type library declarations. Note that ActiveX controls need an ActiveX library to expose their interfaces and method arguments to client applications.

To bring up the ActiveX Control wizard:

1 Choose File|New to open the New Items dialog box.

2 Choose the tab labeled ActiveX.

3 Select the Active Form or ActiveX Control icon.

In the Wizard, you can specify the following:

VCL ClassName

Specify the class on which your ActiveX control is based. For example, to create an ActiveX control that allows client applications to use a TButton object, specify TButton. When creating Active forms, this control is disabled because active forms are always based on TActiveForm.

New ActiveX Name

The wizard provides a default name that clients will use to identify your ActiveX control or Active form. Change this name to provide a different OLE class name.

Implementation Unit

The wizard a default name for the unit that contains the code that implements the behavior of the ActiveX control or Active form. You can accept the default name or type in a new name.

Project Name

ActiveX controls and Active forms must be added to an ActiveX library project. If you currently don't have an ActiveX Library project open, a fourth field allows you to specify which ActiveX Library project to add the ActiveForm control. A default Project Name is provided. This control is disabled if you have an ActiveX Library open.

Threading Model

Choose the threading model to indicate how COM serializes calls to your ActiveX control.
Note: The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.

ActiveX control options

- **Make Control Licensed**

  Making a control licensed ensures that users of the control can't open it either for design purposes or at runtime unless they have a license key for the control. With Make Control Licensed checked, the wizard creates a key for the control that is stored in a .LIC file with the same name as the project. The user of the control must have a copy of the .LIC file to open the control in a development environment. Each control in the project that has Make Control Licensed checked will have a separate key entry in the .LIC file.

  Note: For most containers, adding the control to an application at design time embeds the corresponding runtime license in the executable. An exception is Internet Explorer 4 and later, which requires License package (.LPK) files. LPK files can be generated using LPK_TOOL.EXE, a utility available in Microsoft's Internet SDK. For more information on the license-creation tool and how it is used, visit http:\\support.microsoft.com and search for "LPK".

- **Include Version Information**

  This option includes version information in the .OCX file. Adding this resource to your control allows your control to expose information about the module, such as copyright and file description, which can be viewed in the browser. Version information can be specified by choosing Project|Options and

selecting the Version Info page. Some clients, such as Visual Basic 4.0, require version information for registering controls.

- **Include About Box**
  When this box is checked, an About box is included in the project. The user of the control can display the About box in a development environment. The About box is a separate form that you can modify. By default, the About box includes the name of the ActiveX control, an image, copyright information, and an OK button.

Automation Object wizard

See also

Use the Automation Object wizard to add an Automation server to an ActiveX Library project. The wizard creates an ActiveX Library project (if needed), a type library, and the definition for the Automation object. After exiting the wizard, you can expose the properties and methods of the interface through the type library.

To bring up the Automation Object wizard:

1. Choose File|New to open the New Items dialog box.
2. Choose the tab labeled ActiveX.
3. Select the Automation Object icon.

In the Wizard, you can specify the following:

CoClass Name

Specify the class whose properties and methods you want to expose to client applications. (Delphi prepends a T to this name.)

Instancing

Specify an instancing mode to indicate how your Automation server is launched.

When your COM application creates a new COM object, it can have any of the following instancing types:

Instancing	Meaning
Internal	The object can only be created internally. An external application cannot create an instance of the object directly. For example, a word processor application may have a document object that can only be created by calling a method of the application that can create the document object.
Single Instance	Allows only a single COM interface for each executable (application), so creating multiple instances results in creating multiple applications. Single instance specifies that once an application has connected to the object, it is removed from public view so that no other applications can connect to it. This option is commonly used for multiple document interface (MDI) applications. When a client requests services from a single instance object, all requests are handled by the same server. For example, any time a user requests to open a new document in a word processor application, typically the new document opens in the same application process.
Multiple Instance	Specifies that multiple applications can connect to the object. Any time a client requests service, a separate instance of the server gets invoked. (That is, there can be multiple instances in a single executable.) Any time a user attempts to open the Windows Explorer, a separate Explorer is created.

Note: When your Automation object is used only as an in-process server, instancing is ignored.

Threading Model

Choose the threading model to indicate how COM serializes calls to your Automation object's interface. Automation Objects can have one of the following:

Single	Your code has no thread support. Only one client thread can be serviced at a time.
Apartment	Clients can call the object's methods only from the thread on which the object was created. Different objects from the same server can be called on different threads, but each object is called only from that one thread.
Free	Your code is fully thread-safe. Objects can handle any number of threads at any

	time.
Both	Your server supports clients that use either the Apartment or Free threading model.

Note: The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.

Generate Event support code

Check this box to tell the wizard to implement a separate interface for managing events on your Automation object.

COM object wizard

See also

Use the COM object wizard to create a simple COM object such as a shell extension. Before you create a COM object, create or open the project for an application containing functionality that you want to implement. The project can be either an application or ActiveX library, depending on your needs.

To bring up the COM object wizard,

1. Choose File|New to open the New Items dialog box.
2. Select the tab labeled ActiveX.
3. Double-click the COM object icon.

In the wizard, specify the following:

ClassName	Specify the class whose properties and methods you want to expose to client applications. (Delphi prepends a T to this name.)
Instancing	Specify an instancing mode to indicate how your COM object is launched. **Note:** When your COM object is used only as an in-process server, instancing is ignored.
Threading model	Choose the threading model to indicate how client applications can call your COM object's interface. **Note:** The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.
Implemented interfaces	Specify the names of the COM interfaces that you want this COM object to implement.
Description	Enter a description of the COM object you are creating.
Include Type Library	Check this box to generate a type library for this object. A type library contains type information that allows you to expose any object interface and its methods and properties to client applications.
Mark interface OleAutomation	Check this box to allow type library marshaling, especially for local servers. This flag lets you avoid writing your own proxy-stub DLL for custom marshaling. **Note:** When marking an interface as OleAutomation, You must ensure that it uses OLE Automation compatible types.

Active Server Object wizard

Use the Active Server Object wizard to create a simple active server object. Before you create an Active Server Object, create or open the project for an application containing functionality that you want to implement. The project can be either an application or ActiveX library, depending on your needs.

To bring up the Active Server Object wizard,

1. Choose File|New to open the New Items dialog box.
2. Select the tab labeled ActiveX.
3. Double-click the Active Server Object icon.

In the wizard, specify the following:

CoClassName

Specify the name for the object that you want to implement.

Instancing

Specify an instancing mode to indicate how your Active server is launched.

Instancing	Meaning
Internal	The object can only be created internally. An external application cannot create an instance of the object directly. For example, a word processor application may have a document object that can only be created by calling a method of the application that can create the document object.
Single Instance	Allows only a single COM interface for each executable (application), so creating multiple instances results in creating multiple applications. Single instance specifies that once an application has connected to the object, it is removed from public view so that no other applications can connect to it. This option is commonly used for multiple document interface (MDI) applications. When a client requests services from a single instance object, all requests are handled by the same server. For example, any time a user requests to open a new document in a word processor application, typically the new document opens in the same application process.
Multiple Instance	Specifies that multiple applications can connect to the object. Any time a client requests service, a separate instance of the server gets invoked. (That is, there can be multiple instances in a single executable.) Any time a user attempts to open the Windows Explorer, a separate Explorer is created.

Note: When your active server object is used only as an in-process server, instancing is ignored.

Threading Model

Choose the threading model to indicate how COM serializes calls to your active server object's interface. Active server objects can use the following threading models:

Model	Description
Single	Your code has no thread support. Only one client thread can be serviced at a time.
Apartment	Clients can call the object's methods only from the thread on which the object was created. Different objects from the same server can be called on different threads, but each object is called only from that one thread.
Free	Your code is fully thread-safe. Objects can handle any number of threads at any time.
Both	Your server supports clients that use either the Apartment or Free threading model.

Note: The threading model you choose determines how the object is registered. You must make sure that your object implementation adheres to the model selected.

Active Server Type

Option	Description
Page-level event methods (OnStartPage/ OnEndPage)	Creates an active server object that implements OnStartPage and OnEndPage. These methods are called by the web server on initialization and finalization of the page. This style of active server objects is available for use with IIS 3 and IIS 4. Active server objects used by IIS 5 should be created using the Object Context option.
Object Context	Creates an active server object that uses MTS functionality to retrieve the correct instance data of your object. Recommended for use with IIS 5 (may also work with IIS 4 and MTS).

Options

Option	Description
Generate a template test script for this object	Generates a simple .ASP page that creates a Delphi object based off its ProgID. For an example, see the ActiveX\ASP\PageLevel\HelloWorld or ActiveX\ASP\ObjContext\HelloWorld demos.
Generate Event support code	Implements a separate interface for managing events on your active server object.

Breakpoint List

See also

The Breakpoint List shows all breakpoints currently set in the loaded project. (If no project is loaded, it shows all breakpoints set in the active Code editor page or in the CPU window.) The Breakpoint List shows:

For Source Breakpoints:	The file name and line number location along with any condition and pass count associated with each breakpoint.
For Address Breakpoints:	The file name and line number + a hex offset. The offset is the number of bytes from the source line the address breakpoint is. If no corresponding file line number is found, a raw address is used.
For Data Breakpoints:	The data name or address location and length.

The Breakpoint List also shows conditions associated with the breakpoint and the pass count (including the total pass count and the current pass count).

The Breakpoint List also lets you add, edit, delete, and enable or disable breakpoints. A breakpoint appears grayed if it is either disabled or invalid.

To display the Breakpoint List, choose View|Debug Windows|Breakpoints. Right-click in the Breakpoint List to display the context menu commands.

Breakpoint List context menu

See also

Use the Breakpoint List context menu to access commands that enable you to manipulate breakpoints. The context menu offers two sets of commands depending on whether or not you have highlighted a listed breakpoint.

The following commands appear on the Breakpoint List context menu:

Add	Opens dialog boxes where you can create new breakpoints.
Delete All	Removes all breakpoints
Disable All	Disables all enabled breakpoints
Enable All	Enables all disabled breakpoints
Disable Group	Enables the breakpoint group you select
Enable Group	Disables the breakpoint group that you select
Dockable	Toggles whether the Breakpoint List is dockable

The following commands are available when you right-click on a defined breakpoint:

Enabled	Enables a disabled breakpoint
Delete	Removes a 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
Properties	Opens dialog boxes, where you can modify breakpoints
Dockable	Toggles whether the Breakpoint List is dockable

To display the Breakpoint List context menu:

Choose View|Breakpoints to display the Breakpoint List, then do one of the following:

- Right-click anywhere in the Breakpoint List.
- Press Alt+F10 when the Breakpoint List is active.

Add (Breakpoint List context menu)

See also

Choose Add from the Breakpoint List context menu to open dialog boxes where you can create new breakpoints.

Add opens a different dialog box, as follows:

For Source Breakpoints:	Opens the Add Source Breakpoint dialog box.
For Address Breakpoints:	Opens the Add Address Breakpoint dialog box.
For Data Breakpoints:	Opens the Add Data Breakpoint dialog box.

An alternate way to perform this command is choose Run|Add Breakpoint.

Delete **(Breakpoint List context menu)**

See also

Choose Delete from the Breakpoint List context menu to remove the selected 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.

Enabled **(Breakpoint List and Code editor context menu)**

See also

Choose Enabled from the Breakpoint List or Code editor context menu to toggle a breakpoint on or off. The breakpoint is enabled when the option is checked.

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.

View Source **(Breakpoint List context menu)**

See also

Choose View Source from the Breakpoint List context menu to locate a breakpoint in your source code or an address breakpoint in the CPU window.

The View Source command scrolls the Code editor to the location of the source breakpoint that is selected in the Breakpoint List, or it scrolls the CPU window to the location of the address breakpoint that is selected in the Breakpoint List.

Edit Source **(Breakpoint List context menu)**

See also

Choose Edit Source from the Breakpoint List context menu to locate a source breakpoint in your source code or an address breakpoint in the CPU window.

If a source breakpoint is selected in the Breakpoint List, the Edit Source command scrolls the Code editor to the location of the breakpoint and activates the Code editor. If an address breakpoint is selected in the Breakpoint List, the Edit Source command scrolls the CPU window to the location of the breakpoint and activates the CPU window.

Disable All **(Breakpoint List context menu)**

See also

Choose Disable All from the Breakpoint List context menu 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.

Enable All **(Breakpoint List context menu)**

See also

Choose Enable All to enable all disabled breakpoints.

When you set a breakpoint, it is enabled by default. 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.

Disable Group **(Breakpoint List context menu)**

See also

Choose Disable Group from the Breakpoint List context menu and select a group to disable from the submenu to disable that group of breakpoints.

When you disable a breakpoint group, the group remains defined, but the breakpoints in that group are not active while debugging. See Organizing breakpoints into groups.

Enable Group **(Breakpoint List context menu)**

See also

Choose Enable Group and select a group to enable from the submenu.

Enabling a breakpoint group enables all of the breakpoints in that group while . See Organizing breakpoints into groups.

Delete All **(Breakpoint List context menu)**

See also

Choose Delete All from the Breakpoint List context menu 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.

Properties (Breakpoint List context menu)

See also

Choose Properties from the Breakpoint List context menu to open dialogs that allow you to modify or add new breakpoints:

For source breakpoints	Opens the Source Breakpoint Properties
For address breakpoints	Opens the Address Breakpoint Properties.
For data breakpoints	Opens the Data Breakpoint Properties.

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.

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. If debug information is available for a function listed in the window, it is followed by the arguments that were passed when the call was made.

The Call Stack window also shows the names of member functions (or methods). Each member function is prefixed with the name of the class that defines the function.

Call Stack commands

Use the Call Stack context menu to access commands that enable you to examine previous function calls.

The commands on the Call Stack List context menu 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
Stay On Top	Keeps the Call Stack window visible when out of focus
Dockable	Toggles the Call Stack window so that it can be docked or not

To display the Call Stack context menu:

Choose View|Debug Windows|Call Stack to display the Call Stack window, then do one of the following:

- Right-click anywhere in the Call Stack window.
- Press Alt+F10 when the Call Stack window is active.

View Source **(Call Stack context menu)**

See also

Choose View Source from the Call Stack context menu 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 but does not give the Code editor focus.

Edit Source **(Call Stack context menu)**

See also

Choose Edit Source from the Call Stack context menu 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.

Associating actions with breakpoints

See also

In addition to simply pausing your process when encountered, you can associate one or more actions with a breakpoint. You can also organize breakpoints into groups. (See Organizing breakpoints into groups.)

When a breakpoint is encountered, it performs each associated action. For breakpoints that have multiple actions, the actions are performed in the order listed.

The Breakpoint List includes columns that show the actions associated with each breakpoint and its group name (if any). Breakpoint tooltips are displayed when you point at the breakpoint in the gutter of the source code. Along with the pass and condition of the breakpoint, the tooltips show the actions associated with each breakpoint, and its group name (if any).

Breakpoint actions

When setting a breakpoint using one of the Breakpoint Properties dialog boxes, you can associate actions with the breakpoint.

To associate actions with breakpoints:

1. Set a breakpoint by choosing Run|Add Breakpoint and selecting the type of breakpoint to set.
   One of the Breakpoint Properties dialog boxes is displayed.
2. Click Advanced to expand the dialog box.
   The dialog box displays additional fields that can be set for each possible action.
3. Check the actions you want to occur when the breakpoint is encountered.
   Enter the appropriate text in each field for each action you want to associate with the breakpoint.

Action	Description
Break	When checked, halts execution; the traditional and default action of a breakpoint.
Ignore subsequent exceptions	When checked, ignore all subsequent exceptions raised by the current process during the current debug session (the debugger will not stop on any exception). Use this with "Handle subsequent exceptions" as a pair. You can surround specific blocks of code with the Ignore/Handle pair to skip any exceptions which occur in that block of code.
Handle subsequent exceptions	When checked, handle all subsequent exceptions raised by the current process during the current debug session (the debugger will stop on exceptions based on the current exception settings in the Tools| Debugger options (Language and OS exception pages). This action does not mean stop on all exceptions no matter what. Use it to turn on normal exception behavior after another breakpoint disabled normal behavior using the Ignore subsequent exceptions option.
Log message	Writes the specified message in the event log. You specify the message to log.
Eval expression	Evaluates the specified expression and because Log result is checked by default writes the result of the evaluation to the event log. Uncheck Log result to evaluate without logging.
Log result	Becomes enabled when text is entered into Eval expression and is checked by default. If checked, writes the result of the evaluation in the Eval expression to the event log. If unchecked the evaluation is not logged.
Enable group	Enables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.
Disable group	Disables all breakpoints which are members of the specified group.

Select the group name. See Organizing breakpoints into groups.

Organizing breakpoints into groups

See also

You can organize breakpoints into groups. This way you can perform a similar set of actions on all breakpoints within a specific group. To associate actions with breakpoints, see Associating actions with breakpoints.

When setting a breakpoint using one of the Breakpoint Properties dialogs, you make it a member of a group by entering a group name in the Group field. Once your breakpoints are organized into groups, you can disable and enable groups of breakpoints by using the Disable Group and the Enable Group commands on the Breakpoint List context menu (right-click on the Breakpoint List).

To organize breakpoints into groups:

1. Set a breakpoint by choosing Run|Add Breakpoint and selecting the type of breakpoint to set.
   One of the Breakpoint Properties dialog boxes is displayed.
2. Enter a group name in the Group field (or by selecting a known group from the drop down list box).

About the Component palette

See also

Components are the building blocks of every Delphi application, and the basis of the Visual Component Library. Each page tab in the Component palette displays a group of icons representing the components you can use to design your application interface. To add a component to an open form, double-click it, then set its properties and code its event handlers.

The Component palette's Help Hints feature displays 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, choose Show Hints from the Component palette context menu.

Components may be either visual or nonvisual. Each component has specific attributes that enable you to control your application: Properties, Events, and Methods.

To get Help on a specific component, click the component and press F1.

The default page tabs divide components into the following functional groups:

- Standard components
- Additional components
- Win32 components
- System components
- Data Access components
- Data Controls components
- ADO components
- InterBase components
- MIDAS components
- InternetExpress components
- Internet components
- FastNet components
- Decision Cube components
- QReport components
- Dialogs components
- Win 3.1 components
- Samples components
- ActiveX components
- Servers components

Standard page components

See also Other Palette pages

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

 Frames	Opens a dialog box displaying a list of frames included in the current project. Select any frame and click OK. See Working with frames for more information.
 MainMenu	Creates menu bar menus for your form. To access events for items in a main menu, add the MainMenu component to a form and double-click it to open the Menu Designer
 PopupMenu	Creates popup menus that appear when the user right-clicks. To access events for items in a popup menu, add the PopupMenu component to a form and double-click it to open the Menu Designer.
 Label	Displays text that the user cannot select or manipulate, such as title text or control labels. See About Label.
 Edit	Displays an editing area where the user can enter or modify a single line of text. Edit is one of several Text controls.
 Memo	Displays an editing area where the user can enter or modify multiple lines of data. See About Memo.
 Button	Creates a pushbutton control that users choose to initiate actions. See About Button.
 CheckBox	Presents an option that a user can toggle between Yes/No or True/False. 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. See About CheckBox.
 RadioButton	Presents an option that a user can toggle between Yes/No or True/False. Use radio buttons to display a group of choices that are mutually exclusive. Users can select only one radio button in a group. See About RadioButton.
 ListBox	Displays a scrolling list of choices. See ListBoxes and check-list boxes.
 ComboBox	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. See About ComboBox.
 ScrollBar	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. See About Scrollbar.
 GroupBox	Provides a container to group-related options on a form. See About GroupBox.
 RadioGroup	Creates a group box that contains radio buttons on a form. See About RadioGroup.
 Panel	Creates panels that can contain other components on a form. You can use panels to create toolbars and status-lines. See About Panel.
 ActionList	Creates collections of actions that centralize your

application's responses to user actions. See Using action lists.

Additional page components

See also Other Palette pages

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

 BitBtn Creates a button component that can display a bitmap. See Bitmap Buttons.

 SpeedButton Provides a button that can display a glyph but not a caption. Speed buttons can be grouped within a panel to create a tool palette. See Speed Buttons.

 MaskEdit Allows user to enter and edit data, similar to an edit component, but provides a means to specify particular formats, such as a postal code or phone number. MaskEdit shares the properties of other Text controls.

 StringGrid Creates a grid that you can use to display string data in columns and rows. StringGrid.is a superset of the features provided by the DrawGrid component. See String grids.

 DrawGrid Creates a grid that you can use to display data in columns and rows. See Draw Grids.

 Image Displays a bitmap, icon, or metafile. See Adding an image control.

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

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

 ScrollBox Creates a resizeable container that automatically displays scrollbars when necessary. See About ScrollBox.

 CheckListBox Displays a scrollable list similar to ListBox except that each item has a check box next to it. See List boxes and check-list boxes.

 Splitter Add a splitter to a form between two aligned controls to allow users to resize the controls at runtime by clicking and dragging the split line. See About Splitter.

 StaticText A non-editable text component like Label, except that it has its own window handle, which is useful when the component's accelerator key must belong to a windowed control. Use StaticText to provide users with feedback on the current state of the application.

 ControlBar A layout manager for toolbar components. Use a control bar as a convenient docking site for toolbars.

 ApplicationEvents A component that intercepts application-level events. Use as a way to set event handlers for application events using the IDE or to associate event handlers with each form in the application.

 Chart The chart equivalent of TTable. Place the component on a form and right-click it to display the third-party developer's Help topics.

Win32 page components

See also Other Palette pages

The components on the Win32 page of the Component palette provide access to 32-bit Windows user interface common controls available to your applications.

 TabControl Analogous to a divider in a file cabinet or notebook, this component provides a set of mutually exclusive notebook style tabs.

 PageControl A page set used to make a multipage dialog box. Use this control to define multiple logical pages or sections of information within the same window. See About PageControl.

 ImageList 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). To create an image list, add the ImageList component to the form and double-click it to display the Image List Editor.

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

 TrackBar 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. 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. See About TrackBar.

 ProgressBar A rectangular bar that “fills” from left to right, like that shown when you copy files in the Windows Explorer. Use this control to provide visual feedback to the user about the progress of long operations or background processes. See About ProgressBar.

 UpDown Up and down arrow buttons to increment and decrement values. See About UpDown.

 HotKey Attaches a hot key to any component. See About HotKey.

 Animate A Windows animation control window that silently displays an Audio Video Interleaved (AVI) clip, a series of bitmap frames like a movie. See Adding silent video clips to an application.

 DateTimePicker Displays a list box for entering dates or times. Users can select a date from the calendar or select dates or times by scrolling with Up and Down arrows and by typing. You must have the latest version of COMCTL32.DLL, usually located in the Windows\System or Windows\System32 directory.

 MonthCalendar Displays a calendar that represents a single month. Set the Date property to display a particular month and highlight a date within that month. You can also display a range of dates by setting MultiSelect to True and supplying an EndDate. Optionally, the current date can be displayed below the calendar, even if it does not fall within the month represented by the calendar.

 TreeView Lets you 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. Use a tree view component to display the relationship between a set of containers or other hierarchical elements. See About TreeView.

 ListView Provides a way to display a list in columns. List views display data in a variety of

views. See Handling lists.

 HeaderControl Displays 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 it. See About HeaderControl.

 StatusBar Area to post the status of actions at the bottom of the screen. See About StatusBar.

 ToolBar Manages tool buttons and other controls, arranging them in rows and automatically adjusting their sizes and positions. See About ToolBars and Designing toolbars and cool bars.

 CoolBar Displays a collection of windowed controls (CoolBand objects) within movable, resizable bands. The user positions the controls by dragging the sizing grip to the left of each band. See Adding a cool bar component and Designing toolbars and cool bars.

 PageScroller Contains other objects in a client area that can scroll either horizontally or vertically. Users scroll the contents of the page scroller using large arrows on either end rather than using a scroll bar.

System page components

See also Other Palette pages

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

 Timer Timer is a nonvisual component that triggers a one-time or repeated event after a measured interval. You write the code that you want to occur at the specified time inside the timer component's OnTimer event. See About Timer.

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

 MediaPlayer Displays a VCR-style control panel for playing and recording multimedia video and sound files. See About MediaPlayer and Adding audio and/or video clips to an application.

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

 DdeClientConv Establishes a client connection to legacy Dynamic Data Exchange (DDE) server application. See About DdeClientConv.

 DdeClientItem Specifies the Dynamic Data Exchange (DDE) client data to transfer during a DDE conversation. See About DdeClientItem.

 DdeServerConv Establishes a server connection to legacy Dynamic Data Exchange (DDE) client application. See About DdeServerConv.

 DdeServerItem Specifies the Dynamic Data Exchange (DDE) server data to transfer during a DDE conversation. See About DdeServerItem.

Data Access page components

See also Other Palette pages

The components on the Data Access page of the Component palette let you connect to database information using the Borland Database Engine (BDE):

 DataSource Acts as a conduit between a dataset component such as TTable and data-aware components such as TDBGrid. See Using data sources

 Table Retrieves data from a physical database table via the BDE and supplies it to one or more data-aware components through a DataSource component. Conversely, it also sends data received from a component to a physical database via the BDE. See Working with tables

 Query 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. Conversely, uses SQL statements to send data from a component to a physical database via the BDE. See Working with queries

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

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

 SessionProvides global control over a group of Database components. A default TSession component is automatically created for each Delphi database application. You must use the TSession component only if you are creating a multithreaded database application. Each database thread requires its own session component. See Managing database sessions

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

 UpdateSQL Lets you use cached updates support with read-only datasets. For example, you could use a TUpdateSQL component with a "canned" query to provide a way to update 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. See About UpdateSQL

 NestedTable Retrieves the data in a nested dataset field and supplies it to data-aware controls through a datasource component.

Data Controls page components

See also Other Palette pages

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

 DBGrid 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, and so on. See Viewing and editing data with TDBGrid

 DBNavigator 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. See Navigating and manipulating records

 DBText Data-aware label that displays a field value in the current record. See Displaying data as labels

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

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

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

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

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

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

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

 DBLookupListBox DBLookupListBox is a data-aware list box that derives its list of display items from either a Lookup field defined for a dataset or a 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. See About DBLookupListBox.

To specify list box items using a lookup field, the dataset to which you link the control must already define a lookup field. See Defining a lookup field

 DBLookupComboBox DBLookupComboBox is a data-aware combo box that derives its drop-down list of display items from either a lookup field defined for a dataset or a 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. See About DBLookupComboBox.

To specify combo box list items using a lookup field, the dataset to which you link the control must already define a lookup field. See Defining a lookup field

 DBRichEdit A multiline edit control that can display and edit a rich text memo field in a dataset. See About DBRichEdit

 DBCtrlGrid 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. See About DBControlGrid.

 DBChart Place the component on a form and right-click it to display the third-party developer's Help topics.

ADO page components

See also Other Palette pages

The components on the ADO page of the Component palette let you connect to database information using ActiveX Data Objects (ADO):

ADOConnection	Sets up a persistent connection to an ADO database and provides support for transactions.
ADOCommand	Issues SQL commands directly against an ADO database without returning a result set.
ADODataSet	Represents the data from one or more tables in an ADO database and allows data-aware components to manipulate this data by connecting with a DataSource component. This is the most generic ADO dataset control, and can be used in place of ADOTable, ADOQuery, or ADOStoredProc.
ADOTable	Represents the data from a single database table via ADO and allows data-aware components to manipulate this data by connecting with a DataSource component.
ADOQuery	Uses SQL statements to retrieve data from a physical database table via ADO and allows data-aware components to manipulate this data by connecting with a DataSource component.
ADOStoredProc	Enables an application to access a server's stored procedures using ADO.
RDSConnection	Manages the marshaling of data when a Recordset object is passed from one process or machine to another. Use TRDSConnection when building multi-tier applications using business objects (Application Servers).

InterBase components

See also Other Palette pages

The components on the InterBase page of the Component palette let you connect directly to an InterBase database without using an engine such as the BDE or ActiveX Data Objects (ADO).

IBTable	Represents the data from a single InterBase table or view.
IBQuery	Uses SQL statements to retrieve data from an InterBase table or tables. TIBQuery is more easily scaled than other IB datasets when moving from local InterBase to a remote InterBase server.
IBStoredProc	Executes an InterBase Execute stored procedure. IBStoredProc does not represent a result set: use IBQuery or IBDataSet for stored procedures that return a result set.
IBDataBase	Represents the InterBase database connection. Use this component to manage transactions or provide connection parameters for a remote database.
IBTransaction	Provides discrete transaction control over a one or more database connections. The IBDataBase component uses IBTransaction to represent a transaction.
IBUpdateSQL	Lets you use cached update support with read-only queries. For example, you could use a IBUpdateSQL component with a "canned" query to provide a way to update the underlying datasets, giving you the ability to post updates to a read-only dataset.
IBDataSet	Represents the result set from an SQL SELECT command. IBDataSet lets you specify separate SQL commands for

		inserting, deleting, and updating records.
	IBEvents	Lets an application to register interest in, and asynchronously handle, events posted by an InterBase server.
	IBSQL	Executes an InterBase SQL statement with minimal overhead. IBSQL has no standard interface to data-aware controls and is unidirectional.
	IBDatabaseInfo	Returns information about an attached database, such as the version of the online disk structure (ODS), the number of cache buffers allocated, the number of database pages read from or written to, or write-ahead log information.
	IBSQLMonitor	Monitors dynamic SQL passed to the InterBase server. It introduces a single event, OnSQL, which receives the text for every dynamic SQL statement based to the server.

MIDAS page components

See also Other Palette pages

The components on the MIDAS page of the Component palette (not available in all versions) enable you to build multi-tiered database applications:

	ClientDataSet	Implements a database-independent dataset that can be used independently in a single-tiered application, or to represent data received from a server in a multi-tiered database application. See Creating and using a client dataset
	DCOMConnection	Establishes a DCOM connection to a remote server in a multi-tiered database application. See Connecting to the application server
	SocketConnection	Establishes a TCP/IP connection to a remote server in a multi-tiered database application. See Connecting to the application server
	DataSetProvider	Encodes data into packets than can be sent to client applications and applies updates that are received from client applications. See Creating a data provider for the application server.
	SimpleObjectBroker	Locates a server for a connection component from a list of available application servers. See Brokering connections.
	WebConnection	Establishes an HTTP connection to a remote server in a multi-tiered database application. See Connecting to the application server
	CorbaConnection	Establishes a CORBA connection to a remote server in a multi-tiered database application. See Connecting to the application server

InternetExpress page components

See also Other Palette pages

The components on the InternetExpress page of the Component palette let you build InternetExpress applications that are simultaneously a Web Server application and the client of a multi-tiered database application.

	XMLBroker	Fetches XML datapackets from an application server, which it makes available to the components that generate Web pages, and brokers updates received from a remote Web browser. See Using an XML broker.
	MidasPageProducer	Generates an HTML page that represents database information from an application server. The generated page contains datapackets encoded in XML and embedded javascript that

supplies the ability to navigate and update data. See Creating Web pages with a MIDAS page producer.

Internet page components

See also Other Palette pages

The components on the Internet page of the Component palette (not available in all versions) support the creation of Web server applications:


ClientSocket Add to a form or data module to turn an application into a TCP/IP client. ClientSocket specifies a desired connection to a TCP/IP server, manages the open connection, and terminates the completed connection. See Using client sockets and Working with sockets.


ServerSocket Add to a form or data module to turn an application into a TCP/IP server. ServerSocket listens for requests for TCP/IP connections from other machines and establishes connections when requests are received. See Using server sockets and Working with sockets.


WebBrowser Displays an HTML page in a Web browser-like viewer. You must have IE4 or better installed to use this component.

WebDispatcher Converts an ordinary data module to a Web module and enables the Web server application to respond to HTTP request messages. See About WebDispatcher and The Structure of a Web server application.


PageProducer Converts an HTML template into a string of HTML commands that can be interpreted by a client application such as a Web browser. The commands and HTML-transparent tags are replaced with customized content by the OnHTMLTag event. See Using page producer components.


QueryTableProducer . Assembles a sequence of HTML commands to generate a tabular display of the records from a TQuery object, which obtains its parameters from an HTTP request message. See Setting up a query table producer and Using dataset table producers.


DataSetTableProducer Assembles a sequence of HTML commands to generate a tabular display of the records from a TDataSet object. This allows an application to create images of a dataset for an HTTP response message. See Setting up a dataset table producer and Using dataset table producers.


DataSetPageProducer Converts an HTML template that contains field references into a string of HTML commands that can be interpreted by a client application such as a Web browser. Special HTML-transparent tags are replaced with field values. See Using dataset page producers.

FastNet page components

See also Other Palette pages

The components on the FastNet page of the Component palette offer a variety of internet access protocols for your applications:


NMDayTime Gets the date and time from an internet/intranet daytime server.


NMEcho Sends text to an internet echo server, and echoes it back to you.


NMFinger Gets information about a user from an internet finger server, using the Finger protocol described in RFC 1288.


NMFTP Implements file transfer protocol Invisible ActiveX control provides easy access for Internet File Transfer Protocol (FTP) services for transferring files and data between a remote and local machine.


NMHTTP Invisible ActiveX control implements the HTTP Protocol Client, allowing users to directly retrieve HTTP documents if no browsing or image processing is necessary.


NMMsg Sends simple ASCII text messages across the internet or intranet using TCP/IP protocol.


NMMsgServ Receives messages sent with the TNMMsg component.


NMNNTP Invisible ActiveX Client Control allows applications to access Networking News Transfer Protocol (NNTP) news servers. It provides news reading and posting capabilities.


NMPOP3 Invisible ActiveX control that retrieves mail from UNIX or other servers supporting POP3 protocol.


NMUUProcessor MIME encodes or UUEncodes files and decodes MIME-encoded or UUEncoded files.


NMSMTP ActiveX control that gives applications access to SMTP mail servers and mail posting capabilities.


NMStrm Sends streams to a stream server across the internet or an intranet.


NMStrmServ Receives streams sent with the TNMStrm component.


NMTime Gets the date and time from Internet time servers, as described in RFC 868.


NMUDP Invisible WinSock ActiveX Control provides easy access to User Datagram Protocol (UDP) network services. It implements WinSock for both client and server and represents a communication point utilizing UDP network services. It can also be used to send and retrieve UDP data.


PowerSock Serves as a base for creating controls for dealing with other protocols, or for creating custom protocols.


NMGeneralServer Serves as a base class for developing multi-threaded internet servers, such as custom servers or servers that support RFC standards.


HTML Invisible ActiveX control implements an HTML viewer, with or without automatic network retrieval of HTML documents, and provides parsing and layout of HTML data, as well as a scrollable view of the selected HTML page. The HTML component can also be used as a nonvisual HTML parser to analyze or process HTML documents.

 NMURL Decodes URL data into a readable string, and encodes standard strings into URL data format.

Decision Cube page components

See also Other Palette pages

The components on the Decision Cube page of the Component palette (not available in all versions) add multidimensional data analysis features to your applications. The Decision Cube components provide cross tabulation of data, letting you drill down, pivot, and summarize database information to help users visualize data for decision-making purposes:


DecisionCube A multidimensional data store. See Using decision cubes.


DecisionQuery Specialized form of TQuery used to define the data in a decision cube. See Creating decision datasets with the Decision Query editor.


DecisionSource Defines the current pivot state of a decision grid or a decision graph. See Using decision sources.


DecisionPivot Use to open or close decision cube dimensions or fields by pressing buttons. See Using decision pivots.


DecisionGrid Displays single and multidimensional data in table form. See Creating and Using decision grids.


DecisionGraph Displays fields from a decision grid as a dynamic graph that changes when dimensions are modified. See Using decision graphs.

QReport page components

See also

The Quick Report components on the QReport page of the Component palette enable you to visually design Quick Reports. You build reports with bands, adding titles, page headers and footers, multiple detail sets, summaries, group headers and footers. You can report from any DataSource, including TTable, TQuery, lists, arrays, and so on. Use the on-screen preview to check your results. Automatically perform calculations like summary and counting of fields. You can reset calculations at group level.

To see the third-party developer's help, place the component on a form and right-click it.

 QuickRep The basic report form on which you build all your reports. It is a visual component that takes the shape of the currently selected paper size. Create reports by dropping bands and printable components on the TQuickRep component and connecting it to a dataset.

 QRSubDetail Links additional datasets into a report. Typically you would set up a master/detail relationship between table or query components and create a similar relationship with TQRSubDetail components.

 QRStringsBand Drops bands containing strings onto a report.

 QRBand Drop bands on a TQuickRep component and set the BandType property to tell how the band will behave during report generation.

 QRChildBand If you have bands with expanding components and want other components to be moved down accordingly you can create a child band and put the moving components on it. It's also useful if you have very long bands that span multiple pages.

 QRGroup Allows you to group bands together and provides control for headers, footers, and page breaks.

 QRLabel Prints static or other non-database text. Enter the text to be displayed in the Caption property. You can split text on multiple lines and even multiple pages.

 QRDBText A data-aware version of the TQRLabel that prints the value of a database field. Calculated fields and text field types can be printed, including String fields, various numeric fields, date fields and memo fields. Text can span multiple lines and pages. You connect the component to the data field by setting the DataSource and DataField properties. Unlike regular data-aware components, TQRDBText works even with dataset controls disabled to improve speed.

 QRExprPrints database fields, calculations, and static text. Input a valid QuickReport expression in the Expression property.

 QRSysData Prints system information such as report title, current page number, and so on. Select the data to print in the Data property. Set any preceding text in the Text property.

 QRMemo Prints a large amount of text that does not come from a database field. It can be static text or you can change it during report generation. You can set the field to expand vertically as needed and then span multiple pages if necessary.

 QRExprMemo Allows you to programmatically generate contents using Quick Report expressions.

 QRRichText Allows you to embed rich text into your report.

 QRDBRichText Provides a Quick Report wrapper for accessing DBRichText fields in your reports.

 QRShape Draws simple shapes like rectangles, circles, and lines on a report.

 QRImage Displays a picture on a report. Supports all image formats supported by the TPicture class.

 QRDBImage Prints images stored in binary (BLOB) fields. Prints all graphics formats supported by Delphi.

 QRCompositeReport Allows you to combine more than one report together.

 QRPreview Brings up a form that allows you to preview a report on the screen and print it.

QRTextFilter Lets you export the contents of your report into text format.

 QRCSVFilter Lets you export the contents of your report into a comma-delimited database source file.

 QRHTMLFilter Lets you export the contents of your report into HTML.

 QRChart Allows you to take a TChart component and drop it onto your Quick Report form.

Dialogs page components

See also Other Palette pages

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

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

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

Each common dialog box component (except the PrinterSetup component) has an Options property that affects its appearance and behavior. To display the various Options in the Object Inspector, double-click the Options property.

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

 OpenDialog Displays a Windows common Open dialog box. Users can specify the name of a file to open in this dialog box

 SaveDialog Displays a Windows common Save dialog box. Users can specify the name of a file to save in this dialog box

 OpenPictureDialog Displays a modal Windows dialog box for selecting and opening graphics files. Identical to Open dialog box except that it has an image preview region.

 SavePictureDialog Displays a modal Windows dialog box for entering file names and saving graphics files. Identical to Save dialog box except that it has an image preview region.

 FontDialog Displays a Windows common Font dialog box that lets users can specify font, size, and style information.

 ColorDialog Displays a Windows common Color dialog box that lets users specify color information.

 PrintDialog Displays a Windows common Print dialog box that lets users specify printing information, such as a range of pages and the number of copies.

 PrinterSetupDialog Displays a Windows common Printer Setup dialog box that lets users change and set up printers.

 FindDialog Displays a Windows common Find dialog box that lets users specify a string to search for.

 ReplaceDialog Displays a Windows common Replace dialog box that lets users specify a search string and a replacement string.

Windows 3.1 page components

See also Other Palette pages

The components on the Windows 3.1 page of the Component palette provide Windows 3.1 control elements for backward compatibility with Windows 3.1 applications built with previous versions of Delphi. Many of these older controls implement the same behavior as more recent specialized 32-bit Windows controls.

When creating new applications, do not use these controls.
The following table indicates which control should be used instead:

Win 3.1 control	Replace with	Palette page of new control
DBLookupList	DBLookupListBox	Data Controls
DBLookupCombo	DBLookupComboBox	Data Controls
TabSet	TabControl	Win32
Outline	TreeView	Win32
TabbedNoteBook	PageControl	Win32
NoteBook	PageControl	Win32
Header	HeaderControl	Win32

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

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

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

 Outline Displays information in a variety of outline formats.

 TabbedNotebook 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

 Notebook Creates a component that can contain multiple pages. Used with the Notebook component, it enables users to change pages.

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

 FileListBox Displays a scrolling list of files in the current directory.

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

 DriveComboBox Displays a scrolling list of available drives.

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

Samples page components

See also Other Palette pages

The components on the Sample page of the Component palette are examples of customized components that you can build and add to the Component Palette. Source code to these sample components is included in the \SOURCE\SAMPLES directory of the default installation.


TSpinButton


TSpinEdit


TGauge


TDirectoryOutline


TColorGrid


TCalendar


IBEVentAlerter

ActiveX page components

See also Other Palette pages

The components on the ActiveX page of the Component palette are ActiveX objects. They are complete, portable working applications created by third-party developers.

To use these components, you must first open an ActiveX form with a current ActiveX library project. After placing a component on the ActiveX form, right-click it to display Properties and other commands or dialog boxes for configuring the component's functionality or setting its values. The Properties dialog box or other controls contain Help buttons that display the developer's Help system for the component.

 Chartfx Lets you create highly customized charts. Choose Properties to display a tabbed control panel that lets you define the values, appearance, and end-user functionality of the chart component.

 VSSpell Visual Speller, lets you customize a spelling checker.

 F1Book Formula One, lets you design a spreadsheet with its full-featured Designer.

 VtChart Lets you create true 3D charts.

Servers page components

See also Other Palette pages

The components on the Servers page of the Component palette are VCL wrappers for common COM servers. They are all descendants of TOleServer and were created by importing a type library and installing the resulting component.

These components automatically launch the server when you call one of its methods. You can also connect to the COM server by calling the Connect method. For example:

```
WordApplication1.Connect;
```

After connecting, you will most likely also want to set the Visible property:

```
WordApplication1.Visible := True;
```

You can use any of the properties, events, or methods exposed by the COM server by using the component's properties, events, and methods.

Common component tasks

See also

When designing your application interface, there are procedures you might 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
- Handling user events
- Setting the component focus in a form
- Managing layout
- Setting the tab order
- Enabling and disabling components
- Using action lists
- Implementing drag-and-drop
- Implementing drag-and-dock
- Working with text
- Adding graphics to controls

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. You can save any form you design as a template that you can reuse in other projects.

- Making your form a component
- Controlling when forms reside in memory
- Reusing forms as DLLs

Tasks

- To make the form stay on top of other open windows (for instance, the Project Manager or Alignment Palette) at runtime, 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 runtime.)
- 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 runtime, 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.

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 OnClickevent 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 user events, choose from the following topics.

- Defining the handler type
- Keyboard events
- Mouse events
- Drag and drop 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 runtime.

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

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.

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.

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) 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.)

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 OnKeyPressevent because it has no alphanumeric value. See Responding To The OnKeyPress Event

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;
```

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

See also 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.

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.

Creating a Select File dialog box

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 event

   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;
```

About the Timer component

TTimer reference Common component tasks

Purpose

Use the Timer component to trigger an event, either one time or repeatedly, after a measured interval. 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;
```

About the PaintBox component

TPaintBox reference Common component tasks

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.

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 certain capabilities.

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.

Here are the methods (listed alphabetically) for controlling a device:

*Back	*Pause	Rewind
Close	PauseOnly	*StartRecording
*Eject	*Play	*Step

*Next *Previous *Stop

Open Resume

About the OLEContainer component

TOLEContainer reference Creating an AutomationController

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.

About the DDEClientConv component

TDDEClientConv reference About DDE

Purpose

DDE is an older technology for interapplication communication. For new projects that need not access legacy DDE server applications, you should instead use Automation or other COM technology.

Use the DDEClientConv component to provide your application with the ability to establish a DDE conversation with a legacy DDE server 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.

Important properties

- To specify the topic of a DDE conversation, use the DDETopic property.
- To specify the DDE server application, use the DDEService property. Depending on the server, the server application name is usually but not always the executable.
- To specify the executable name for the DDE server application, use the ServiceApplication property.
- To define whether the connection to the DDE server application is established automatically when the form runs, use the ConnectMode property.
- To filter characters out of the text data transfer from a server application, use the FormatChars property.

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

About the DDEClientItem component

TDDEClientItem reference About DDE

Purpose

DDE is an older technology for interapplication communication. For new projects that need not access legacy DDE server applications, you should instead use Automation or other COM technology.

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 to access legacy DDE server applications. If you will need only one item, the DDEClientConv is not necessary.

Important properties

- To specify the DDE client conversation component, use the DDEConv property.
- To specify the item of the DDE conversation, use the DDEItem.
- To find out what the DDE Server sent, read the Lines property.

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

About the DDEServerConv component

TDDEServerConv reference About DDE

Purpose

DDE is an older technology for interapplication communication. For new projects that need not support legacy DDE client applications, you should instead use Automation or other COM technology.

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

This component works in conjunction 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

About the DDEServerItem component

TDDEServerItem reference About DDE

Purpose

DDE is an older technology for interapplication communication. For new projects that need not support DDE client applications, you should instead use Automation or other COM technology.

Use the DDEServerItem component to 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.

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 effect.

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 into its new location.
4. Click OK for your changes to take effect.

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 effect.

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 effect.

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 effect.

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 effect.

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 effect.

Select Frame dialog box

See also

The Select Frame dialog lists all the frames included in the current project. Choose the frame you want to embed in the form or frame you just clicked on, then press OK.

CPU window

See also

The CPU window consists of five separate panes. Each pane gives you a view into a specific low-level aspect of your running application.

- Disassembly pane displays the assembly instructions that have been disassembled from your application's machine code. In addition, the Disassembly pane displays the original program source code above the associated assembly instructions.
- Memory Dump pane displays a memory dump of any memory accessible to the currently loaded executable module. By default, memory is displayed as hexadecimal bytes.
- Machine Stack pane displays the current contents of the program stack. By default, the stack is displayed as hexadecimal longs (32-bit values).
- CPU Registers pane displays the current values of the CPU registers.
- Flags pane displays the current values of the CPU flags.

Right-click anywhere on the CPU window to access commands specific to the contents of the current pane.

Opening the CPU window

To open the CPU window anytime during a debugging session,

Choose View|Debug Windows|CPU or right-click the Code editor and choose Debug|CPU View to open the Disassembly pane at the location of the execution point.

- The CPU window opens automatically whenever program execution stops at a location for which source code is unavailable. For example, the debugger cannot open the source file if you link a DLL built with debug information but do not include its source file in your project, or if you place the source file in a directory not specified in your project.

Resizing the CPU window panes

You can customize the layout of the CPU window by resizing the panes within the window. Drag the pane boarders within the window to enlarge or shrink the windows to your liking.

Disassembly pane

The Disassembly pane is part of the CPU Window.

The left side of the Disassembly pane lists the address of each disassembled instruction. A green arrow to the left of the memory address indicates the location of the current execution point. To the right of the memory addresses, the Disassembly pane displays the assembly instructions that have been disassembled from the machine code produced by the compiler. If you make the Disassembly pane wide enough, the debugger displays the instruction opcodes following the listing of the instruction memory addresses.

When you click an address in the Disassembly pane,

- the upper left corner shows the effective address (when available) and the value it stores. For example, if you select an address containing an expression in brackets such as `[eax+edi*4-0x0F]`, the top of the Disassembly pane shows the location in memory being referenced and its current value.
- the upper right corner shows the current thread ID.

If you are viewing code that has debug information available, the debugger displays the source code that is associated with the disassembled instructions.

- Press Ctrl+Left Arrow and Ctrl+Right Arrow to shift the starting point of the display up or down one byte. Beware that changing the starting point of the display in the Disassembly pane changes where the debugger begins disassembling the machine code.

Disassembly pane commands

Right-click the Disassembly pane to access the following commands:

- Enabled.This menu option is only available by right-clicking in the gutter.
- Breakpoint Properties.This menu option is only available by right-clicking in the gutter.
- Run to Current
- Toggle Breakpoint
- Go to Address
- Go to Current EIP
- Follow
- Caller
- Previous
- Search
- View Source
- Mixed
- New EIP
- Change Thread
- View FPU

Run to Current

The Run To Current command lets you run your program at full speed to the instruction that you have selected in the Disassembly pane. After your program is paused, you can use this command to resume debugging at a specific program instruction.

Toggle Breakpoint

This command adds or removes a breakpoint at the selected instruction in the Disassembly pane. When you choose Toggle Breakpoint, the debugger sets an unconditional (simple), breakpoint at the instruction that you have selected in the Disassembly pane. A simple breakpoint has no conditions, and the only action is that it will pause the program's execution.

If a breakpoint exists on the selected instruction, then Toggle Breakpoint will delete the breakpoint at that code location.

Go to Address

The Go to Address command prompts you for a new area of memory to display in the Disassembly pane of the CPU window. Enter any expression that evaluates to a program memory location. Be sure to precede hexadecimal values with $.

- The debugger displays dashes if you view a program memory location in which nothing is loaded.

You can also press Ctrl+Left Arrow and Ctrl+Right Arrow to shift the starting point of the display up or down one byte.

Go to Current EIP

This command positions the Disassembly pane at the location of the current program counter (the location indicated by the EIP register). This location indicates the next instruction to be executed by your program.

This command is useful when you have navigated through the Disassembly pane, and you want to return to the next instruction to be executed.

Follow

This command positions the Disassembly pane at the destination address of the instruction currently highlighted.

Use the Follow command in conjunction with instructions that cause a transfer of control (such as **CALL**, **JMP**, and **INT**) and with conditional jump instructions (such as **JZ**, **JNE**, **LOOP**, and so forth). For conditional jumps, the address is shown as if the jump condition is TRUE. Use the Previous command to return to the origin of the jump.

Caller

This command positions the Disassembly pane at the instruction past the one that called the current interrupt or subroutine.

- If the current interrupt routine has pushed data items onto the stack, the debugger might not be able to determine where the routine was called from.

Note: Caller will not work unless you turn on stack frames (Project|Options Compiler page).

Previous

This command restores the Disassembly pane to the display it had before you issued the last Follow command.

Search

This command searches forward in the Disassembly pane for an expression or byte list that you supply. In the Enter Search Bytes dialog, supply a byte list to search for two or more values located in a specific order. Be sure to precede hexadecimal values with $

For example, if you enter

```
$5D $C3
```

the debugger goes to the following location:

```
004013AB 5D
004013AC C3
```

You can also search for DWords, but you must reverse the order of the bytes.

For example, if you enter

```
$1234
```

the debugger positions the pane at the following location in memory:

```
34 12
```

Enter Search Bytes dialog box

This dialog is displayed when you choose the Search command from the CPU window. You can search for decimal numbers, hexadecimal numbers (precede values with $), strings (surround in single quotes or specify its ASCII equivalent), or DWords (reverse byte order).

You can enter a byte list to search for two or more values located in a specific order. Be sure to precede hexadecimal values with $.

For example, if you enter

```
$5D $C3
```

the debugger goes to the following location:

```
004013AB 5D
004013AC C3
```

You can also search for DWords, but you must reverse the order of the bytes.

For example, if you enter

```
$1234
```

the debugger positions the pane at the following location in memory:

```
34 12
```

View source

This command activates the Code editor and positions the insertion point at the source code line that most closely corresponds to the disassembled instruction selected in the Disassembly pane. If there is no corresponding source code (for example, if you are examining Windows kernel code), this command has no effect.

Mixed

Switches the display format of the Disassembly pane:

When Mixed is….	The Disassembly pane displays….
checked	source code lines before the first disassembled instruction relating to that source line.
unchecked	disassembled instructions without source code.

Change thread

This command opens the Select a Thread dialog box. Select the thread you want to debug from the threads listed. When you choose a new thread from the Disassembly pane, all panes in the CPU window reflect the state of the CPU for that thread.

View FPU

Right-click in any of the CPU view panes and choose View FPU to display the FPU window. You use the FPU window to view the contents of the FPU component of the CPU. You can display either floating-point information or MMX information.

The FPU window displays values and status for each register in the FPU as well as the FPU status, control, and tag words. The flags encoded in the control and status word are displayed in separate panes. You can also view the address, opcode, and operand that corresponds to the last FPU instruction executed.

Select a Thread dialog box

The Select a Thread dialog box is displayed when you right-click in any of the panes in the CPU window. It lists processes and threads so you can change the current process or thread directly from the CPU window.

Select the thread you want to debug from the threads listed. When you choose a new thread from the Disassembly pane, all panes in the CPU Window reflect the state of the CPU for that thread.

New EIP

This command changes the location of the instruction pointer (the value of EIP register) to the line currently highlighted in the Disassembly pane. Use this command when you want to skip certain machine instructions. When you resume program execution, execution starts at this address.

- This command is not the same as stepping through instructions; the debugger does not execute any instructions that you might skip.
- Use this command with extreme care; it is easy to place your system in an unstable state when you skip over program instructions.

Memory Dump pane

The Memory Dump pane is part of the CPU Window.

The Memory Dump pane displays the raw values contained in addressable areas of your program. The pane has three sections: the memory addresses, the current values in memory, and an ASCII representation of the values in memory.

The Memory Dump pane displays the memory values in hexadecimal notation. The leftmost part of each line shows the starting address of the line. Following the address listing is an 8-byte hexadecimal listing of the values contained at that location in memory. Each byte in memory is represented by two hexadecimal digits. Following the hexadecimal display is an ASCII display of the memory. Non-printable values are represented with a period.

The format of the memory display depends on the format selected with the Display As command. If you choose a floating-point display format (Single, Double, or Extended), a single floating-point number is displayed on each line. The Bytes format (default) displays 8 bytes per line, Words displays 4 words per line, DWords displays 2 long words per line, and QWords displays a single quadword per line.

- You can press Ctrl+Left Arrow and Ctrl+Right Arrow to shift the starting point of the display up or down one byte. Using these keystrokes is often faster than using the Go to Address command to make small adjustments to the display.

Memory Dump pane commands

Right-click the Memory Dump pane to access the following commands:

- Go to Address
- Search
- Next
- Change
- Follow
- Previous
- Display As
- Change Thread
- View FPU

Go to Address

The Go to Address command prompts you for a new area of memory to display in the Memory Dump pane of the CPU window. Enter any expression that evaluates to a program memory location. Be sure to precede hexadecimal values with $.

- The debugger displays dashes if you view a program memory location in which nothing is loaded.

You can also press Ctrl+Left Arrow and Ctrl+Right Arrow to shift the starting point of the display up or down one byte.

Change thread

This command opens the Select a Thread dialog box. Select the thread you want to debug from the threads listed. When you choose a new thread from the Memory Dump pane, all panes in the CPU window reflect the state of the CPU for that thread.

Search

This command searches forward in the Memory Dump pane for an expression or byte list that you supply. Supply a byte list to search for two or more values located in a specific order. Be sure to precede hexadecimal values with $.

For example, if you enter

```
$5D $C3
```

the debugger positions the pane at the following location:

```
004013AB 5D
004013AC C3
```

You can also search for DWords, but you must reverse the order of the bytes.

For example, if you enter

```
$1234
```

the debugger positions the pane at the following location in memory:

```
34 12
```

Next

Finds the next occurrence of the item you last Searched for in the Memory Dump pane.

Change

Lets you modify the bytes located at the current cursor location and prompts you for an item of the current display type.

- You can invoke this command by typing directly in the Dump pane.

Follow

Lets you choose the following commands:

Near Code	Positions the Disassembly pane at the address currently selected in the Memory Dump pane.
Offset to Data	Lets you follow DWord-pointer chains (near and offset only) and positions the Memory Dump pane at the address specified by the DWord currently highlighted.

Previous

This command restores the Memory Dump pane of the CPU window to the location displayed before you issued the last Follow command.

Display As

Use the Display As command to format the data listed in the Memory Dump pane of the CPU window. You can choose any of the data formats listed in the following table:

Data type	Display format
Bytes	Hexadecimal bytes
Words	2-byte hexadecimal numbers
DWords	4-byte hexadecimal numbers
QWords	8-byte hexadecimal numbers
Singles	4-byte floating-point numbers using scientific notation
Doubles	8-byte floating-point numbers using scientific notation
Extendeds	10-byte floating-point numbers using scientific notation

Machine Stack pane

The Machine Stack pane is part of the CPU Window.

The Machine Stack pane displays the raw values contained in the your program stack. The pane has three sections: the memory addresses, the current values on the stack, and an ASCII representation of the stack values.

- A green arrow indicates the value at the top of the call stack.

The Machine Stack pane displays the memory values in hexadecimal notation. The leftmost part of each line shows the starting address of the line. Following the address listing is a 4-byte listing of the values contained at that memory location. Each byte is represented by two hexadecimal digits. Following the hexadecimal display is an ASCII display of the memory. Non-printable values are represented with a period.

The format of the memory display depends on the format selected with the Display As command. If you choose a floating-point display format (Single), a single floating-point number is displayed on each line. The Bytes format displays 4 bytes per line, Words displays 2 words per line, and DWords (the default) displays 1 long word per line.

- You can press Ctrl+Left Arrow and Ctrl+Right Arrow to shift the starting point of the display up or down one byte. Using these keystrokes is often faster than using the Go to Address command to make small adjustments to the display.

Machine Stack pane commands

Right-click the Machine Stack pane to access the following commands:

- Go to Address
- Top of Stack
- Follow
- Previous
- Change
- Display As
- Change Thread
- View FPU

Go to Address

The Go to Address command prompts you for a new area of memory to display in the Machine Stack pane of the CPU window. Enter any expression that evaluates to a program memory location. Be sure to precede hexadecimal values with $.

- The debugger displays dashes if you view a program memory location in which nothing is loaded.

You can also press Ctrl+Left Arrow and Ctrl+Right Arrow to shift the starting point of the display up or down one byte.

Change thread

This command opens the Select a Thread dialog box. Select the thread you want to debug from the threads listed. When you choose a new thread from the Machine Stack pane, all panes in the CPU window reflect the state of the CPU for that thread.

Top of stack

Positions the Machine Stack pane at the address of the stack pointer (the address held in the ESP register).

Follow

Lets you choose the following commands:

Offset to stack	Lets you follow DWord-pointer chains (near and offset only) on the call stack and positions the Machine Stack pane at the address location of the value currently selected in the Machine Stack pane.
Near Code	Positions the Disassembly pane at the address location of the value currently selected in the Machine Stack pane.
Offset to Data	Lets you follow DWord-pointer chains (near and offset only) and position the Memory Dump pane at the address location of the value currently selected in the Machine Stack pane.

Previous

This command restores the Machine Stack pane in CPU window to the location displayed before you issued the last Follow command.

Change

Lets you enter a new value for the stack word currently highlighted.

- You can invoke this command by typing directly in the Machine Stack pane.

Display As

Use the Display As command to format the data that's listed in the Machine Stack pane of the CPU window. You can choose any of the data formats listed in the following table:

Data type	Display format
Bytes	Displays data in hexadecimal bytes
Words	Displays data in 2-byte hexadecimal numbers
DWords	Displays data in 4-byte hexadecimal numbers
Singles	Displays data in 4-byte floating-point numbers using scientific notation

CPU Registers pane

The CPU Registers pane is part of the CPU Window.

The CPU Registers pane displays the contents of the CPU registers of the 80386 and greater processors. These registers consist of eight 32-bit general purpose registers, six 16-bit segment registers, the 32-bit program counter (EIP), and the 32-bit flags register (EFL).

After you execute an instruction, the CPU Registers pane highlights in red any registers that have changed value since the program was last paused.

Registers pane commands

Right-click the CPU Registers pane to access the following commands:

- Increment Register
- Decrement Register
- Zero Register
- Change Register
- Change Thread
- View FPU

Increment register

Increment Register adds 1 to the value in the currently highlighted register. This option lets you test “off-by-one” bugs by making small adjustments to the register values.

Decrement register

Decrement Register subtracts 1 from the value in the currently highlighted register. This option lets you test “off-by-one” bugs by making small adjustments to the register values.

Zero register

The Zero Register command sets the value of the currently highlighted register to 0.

Change register

Lets you change the value of the currently highlighted register. This command opens the Change Register dialog box where you enter a new value. You can make full use of the expression evaluator to enter new values. Be sure to precede hexadecimal values with $.

Change thread

This command opens the Select a Thread dialog box. Select the thread you want to debug from the threads listed. When you choose a new thread from the CPU Registers pane, all panes in the CPU window reflect the state of the CPU for that thread.

Flags pane

The Flags pane is part of the CPU Window.

The Flags pane shows the current state of the flags and information bits contained in the 32-bit register EFL. After you execute an instruction, the Flags pane highlights in red any flags that have changed value since the program was last paused.

The processor uses the following 15 bits in this register to control certain operations and indicate the state of the processor after it executes certain instructions:

Value	Flag/bit name	EFL register bit number
CF	Carry flag	0
PF	Parity flag	2
AF	Auxiliary carry flag	4
ZF	Zero flag	6
SF	Sign flag	7
TF	Trap flag	8
IF	Interrupt flag	9
DF	Direction flag	10
OF	Overflow flag	11
IO	I/O privilege level	12 and 13
NF	Nested task flag	14
RF	Resume flag	16
VM	Virtual 8086 mode	17
AC	Alignment check	18

Flags pane commands

Right-click the Flags pane to access the following commands:

- Toggle Flag
- Change Thread
- View FPU

Toggle flag

The flag and information bits in the Flags pane can each hold a binary value of 0 or 1. This command toggles the selected flag or bit between these two binary values.

Change thread

This command opens the Select a Thread dialog box. Select the thread you want to debug from the threads listed. When you choose a new thread from the Flags pane, all panes in the CPU window reflect the state of the CPU for that thread.

Event Log

The event log shows process control messages, breakpoint messages, OutputDebugString messages, and window messages. Using the right-click menu, you can clear the event log, save the event log to a text file, add a comment to the event log and set options for the event log. To display the event log, select View|Debug Windows|Event Log.

The context menu (right-click on the event log) displays the following menu options:

- Clear Events
- Save Events to File
- Add Comments
- Properties
- Dockable - Toggles whether the Event Log Window is enabled for docking

To set the properties for the Event Log, right-click in the Event Log and select Properties or select Tools| Debugger Options. The properties include which messages to display in the event log and how many events to show in the event log.

COM options

When the Enable COM cross-process support option on the Distributed Debugging page of the Debugger Options dialog box is checked, COM events are added to the event log. There are three types of COM events: ClientStart, ServerStart, and ClientEnd. Each event shows the GUID, the method number, and the HResult of the COM RPC.

CORBA options

When the Enable CORBA cross-process support option on the Distributed Debugging page of the Debugger Options dialog box is checked, you can step into remote CORBA processes while debugging. When checked, the controls in the ORB events list box are enabled and you can set options for each ORB event that the debugger sends notifications for. For each event, you can define any number of action sets. Events for which you set options are added to the event log.

Clear Events **(Event Log context menu)**

See also

Select Clear Events to remove all messages from the event log window. Clear events is disabled when the Event Log is empty.

To save the current messages to a text file before clearing the event log, use Save Events To File.

To limit the size of the event log so that earlier messages are automatically cleared, use Properties.

Save Events to File **(Event Log context menu)**

See also

Select Save Events to File to save the messages in the event log window to a text file. Choosing this command displays the Save Events to File dialog, where you can specify the name of a text file that will contain the current contents of the Event Log.

Save Events to File is disabled when the Event Log is empty.

Add Comment **(Event Log context menu)**

See also

Select Add Comment to add a message to the event log. This command displays the Add Comment to Event Log dialog, where you can type in any message. The message then appears at the end of the event log.

Properties (Event Log context menu)

See also

Select Properties to display the Event Log page of the Tools|Debugger Options dialog. Use the Event Log page to:

- Limit the number of messages that can appear in the event log (Length), allow the event log to grow until you manually clear it (Unlimited), or clear the event log (Clear log on run). When debugging multiple processes, only the first process loaded will cause the event log to be cleared.
- Instruct the debugger to send messages to the event log when.
- It encounters breakpoints (Breakpoint messages).
- Changes occur to the process state (Process messages).
- OutputDebugStrings is called (Output messages).
- The process receives Windows messages (Window messages).

FPU window

See also

The FPU window is an IDE debugger window that lets you view the contents of the Floating-Point Unit in the CPU. You can use the FPU window to display floating-point or MMX information. (MMX is Intel's enhanced version of the Pentium processor. It has additional instructions for handling multimedia operations, and uses a streamlined internal architecture that yields increased program speed and efficiency.)

The FPU window displays register values, status, control and tag words in the FPU. The FPU window displays information in three panes:

- FPU Registers pane, the largest pane, displays the floating-point register stack.
- Control Flags pane lists the control flags encoded in the control word.
- Status Flags pane lists the status flags encoded in the status word.

Above the FPU Registers pane is a panel control that displays the Instruction Pointer (IPTR) address, opcode, operand (OPTR) address of the last floating-point instruction executed.

The FPU window is not available in the Standard edition of Delphi.

FPU Registers pane

See also

The FPU Registers pane is part of the FPU Window.

The FPU Registers pane displays the floating-point register stack (ST0 through ST7) in ascending order. After the list, the control word, status word, and tag word are shown.

The information displayed for each of the eight registers is shown as follows:

Register name Register status Register value

The register status can be one of the following values:

Register status	Description
Empty	Indicates that the register contains invalid data. When a register is empty, no value is displayed for that register, because the data in the register is presumed to be invalid.
Valid	Indicates that the register contains nonzero valid data.
Zero	Indicates that the register contains a valid value of zero.
Spec. (Special)	Indicates that the register contains valid data, but the valid data represents a special condition, either NAN (not a number), infinity, or a denormalized value.

The status of each register is determined by examining the tag word and the eleventh through thirteenth bits of the status word (top of stack indicator). When a register's status is not Empty, the value of the register in long double (extended) format is displayed immediately following the status. The registers can be displayed in different formats (other than long doubles).

The control, status, and tag words are displayed in hexadecimal format only. For these three words, any values that were altered by the last run operation are displayed in red.

Right-click on this pane to display the FPU Registers pane context menu.

FPU Registers pane context menu

See also

Right-click on the FPU Registers pane to display the context menu which includes the following menu options:

- Zero
- Empty
- Change
- Display As
- Radix (available only when MMX registers are shown)
- Show

Zero **(FPU Registers pane context menu)**

Zero sets the selected register's value to 0. When used on one of the seven FPU registers, this command also sets that register's tag bits in the tag word to 01 indicating that the register holds a zero value.

Empty **(FPU Registers pane context menu)**

Empty sets the selected register's tag bits in the tag word to 11 indicating that the register is empty. This command is grayed out if the selected register is the CTRL word, STAT word, or TAG word.

Change **(FPU Registers pane context menu)**

Change brings up a dialog in which the user can enter a new value for the selected register. When used on one of the seven FPU registers, this command also sets that register's tag bits in the tag word to 00 indicating that the register holds a valid value.

The value you enter in the Change dialog should be an Extended (long double) value when the contents are displayed as Extendeds (long doubles). Otherwise, the value should be an integer.

Display As **(FPU Registers pane context menu)**

Display As brings up a menu that contains the possible Display types for the view. The current display type is indicated on the menu with a bullet point. The items on the submenu change depending on which item under the Show menu is selected.

For FPU registers, the possible display types are Words and Extendeds (long doubles). For MMX registers, the possible display types are Bytes, Words, DWords (doubles), and QWords (quad words).

Radix **(FPU Registers pane context menu)**

Radix is only visible when the MMX register are shown (right-click and choose Show|MMX Registers). The current radix is indicated on the menu with a bullet point. The possible Radix values are Binary, Decimal, and Hexadecimal. Selecting one determines how the values in the MMX register are displayed.

Show **(FPU Registers pane context menu)**

Show brings up a menu that contains the possible show modes for the view. The current show mode is indicated on the menu with a bullet point. The possible show modes are FPU Registers and MMX Registers. Selecting one toggles which registers are shown in the Registers pane.

When FPU Registers is selected, the registers shown are the 10-byte FPU registers ST(0) through ST(7). The registers can be viewed as either Extended (long double) values or as 5 DWord values.

MMX registers can only be shown on a computer that is MMX enabled. When MMX Registers is selected, the registers shown are the 8-byte MMX registers MM0 through MM7. The registers can be viewed as 8 Byte values, 4 Word values, 2 DWord values, or 1 QWord value. These values can be shown in either binary, decimal, or hexadecimal format (see Radix).

Control Flags pane

See also

The Control Flags pane is part of the FPU Window. It lists the control flags encoded in the control word. Any flags that were altered by the last run operation are displayed in red. The control flags are as follows:

Flag	Description	Bit # in control word
IM	Invalid Operation Exception	0
DM	Denormalized Operation Exception Mask	1
ZM	Zero Divide Exception Mask	2
OM	Overflow Exception Mask	3
UM	Underflow Exception Mask	4
PM	Precision Exception Mask	5
PC	Precision Control	8, 9
RC	Rounding Control	10, 11
IC	Infinity Control (Obsolete)	12

Select any of the flags and right-click to change the flag's value. For single-bit flags, it changes the value from 0 to 1 or from 1 to 0. For multi-bit flags, it cycles through all possible values.

Control Flags pane context menu

See also

Right-click on the Control Flags pane to change the value of any of the control flags. For single-bit flags, it changes the value from 0 to 1 or from 1 to 0. For multi-bit flags, it cycles through all possible values.

Status Flags pane

See also

The Status Flags pane is part of the FPU Window. It lists the status flags encoded in the status word. Any flags that were altered by the last run operation are displayed in red. The flags shown are listed below:

Flag	Description	Bit # in control word
IE	Invalid Operation Exception	0
DE	Denormalized Operation Exception	1
ZE	Zero Divide Exception	2
OE	Overflow Exception	3
UE	Underflow Exception	4
PE	Precision Exception	5
SF	Stack Fault	6
ES	Error Summary Status	7
C0	Condition Code 0 (CF)	8
C1	Condition Code 1	9
C2	Condition Code 2 (PF)	10
ST	Top of Stack	11-13
C3	Condition Code 3 (ZF)	14
BF	FPU Busy	15

Select any of the flags and right-click to change the flag's value. For single-bit flags, it changes the value from 0 to 1 or from 1 to 0. For multi-bit flags, it cycles through all possible values.

Status Flags pane context menu

See also

Right-click on the Status Flags pane to change the value of any of the control flags. For single-bit flags, it changes the value from 0 to 1 or from 1 to 0. For multi-bit flags, it cycles through all possible values.

Toggle Flag **(FPU flags pane context menu)**

See also

Select any flag in either the Status Flags pane or the Control Flags pane. Right-click and choose Toggle Flag to change the value of the selected flag. For single-bit flags, the value changes from 0 to 1 or from 1 to 0. For multi-bit flags, all possible values are cycled through.

Enter New Value dialog box

See also

The Enter New Value dialog box is displays when you right-click and choose Change from the Dump, Stack, or Register pane of the CPU window or the Register pane of the FPU window. Enter a value for the currently selected item. Precede hexadecimal values with $.

From the Dump and Stack panes of the CPU windw, you can enter more than one value separated by a space. Note that you must enter a value that corresponds to the current display type set using Display As.

From the Register pane of the FPU view, you must specify a single 32-bit hexadecimal value (use of decimal numbers is allowed but is not typical).

Add Source Breakpoint dialog box (Run|Add Breakpoint|Source Breakpoint or Add|Source Breakpoint)

See also

The Source Breakpoint command displays the Add Source Breakpoint dialog box where you can set a breakpoint on a specific line location in your source code. When you run your program, the execution point in the Code editor indicates the breakpoint location. The breakpoint appears in the Code editor and the Breakpoint List.

You can also associate actions with the breakpoints you add. See Associating actions with breakpoints.

Filename

Specifies the source file for the source breakpoint. Enter the name of the source 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 encountered. 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. Functions are valid if they return a Boolean type.

Pass count

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

The debugger increments the pass count each time the line containing the breakpoint is encountered. When the pass count equals the specified number, the debugger pauses program execution. For example, if the pass count is set to 3, you will see 0 of 3, 1 of 3, 2 of 3, then 3 of 3 in the pass count. Program execution stops at 3 of 3.

Because the debugger increments the count 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 number of passes that occurred.

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

Group

When setting a breakpoint using one of the Breakpoint Properties dialogs, you make it a member of a group by entering a group name in the Group field. See Organizing breakpoints into groups. Once your breakpoints are organized into groups, you can disable and enable groups of breakpoints by using the Disable Group and the Enable Group commands on the Breakpoint List context menu (right-click on the Breakpoint List).

Advanced button

Click the Advanced button if you want to associate actions with breakpoints: Enter the appropriate text in each field for each action you want to associate with the breakpoint.

Action	Description
Break	When checked, halts execution; the traditional and default action of a breakpoint.
Ignore subsequent exceptions	When checked, ignore all subsequent exceptions raised by the current process during the current debug session (the debugger will not stop on any exception). Use this with "Handle subsequent exceptions" as a

	pair. You can surround specific blocks of code with the Ignore/Handle pair to skip any exceptions which occur in that block of code.
Handle subsequent exceptions	When checked, handle all subsequent exceptions raised by the current process during the current debug session (the debugger will stop on exceptions based on the current exception settings in the Tools|Debugger options (Language and OS exception pages). This action does not mean stop on all exceptions no matter what. Use it to turn on normal exception behavior after another breakpoint disabled normal behavior using the Ignore subsequent exceptions option.
Log message	Writes the specified message in the event log. You specify the message to log.
Eval expression	Evaluates the specified expression and because Log result is checked by default writes the result of the evaluation to the event log. Uncheck Log result to evaluate without logging.
Log result	Becomes enabled when text is entered into Eval expression and is checked by default. If checked, writes the result of the evaluation in the Eval expression to the event log. If unchecked the evaluation is not logged.
Enable group	Enables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.
Disable group	Disables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.

Source Breakpoint Properties dialog box **(Breakpoint List context menu)**

See also

Use the Source Breakpoint Properties dialog box to change a source breakpoint or set a new one.

Keep existing Breakpoint

Check “Keep existing Breakpoint” to keep the old breakpoint and create a new one. If you do **not** check “Keep existing Breakpoint”, the breakpoint will be changed and the old breakpoint will not be saved.

Filename

Specifies the source file for the source breakpoint. Enter the name of the source 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 encountered. 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.

Pass count

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

The debugger increments the pass count each time the line containing the breakpoint is encountered. When the pass count equals the specified number, the debugger pauses program execution. For example, if the pass count is set to 3, you will see 0 of 3, 1 of 3, 2 of 3, then 3 of 3 in the pass count. Program execution stops at 3 of 3.

Because the debugger increments the count 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 number of passes that occurred.

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

Group

When setting a breakpoint using one of the Breakpoint Properties dialogs, you make it a member of a group by entering a group name in the Group field. See Organizing breakpoints into groups. Once your breakpoints are organized into groups, you can disable and enable groups of breakpoints by using the Disable Group and the Enable Group commands on the Breakpoint List context menu (right-click on the Breakpoint List).

Advanced button

Click the Advanced button if you want to associate actions with breakpoints: Enter the appropriate text in each field for each action you want to associate with the breakpoint.

Action	Description
Break	When checked, halts execution; the traditional and default action of a breakpoint.
Ignore subsequent exceptions	When checked, ignore all subsequent exceptions raised by the current process during the current debug session (the debugger will not stop on any exception). Use this with "Handle subsequent exceptions" as a pair. You can surround specific blocks of code with the Ignore/Handle pair to skip any exceptions which occur in that block of code.
Handle subsequent exceptions	When checked, handle all subsequent exceptions raised by the current process during the current debug session (the debugger will stop on exceptions based on the current exception settings in the Tools|Debugger options (Language and OS exception pages). This action does not mean stop on all exceptions no matter what. Use it to turn on normal exception behavior after another breakpoint disabled normal behavior using the Ignore subsequent exceptions option.
Log message	Writes the specified message in the event log. You specify the message to log.
Eval expression	Evaluates the specified expression and because Log result is checked by default writes the result of the evaluation to the event log. Uncheck Log result to evaluate without logging.
Log result	Becomes enabled when text is entered into Eval expression and is checked by default. If checked, writes the result of the evaluation in the Eval expression to the event log. If unchecked the evaluation is not logged.
Enable group	Enables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.
Disable group	Disables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.

Source Breakpoint Properties dialog box (Breakpoint context menu)

See also

Use the Source Breakpoint Properties dialog box to modify the condition or pass count of source breakpoint. Right-click in the breakpoint gutter in the editor and choose Breakpoint Properties to display.

Filename

Not applicable.

Line number

Not applicable.

Condition

Specifies a conditional expression that is evaluated each time the breakpoint is encountered. 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.

Pass count

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

The debugger increments the pass count each time the line containing the breakpoint is encountered. When the pass count equals the specified number, the debugger pauses program execution. For example, if the pass count is set to 3, you will see 0 of 3, 1 of 3, 2 of 3, then 3 of 3 in the pass count. Program execution stops at 3 of 3.

Because the debugger increments the count 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 number of passes that occurred.

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

Group

When setting a breakpoint using one of the Breakpoint Properties dialogs, you make it a member of a group by entering a group name in the Group field. See Organizing breakpoints into groups. Once your breakpoints are organized into groups, you can disable and enable groups of breakpoints by using the Disable Group and the Enable Group commands on the Breakpoint List context menu (right-click on the Breakpoint List).

Advanced button

Click the Advanced button if you want to associate actions with breakpoints: Enter the appropriate text in each field for each action you want to associate with the breakpoint.

Action	Description
Break	When checked, halts execution; the traditional and default action of a breakpoint.
Ignore subsequent exceptions	When checked, ignore all subsequent exceptions raised by the current process during the current debug session (the debugger will not stop on any exception). Use this with "Handle subsequent exceptions" as a pair. You can surround specific blocks of code with the Ignore/Handle pair to skip any exceptions which occur in that block of code.

Handle subsequent exceptions	When checked, handle all subsequent exceptions raised by the current process during the current debug session (the debugger will stop on exceptions based on the current exception settings in the Tools|Debugger options (Language and OS exception pages). This action does not mean stop on all exceptions no matter what. Use it to turn on normal exception behavior after another breakpoint disabled normal behavior using the Ignore subsequent exceptions option.
Eval expression	Evaluates the specified expression and because Log result is checked by default writes the result of the evaluation to the event log. Uncheck Log result to evaluate without logging.
Log result	Becomes enabled when text is entered into Eval expression and is checked by default. If checked, writes the result of the evaluation in the Eval expression to the event log. If unchecked the evaluation is not logged.
Log result	If checked, writes the result of the evalution (specified in Eval expression) to the event log.
Enable group	Enables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.
Disable group	Disables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.

Add Address Breakpoint dialog box **(Run|Add Breakpoint|Address Breakpoint or Add|Address Breakpoint)**

See also

The Address Breakpoint command displays the Add Address Breakpoint dialog box which you can use to set a breakpoint on a specific machine instruction. When you run your program, the execution point in the CPU window Disassembly pane indicates the breakpoint location. The breakpoint appears in the Code editor, if the address corresponds to a source line, and the Breakpoint List.

Address

Specifies the address for the address breakpoint. Enter the address for the breakpoint. When the address is executed, the program execution halts as modified by the condition and pass count. If the address can be correlated to a source line number, the address breakpoint is created as a source breakpoint. When the breakpoint is inspected in the Breakpoint Properties dialog, a Source Breakpoint Properties dialog will be displayed.

Condition

Specifies a conditional expression that is evaluated each time the breakpoint is encountered. 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.

Pass count

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

The debugger increments the pass count each time the line containing the breakpoint is encountered. When the pass count equals the specified number, the debugger pauses program execution. For example, if the pass count is set to 3, you will see 0 of 3, 1 of 3, 2 of 3, then 3 of 3 in the pass count. Program execution stops at 3 of 3.

Because the debugger increments the count 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 number of passes that occurred.

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

Group

When setting a breakpoint using one of the Breakpoint Properties dialogs, you make it a member of a group by entering a group name in the Group field. See Organizing breakpoints into groups. Once your breakpoints are organized into groups, you can disable and enable groups of breakpoints by using the Disable Group and the Enable Group commands on the Breakpoint List context menu (right-click on the Breakpoint List).

Advanced button

Click the Advanced button if you want to associate actions with breakpoints: Enter the appropriate text in each field for each action you want to associate with the breakpoint.

Action	Description
Break	When checked, halts execution; the traditional and default action of a breakpoint.
Ignore subsequent exceptions	When checked, ignore all subsequent exceptions raised by the current process during the current debug session (the debugger will

	not stop on any exception). Use this with "Handle subsequent exceptions" as a pair. You can surround specific blocks of code with the Ignore/Handle pair to skip any exceptions which occur in that block of code.
Handle subsequent exceptions	When checked, handle all subsequent exceptions raised by the current process during the current debug session (the debugger will stop on exceptions based on the current exception settings in the Tools|Debugger options (Language and OS exception pages). This action does not mean stop on all exceptions no matter what. Use it to turn on normal exception behavior after another breakpoint disabled normal behavior using the Ignore subsequent exceptions option.
Eval expression	Evaluates the specified expression and because Log result is checked by default writes the result of the evaluation to the event log. Uncheck Log result to evaluate without logging.
Log result	Becomes enabled when text is entered into Eval expression and is checked by default. If checked, writes the result of the evaluation in the Eval expression to the event log. If unchecked the evaluation is not logged.
Log result	If checked, writes the result of the evalution (specified in Eval expression) to the event log.
Enable group	Enables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.
Disable group	Disables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.

Address Breakpoint Properties dialog box **(Breakpoint List context menu)**

See also

Use the Address Breakpoint Properties dialog box to change an address breakpoint or set a new one.

Keep existing Breakpoint

Check “Keep existing Breakpoint” to keep the old breakpoint and create a new one. If you do **not** check “Keep existing Breakpoint”, the breakpoint will be changed and the old breakpoint will not be saved.

Address

Specifies the address for the address breakpoint. Enter the address for the breakpoint. When the address is executed, the program execution halts as modified by the condition and pass count. If the address can be correlated to a source line number, the address breakpoint is created as a source breakpoint. When the breakpoint is inspected in the Breakpoint Properties dialog, a Source Breakpoint Properties dialog will be displayed.

Condition

Specifies a conditional expression that is evaluated each time the breakpoint is encountered. 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.

The debugger increments the pass count each time the line containing the breakpoint is encountered. When the pass count equals the specified number, the debugger pauses program execution. For

example, if the pass count is set to 3, you will see 0 of 3, 1 of 3, 2 of 3, then 3 of 3 in the pass count. Program execution stops at 3 of 3.

Because the debugger increments the count 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 number of passes that occurred.

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

Group

When setting a breakpoint using one of the Breakpoint Properties dialogs, you make it a member of a group by entering a group name in the Group field. See Organizing breakpoints into groups. Once your breakpoints are organized into groups, you can disable and enable groups of breakpoints by using the Disable Group and the Enable Group commands on the Breakpoint List context menu (right-click on the Breakpoint List).

Advanced button

Click the Advanced button if you want to associate actions with breakpoints: Enter the appropriate text in each field for each action you want to associate with the breakpoint.

Action	Description
Break	When checked, halts execution; the traditional and default action of a breakpoint.
Ignore subsequent exceptions	When checked, ignore all subsequent exceptions raised by the current process during the current debug session (the debugger will not stop on any exception). Use this with "Handle subsequent exceptions" as a pair. You can surround specific blocks of code with the Ignore/Handle pair to skip any exceptions which occur in that block of code.
Handle subsequent exceptions	When checked, handle all subsequent exceptions raised by the current process during the current debug session (the debugger will stop on exceptions based on the current exception settings in the Tools|Debugger options (Language and OS exception pages). This action does not mean stop on all exceptions no matter what. Use it to turn on normal exception behavior after another breakpoint disabled normal behavior using the Ignore subsequent exceptions option.
Log message	Writes the specified message in the event log. You specify the message to log.
Eval expression	Evaluates the specified expression and because Log result is checked by default writes the result of the evaluation to the event log. Uncheck Log result to evaluate without logging.
Log result	Becomes enabled when text is entered into Eval expression and is checked by default. If checked, writes the result of the evaluation in the Eval expression to the event log. If unchecked the evaluation is not logged.
Enable group	Enables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.
Disable group	Disables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.

Address Breakpoint Properties dialog box (Code editor or CPU window context menu)

See also

Use the Address Breakpoint Properties dialog box to change the condition or pass count of an address breakpoint.

Address

Grayed out.

Condition

Specifies a conditional expression that is evaluated each time the breakpoint is encountered. 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.

The debugger increments the pass count each time the line containing the breakpoint is encountered. When the pass count equals the specified number, the debugger pauses program execution. For example, if the pass count is set to 3, you will see 0 of 3, 1 of 3, 2 of 3, then 3 of 3 in the pass count. Program execution stops at 3 of 3.

Because the debugger increments the count 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 number of passes that occurred.

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

Group

When setting a breakpoint using one of the Breakpoint Properties dialogs, you make it a member of a group by entering a group name in the Group field. See Organizing breakpoints into groups. Once your breakpoints are organized into groups, you can disable and enable groups of breakpoints by using the Disable Group and the Enable Group commands on the Breakpoint List context menu (right-click on the Breakpoint List).

Advanced button

Click the Advanced button if you want to associate actions with breakpoints: Enter the appropriate text in each field for each action you want to associate with the breakpoint.

Action	Description
Break	When checked, halts execution; the traditional and default action of a breakpoint.
Ignore subsequent exceptions	When checked, ignore all subsequent exceptions raised by the current process during the current debug session (the debugger will not stop on any exception). Use this with "Handle subsequent exceptions" as a pair. You can surround specific blocks of code with the Ignore/Handle pair to skip any exceptions which occur in that block of code.
Handle subsequent exceptions	When checked, handle all subsequent exceptions raised by the current process during the current debug session (the debugger will stop on exceptions based on the current exception settings in the

	Tools|Debugger options (Language and OS exception pages). This action does not mean stop on all exceptions no matter what. Use it to turn on normal exception behavior after another breakpoint disabled normal behavior using the Ignore subsequent exceptions option.
Eval expression	Evaluates the specified expression and because Log result is checked by default writes the result of the evaluation to the event log. Uncheck Log result to evaluate without logging.
Log result	Becomes enabled when text is entered into Eval expression and is checked by default. If checked, writes the result of the evaluation in the Eval expression to the event log. If unchecked the evaluation is not logged.
Log result	If checked, writes the result of the evalution (specified in Eval expression) to the event log.
Enable group	Enables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.
Disable group	Disables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.

Add Data Breakpoint dialog box **(Run|Add Breakpoint)**

See also

Use the Add Data Breakpoint dialog box to set a breakpoint on a specific address that halts execution when that address is written to. The breakpoint appears in the Breakpoint List, and, if there is a watch set in the watch view, the item will appear in red.

A data breakpoint is only valid for the current debug session. On the next debug session you must go to the Breakpoint view (Breakpoint List) and re-enable the data breakpoint. You can also re-select Break When Changed from the Watch view (Watch List).

Address

Specifies the address for the data breakpoint. Enter the variable name or address for the data breakpoint. When the address (up to the specified length) is written to, the program execution halts. Valid data names may be entered. For example, if you have an Integer variable X, you can enter X as the address.

Length

Specifies the length of the data breakpoint, beginning at “Address”. This is automatically calculated for standard data types.

Condition

Specifies a conditional expression that is evaluated each time the breakpoint is encountered. 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.

The debugger increments the pass count each time the line containing the breakpoint is encountered. When the pass count equals the specified number, the debugger pauses program execution. For example, if the pass count is set to 3, you will see 0 of 3, 1 of 3, 2 of 3, then 3 of 3 in the pass count. Program execution stops at 3 of 3.

Because the debugger increments the count 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 number of passes that occurred.

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

Group

When setting a breakpoint using one of the Breakpoint Properties dialogs, you make it a member of a group by entering a group name in the Group field. See Organizing breakpoints into groups. Once your breakpoints are organized into groups, you can disable and enable groups of breakpoints by using the Disable Group and the Enable Group commands on the Breakpoint List context menu (right-click on the Breakpoint List).

Advanced button

Click the Advanced button if you want to associate actions with breakpoints: Enter the appropriate text in each field for each action you want to associate with the breakpoint.

Action	Description
Break	When checked, halts execution; the traditional and default action of a breakpoint.
Ignore subsequent exceptions	When checked, ignore all subsequent exceptions raised by the current process during the current debug session (the debugger will not stop on any exception). Use this with "Handle subsequent exceptions" as a pair. You can surround specific blocks of code with the Ignore/Handle pair to skip any exceptions which occur in that block of code.
Handle subsequent exceptions	When checked, handle all subsequent exceptions raised by the current process during the current debug session (the debugger will stop on exceptions based on the current exception settings in the Tools|Debugger options (Language and OS exception pages). This action does not mean stop on all exceptions no matter what. Use it to turn on normal exception behavior after another breakpoint disabled normal behavior using the Ignore subsequent exceptions option.
Eval expression	Evaluates the specified expression and because Log result is checked by default writes the result of the evaluation to the event log. Uncheck Log result to evaluate without logging.
Log result	Becomes enabled when text is entered into Eval expression and is checked by default. If checked, writes the result of the evaluation in the Eval expression to the event log. If unchecked the evaluation is not logged.
Log result	If checked, writes the result of the evalution (specified in Eval expression) to the event log.
Enable group	Enables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.
Disable group	Disables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.

Data Breakpoint Properties dialog box **(Breakpoint List context menu)**

See also

Use the Add Data Breakpoint dialog box to add a data breakpoint. The breakpoint appears in the Breakpoint List, and, if there is a watch set in the watch view, the item will appear in red.

A data breakpoint is only valid for the current debug session. On the next debug session you must go to the Breakpoint view (Breakpoint List) and re-enable the data breakpoint. You can also re-select Break When Changed from the Watch view (Watch List).

Address

Specifies the variable name or address for the data breakpoint. Enter the address for the data breakpoint. When the address (up to the specified length) is written to, the program execution halts.

Length

Specifies the length of the data breakpoint, beginning at “Address”.

Condition

Specifies a conditional expression that is evaluated each time the breakpoint is encountered. 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.

The debugger increments the pass count each time the line containing the breakpoint is encountered. When the pass count equals the specified number, the debugger pauses program execution. For example, if the pass count is set to 3, you will see 0 of 3, 1 of 3, 2 of 3, then 3 of 3 in the pass count. Program execution stops at 3 of 3.

Because the debugger increments the count 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 number of passes that occurred.

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

Group

When setting a breakpoint using one of the Breakpoint Properties dialogs, you make it a member of a group by entering a group name in the Group field. See Organizing breakpoints into groups. Once your breakpoints are organized into groups, you can disable and enable groups of breakpoints by using the Disable Group and the Enable Group commands on the Breakpoint List context menu (right-click on the Breakpoint List).

Keep existing Breakpoint

Check “Keep existing Breakpoint” to keep the old breakpoint and create a new one. If you do **not** check “Keep existing Breakpoint”, the breakpoint will be changed and the old breakpoint will not be saved.

Advanced button

Click the Advanced button if you want to associate actions with breakpoints: Enter the appropriate text in each field for each action you want to associate with the breakpoint.

Action	Description

Break	When checked, halts execution; the traditional and default action of a breakpoint.
Ignore subsequent exceptions	When checked, ignore all subsequent exceptions raised by the current process during the current debug session (the debugger will not stop on any exception). Use this with "Handle subsequent exceptions" as a pair. You can surround specific blocks of code with the Ignore/Handle pair to skip any exceptions which occur in that block of code.
Handle subsequent exceptions	When checked, handle all subsequent exceptions raised by the current process during the current debug session (the debugger will stop on exceptions based on the current exception settings in the Tools| Debugger options (Language and OS exception pages). This action does not mean stop on all exceptions no matter what. Use it to turn on normal exception behavior after another breakpoint disabled normal behavior using the Ignore subsequent exceptions option.
Log message	Writes the specified message in the event log. You specify the message to log.
Eval expression	Evaluates the specified expression and because Log result is checked by default writes the result of the evaluation to the event log. Uncheck Log result to evaluate without logging.
Log result	Becomes enabled when text is entered into Eval expression and is checked by default. If checked, writes the result of the evaluation in the Eval expression to the event log. If unchecked the evaluation is not logged.
Enable group	Enables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.
Disable group	Disables all breakpoints which are members of the specified group. Select the group name. See Organizing breakpoints into groups.

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.

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 Debug|Add Watch At Cursor from the Code editor context menu.
- Choose Run|Add Watch.
- Choose Add Watch from the Watch List context menu.
- Right-click an existing watch in the Watch List and choose Edit Watch from the Watch List context menu.

Watch Properties

You can set the following properties for a watch expression:

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, or specifies the number of elements in an array when the watch expression represents an array.

When you watch an array and specify the number of elements as a repeat count, the Watch List 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.

- This option takes affect only when you select Floating Point as the Display format. For more information, see Formatting watch expressions.

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 the debugger 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.

Allow Function Calls

When set, the watch is evaluated even if doing so would cause function calls. This option is off by default for all watches. When off, watches that would make function calls are not evaluated but instead generate the error message ”Inaccessible value.”

Display format radio buttons

To format the display of a watch expression, select a radio button.

- For more information, see Formatting watch expressions.

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.

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, the display format changes from decimal to hexadecimal.

Character

Shows special display characters for ASCII 0 to 31 (displayed as #$0, #$1F, and so on). This format type affects characters and strings.

String

Shows characters for ASCII 0 to 31 in the Pascal #nn notation (#$0, and so on.) 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/Structure

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.

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

The debugger 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 Debug|Evaluate/Modify from the Code editor context menu.

Dialog box options

Expression

Specifies the variable, field, array, or object to evaluate 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 tool button

Evaluates the expression in the Expression edit box and displays its value in the Result edit box.

Modify tool button
Changes the value of the expression in the Expression edit box using the value in the New Value edit box.

Watch tool button
Creates a watch for the expression you have selected.

Inspect tool button
Opens a new Inspector window on the data element you have selected. This is useful for seeing the details of data structures, classes, and arrays.

Help button
Displays Help on the dialog box.

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

Inspector window

See also

The number of tabs and the appearance of the data in the Inspector window depends on the type of data you inspect. You can inspect the following types of data: arrays, classes, constants, functions, pointers, scalar variables, **integer**, and so on), and interfaces. The Inspector window contains three areas:

- The top of the Inspector window shows the name, type, and address or memory location of the inspected element, if available. (When inspecting a function call that returns an object, record, set, or array, the debugger displays “In debugger” in place of the temporarily allocated address.)
- The middle pane contains one or more of the following views depending on the type of data you inspect. To change the view, click its tab.

Data	Shows data names (or class data members) and current values.
Methods	This view appears only when you inspect a class, or interface and shows the class methods (member functions) and current address locations.
Properties	This view displays only when you inspect an Object class with properties (such as a VCL Object) and shows the property names and current values. The inspector does not automatically report the values of all properties because a function called to evaluate certain properties may have side effects that can affect the behavior of the program you are debugging. For example, if you evaluate certain properties before an object is fully constructed or before the object's associated window is created, some of the functions called will actually try to create the window. When your program actually creates the window, the app will likely throw an exception. Therefore, for a property whose getters are member functions, the inspector window shows the value as <dynamic> on the properties page. To see the value of the property, click the **?** button that appears next to <dynamic>. The debugger will continue to recalculate the value of the property each time the process stops (such as after a step or at a breakpoint). If you click the **?** button again, the debugger stops recalculating the value of the property and again will show <dynamic> as the property's value each time the process stops.

- The bottom of the Inspector window shows the data type of the item currently selected in the middle pane.

Inspector window commands

Right-click the Inspector window to access the following commands:

Change	Lets you assign a new value to a data item. An ellipsis (…) appears next to an item that can be changed. You can click the ellipsis as alternative to choosing the change command.
Show Inherited	Switches the view in the Data, Methods, and Properties panes between two modes: one that shows all intrinsic and inherited data members or properties of a class, or one that shows only those declared in the class.
Show Fully Qualified Names	Shows inherited members using their fully qualified names.
Inspect	Opens a new Inspector window on the data element you have selected. This is useful for seeing the details of data structures, classes, and arrays.
Descend	Same as the Inspect command, except the current Inspector window is replaced with the details that you are inspecting (a new Inspector window is not opened). To return to a higher level, use the history list.
New Expression	Lets you inspect a new expression.

Type Cast	Lets you specify a different data type for an item you want to inspect. Type casting is useful if the Inspector window contains a symbol for which there is no type information, and when you want to explicitly set the type for untyped pointers.

Multiple process debugging

Inspectors are associated with the thread that was active when they were created. When a thread terminates, only the inspectors that were created while the thread was active are destroyed.

Project projectname raised too many consecutive exceptions: application defined exception (code xxxx) at xxxx. Process stopped. Use Step or Run to continue.

The debugger tracks all exceptions that occur including those which may be handled by your application. Although this is likely to indicate a program failure, it need not always indicate a failure. This informational message occurs when your application encounters a large number of a specific system exceptions at the same address without any intervening exceptions (including those that result from stepping and hitting breakpoints).

For example, the following code will trigger the notification:

```
while true do
    IsBadReadPtr(Pointer(13), 4); // AV at 0x77f1b347
```

but this will not:

```
while true do
begin
    IsBadReadPtr(Pointer(13), 4); // AV at 0x77f1b347
    IsBadWritePtr(Pointer(13), 4); // AV at 0x77f1b34c, resets counter
end;
```

To resume execution of the program, you can typically use the Step or Run debugging commands.

Note: Your program may be in a state where attempting to continue will result in the error again. In this situation, you may need to terminate the application and investigate the cause of the exception.

Source File not Found: sourcefile

When the debugger can't find a file, it displays this dialog box. Following are descriptions of the items in the dialog box.

Item	Description
Path to source file	The name of the source file it can't find is shown in the title bar and in the edit control. Click the Browse button to browse for the source file or type the full path name of the source file.
OK button	The file specified is validated to make sure it exists. If not, the dialog will give an error and will not close. If the file exists, the dialog box closes and the file is opened. The path to this file is updated so the debugger will locate it in the future.
Add directory to Debug Source Path check box	If checked and you press OK, the path to the file specified is appended to the end of the debug source path (in Project|Options|Directories/Conditionals).
Cancel button	The debugger does not try to open the file now.
Ignore button	The debugger does not try to open the file now. It calls SetFileName with an empty string to tell the evaluator to ignore the source file for the rest of the debug session.
Help button	Displays help about the dialog box.

Stream read error

See also

This error is usually caused by corruption of project files automatically created by Delphi. Stream read error may be generated when the IDE is attempting to restore a state from disk file(s). If the error occurs on startup of the IDE, look for corrupt files used by the IDE to store settings, that is .DSK .DRO .DCT and/or especially .DMT files. If it happens only when opening one project, look for corrupt projects files (such as the.DFM).

Decision Query Editor dialog box

The Decision Query Editor dialog box defines queries for the active decision query component. This dialog box specifies the database, tables, available fields, dimensions, and summaries for decision cubes bound to the decision query. It also displays the defined query in ANSI-92 SQL syntax.

To display the Decision Query Editor dialog box, apply a decision query component to a form, then right-click and choose Decision Query Editor from the menu.

Dialog box options

Dimensions/Summaries tab

Specifies the database, tables, available fields, dimensions, and summaries for decision cubes bound to the decision query.

SQL Query tab

Displays the defined query in ANSI-92 SQL syntax. You can edit the query by editing the ANSI-92 SQL statements.

Query Builder button

Displays the Visual Query Builder to select and join data tables.

Dimensions/Summaries tab (Decision Query editor)

Use this tab of the Decision Query Editor dialog box to define queries for the active decision query component. This tab specifies the database, tables, available fields, dimensions, and summaries for decision cubes bound to the decision query.

To display the Decision Query Editor dialog box, apply a decision query component to a form, then right-click and choose Decision Query Editor from the menu. Then, click the Dimension/Summaries tab if it isn't already visible.

To display a text version of the defined query, click the SQL Query tab.

Dialog box options

List Of Available Fields
Fields available for use in the decision query. If you're using more than one table and need to create a join, you can click the Query Builder button to launch the Visual Query Builder.

All Fields/Query Fields button
Toggles between displaying all fields in the selected table and only fields selected by the active query.

Dimensions
Lists fields selected as decision cube dimensions. To add a field to the Dimensions list, select a field in the List Of Available Fields, then click the right-arrow. To remove a field from the Dimensions list, select it in the list, then click the left-arrow.

Summaries
Lists fields selected as decision cube summaries.

To add a field to the Summaries list, select a field in the List Of Available Fields, then click the right-arrow. You will then see a list of summary operators (sum, count, or average). Choose the operator that you want to use to summarize the field values. If the selected operator is not appropriate for the type of the selected field, you will see an error message. If you want to let the decision cube compute averages (allowing for averaged summaries that can be subtotaled, drilled, and pivoted correctly), you must add both a count and a sum summary for the field. If you are creating several averages (over fields that do not contain blank values), you can check the Count (*) for averages check box instead of adding a count summary for each field you will average.

To remove a field from the Summaries list, select it in the list, then click the left-arrow.

Table
Shows the active table from those included in the active database. You can select a different table from this dropdown list.

Database
Shows the active database from those included in the query. You can select a different database from this dropdown list. If you are using local tables, you can type in the path to the directory that contains the tables.

SQL Query tab (Decision Query editor)

Use this tab of the Decision Query Editor dialog box to display SQL queries for the active decision query component.

If a query hasn't yet been defined, you can enter one directly in ANSI-92 SQL. Or, you can define it visually by selecting a database, tables, and fields on the Dimensions/Summaries tab.

To display the Decision Query Editor dialog box, apply a decision query component to a form, then right-click and choose Decision Query Editor from the menu. Then, click the SQL Query tab.

Dialog box options

Query Text

Displays SQL statements that define the current decision query. The defined query appears in ANSI-92 SQL syntax. You can edit the query directly by editing the ANSI-92 SQL statements; they are automatically converted to and from any appropriate dialect used to communicate with the server.

Edit Query/Edit Done button

When the Edit Query button is active, text appears in the Query Text window. You can type over it to edit it.

When you begin typing, button text changes to Edit Done. Click the button when your edit is complete. You can click Cancel Edit to restore the original query text.

Cancel Edit button

Cancels the current text edit and restores the original query.

Decision Cube Editor dialog box

The Decision Cube Editor dialog box defines active dimensions and summaries for the active decision cube component. These settings are reflected in decision pivots bound to the decision cube.

To display the Decision Cube Editor dialog box, apply a decision cube component to a form, bind it to a decision query, then right-click the decision cube and choose Decision Cube Editor from the menu.

Dialog box options

Dimension Settings tab

Specifies the display name, type, active type, format, grouping, and initial value for fields supplied to the decision cube from the decision query component.

Memory Control tab

Displays the following settings for memory protection:

- Decision cube dimension, summary, and cell maximums
- Designer data display choices

Dimension Settings tab (Decision Cube editor)

Use this tab of the Decision Cube Editor dialog box to specify the display name, type, active type, format, grouping, and initial value for fields supplied to the decision cube from the decision query component.

To display the Decision Cube Editor dialog box, apply a decision cube component to a form, bind it to a decision query, then right-click the decision cube and choose Decision Cube Editor from the menu. Then, click the Dimension Settings tab if it isn't already visible.

Dialog box options

Available Fields
Fields supplied to the decision cube component by the decision query component. When a field in the list is highlighted, its settings appear in the text boxes and lists on the right side of the dialog box.

Display Name
The name to appear in decision pivot, decision grid, and decision graph labels for the highlighted field.

Type
Whether the highlighted field is a dimension or a summary (for information only, not editable).

Active Type
When the information for the highlighted field is loaded: As Needed, when required for display; Active, all the time; Inactive, never.

Format
The format string that describes how to display values for the highlighted field.

Grouping
Whether to display all values or ranges of values. Use None to display all values. Use Year, Quarter, or Month to display a range of dates. Use Single Value for a single-dimension display. If custom ranges have been added by an application developer, they also appear in the list.

Initial Value
The starting value for a date or custom range, or the single value to display.

Memory Control tab (Decision Cube editor)

Use this tab of the Decision Cube Editor dialog box to change the following settings for memory protection:

- Decision cube dimension, summary, and cell maximums
- Designer data display choices

To display the Decision Cube Editor dialog box, apply a decision cube component to a form, bind it to a decision query, then right-click the decision cube and choose Decision Cube Editor from the menu. Then, click the Memory Control tab.

Dialog box options

Cube Maximums

Sets maximums and displays current values for decision cube dimensions, summaries, and cells.

Maximum
Sets the maximum allowable dimensions, summaries, and cells for the selected decision cube. The lower the number, the less memory is used.

Current
The current number of dimensions, summaries, and cells in use by the decision cube (for information only, not editable).

Active+Needed
The number of dimensions and summaries that have Active Type set to Active or As Needed on the Dimension Settings tab of the Decision Cube Editor dialog box, and the number of cells required to display them (for information only, not editable).

Active
The number of dimensions and summaries that have Active Type set to Active on the Dimension Settings tab of the Decision Cube Editor dialog box, and the number of cells required to display them (for information only, not editable).

Get Cell Counts button
When clicked, runs a query to estimate the number of cells used.

Designer Data Options

Saves time and memory by displaying only the specified data at design time.

Display Dimension Names
When checked, displays only dimension and summary names at design time. No values appear.

Display Names And Values
When checked, displays dimension and summary names and values at design time. No totals appear.

Display Names, Values, And Totals
When checked, displays dimension and summary names, values, and totals at design time.

Runtime Display Only
When checked, displays dimension and summary names, values, and totals only at runtime. None of this data appears at design time.

About the Data Module Designer

See also

Use the Data Module Designer to create and maintain data modules. The Data Module Designer opens when you choose File|New|Data Module or when you reopen any existing data module.

The Data Module Designer is divided into two panes:

- The left pane, called the Tree view, shows parent-child relationships among data-access components.
- The right pane has two tabs: Components and Data Diagram.
- Click **Components** to view the module's components as they would appear in the Form Designer (represented by their Component-palette icons).
- Click Data Diagram to see dependencies among components. Components do not appear on the Data Diagram page until you drag them from the Tree view on the left.

You can add components to a data module by selecting them on the Component palette and clicking in the Tree view or Components page of the Data Module Designer. You can also use the Object Inspector to set component properties while working in the Designer.

The Data Module Designer stores information about each module's Data Diagram page in a file that ends with the .DTI extension. DTI files have no effect on compilation.

Printing

Both the Tree view and Data Diagram page are printable. To print, make sure that the view you want to print is active, then choose File|Print (or right-click and choose Print). The Components page is not printable.

Note: If you find incorrectly positioned text in printout from the Data Module Designer, you may need to change your printer driver settings. Open the Printers folder from the Windows Control Panel and select the printer you are using, then look for Print Text as Graphics; this option is usually found under Properties or Document Defaults. Make sure that Print Text as Graphics is enabled.

Using the Tree view

See also

In the Tree view, you can drag and drop components to change their relationships. For example, you can drag

- data sources from one table to another.
- databases from one session to another.
- datasets (such as tables and queries) from one database to another.

When you drop a component from the Component palette onto the Tree view, it becomes, if appropriate, associated with the item you drop it on. For example, if you drop a new data source onto a table, the data source automatically becomes a child of that table and its DataSet property is set to the name of the table.

When you delete an item in the Tree view, the Data Module Designer asks if you want to delete its children (the nodes under it) as well.

When you right-click on a component in the Tree view, you'll see an abridged version of the component's context menu. To access the full menu, right-click on the same component in the Components page.

If a node in the Tree view has a red outline, that means that the item is defective or not completely defined. For example, a DataSource whose DataSet property has no value appears with a red outline.

Some nodes in the Tree view are shown with their icons grayed out. These nodes represent "implied" components. For example, a dataset has a default session associated with it. The default session is created by your application at runtime, but it appears (grayed) at design time in the Tree view.

Using the Data Diagram page

See also

The Data Diagram page provides visual tools for setting up relationships among database elements. It is also a documentation tool, since it illustrates these relationships schematically and lets you add comments to the diagram; you can even print the diagrams.

Components do not appear on the Data Diagram page until you drag them from the Tree view. You can place a component and all of its siblings on the Data Diagram page by pressing the <control> key while dragging the component:

- To arrange a component and its siblings horizontally on the page, press **Ctrl** while dragging.
- To arrange a component and its siblings vertically on the page, press **Ctrl+Shift** while dragging.
- If the components are already on the page, you can still rearrange them by pressing **Ctrl** or **Ctrl+Shift** while dragging one of them.

The Data Diagram page shows five types of relationship:

- Property (line with solid arrow)
- Master-detail (line with asymmetric "drum" glyphs at either end)
- Lookup (line with "eye" glyph at end)
- Allude (arrow)
- Parent (line with hollow arrow)

To delete any relationship, right-click over the line or arrow and choose Remove Relationship.
You can also add Comment blocks to the Data Diagram page.
All elements in a data diagram can be moved and resized with the mouse. You can bend lines and arrows by clicking in the middle and dragging, and you can change the color and other properties of some elements (including lines) by right-clicking and selecting a submenu. To display a complete list of fields for a dataset object, right-click on the object and choose Show Field Info.

When you remove components from the Data Diagram page, you are removing them from the data diagram only; they can be restored by dragging them from the Tree view back to the Data Diagram page. But when you remove relationships (represented by lines), you are deleting them from your project completely.

Property relationships

See also

Property relationships include all properties of a component that refer to other components. For example, if DataSource1.DataSet is set to Table1, then DataSource1 and Table1 stand in a property relationship.

Property relationships are represented by solid arrows pointing away from the component that has the property and toward the component referred to by the property. The name of the property is shown as the caption of the arrow.

To create a property relationship,

1. Select the Property tool from the left side of the Data Diagram page.
2. Click on the component that has the property and drag to the component that will be referred to by the property. (For example, you would drag *from* a data source *to* a table.)

   If the selected component has only one property that can reference the target, you don't need to provide any additional information. If more than one property could point to the target, a pop-up menu is displayed allowing you to select which property to set; properties that already have values appear with check marks next to them.

Master-detail relationships

See also

Master-detail relationships are represented by lines with asymmetric "drum" glyphs at either end. The larger drum indicates the master dataset and the smaller drum indicates the detail dataset. The value of the detail dataset's MasterFields property is shown as the caption of the line.

To create a master-detail relationship,

1. Select the Master-Detail tool from the left side of the Data Diagram page.
2. Click on the the table component that you want to make into the detail dataset and drag to the master dataset.

When you create a master-detail relationship, the Field Link Designer dialog usually appears requesting additional information.

The Data Module Designer automatically generates required data sources when you create a master-detail relationship. If you later remove the master-detail relationship, it does *not* delete these data sources from your project. If, however, you delete a required data source, the master-detail relationship is automatically removed.

Lookup relationships

See also

Lookup relationships are represented by lines with an "eye" glyph next to the lookup dataset. The name of the lookup field is shown as the caption of the line.

To create a lookup relationship,

1. Select the Lookup tool from the left side of the Data Diagram page.
2. Click on the the dataset for which you want to create a lookup field and drag to the lookup dataset.

When you create a lookup relationship, the New Field dialog appears requesting additional information. After you fill in the dialog and click OK, a lookup field is created. If you remove a lookup relationship, the lookup field is deleted.

Allude relationships

See also

An allude relationship is simply an arrow pointing from one item in the Data Diagram page to another. Like a comment, an allude is a form of documentation and has no effect on the behavior of your program. You can use alludes in conjunction with comment blocks to annotate your data diagrams.

To create an allude relationship,

1. Select the Comment Allude tool from the left side of the Data Diagram page.
2. Click on an item in the Data Diagram page and drag to another item.

You can reposition the ends of an allude arrow as well as bend the arrow in the middle. You can also change the ends of the arrow by right-clicking on it and selecting Starts With or Ends With.

Comment blocks

See also

Comment blocks are rectangular areas that contain text. To add a comment block to a data diagram,

1. Select the Comment Block tool from the left side of the Data Diagram page.
2. With the mouse pointer in the client area of the page, press the left mouse button, drag the mouse, then release the mouse button.

To add or edit text in a comment block, click twice in the comment block and type.

You can use comment blocks with allude arrows to annotate your data diagrams. Comment blocks are often useful for documenting non-database items—such as menus, common dialogs, and system components—in your data modules.

Parent relationships

See also

When one element appears below another in the Tree View hierarchy, this parent-child relationship is represented on the Data Diagram page by a line with hollow arrow pointing from the child to the parent. But if the two elements also stand in a property relationship, their parent-child relationship is not displayed separately.

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 the Code editor. Each file opens on a new page of the Code editor, and each page is represented by a tab at the top 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 the Code editor, you can do one of the following:

- Click on the name of a file in the Project Manager.
- With a project open, choose Project|View Source.
- Open a file using File|Open.
- Choose View|New Edit Window.

The New Edit 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.

Class completion

See also

Class completion automates the definition of new classes by generating skeleton code for the class members you declare. Here's how it works:

- Place the cursor anywhere within a class declaration in the **interface** section of a unit; press Ctrl+Shift+C, or right-click and select Complete Class at Cursor. Delphi automatically adds private **read** and **write** specifiers to the declarations for any properties that require them, then adds skeleton code in the **implementation** section for all the class's methods.

For example, if you type the following code in the **interface** section—

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

—and press **Ctrl+Shift+C**, Delphi adds **read** and **write** specifiers to your **interface** declaration—

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

—and adds

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

end;

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

to the **implementation** section of the unit.

You can also use class completion to fill in **interface** declarations for methods you define in the **implementation** section:

- Place the cursor within a method definition in the **implementation** section and press Ctrl+Shift+C (or right-click and select Complete Class at Cursor). If there is no prototype for the method in the **interface** section, Delphi adds one.

If your declarations and implementations are sorted alphabetically, class completion maintains their sorted order. Otherwise, new routines are placed at the end of the implementation section of the unit and new declarations are placed in private sections at the beginning of the class declaration.

If you want class completion to complete property declarations, make sure that Finish Incomplete Properties is checked on the Explorer page of Tools|Environment Options.

Code Explorer

See also

The Code Explorer makes it easy to navigate through your unit files. By default, the Code Explorer is docked to the left of the Code editor.

- To close the Code Explorer, undock it and click the upper right corner.
- To reopen the Code Explorer, choose View|Code Explorer from the main menu or right-click in the Code editor and choose View Explorer.

The Code Explorer window contains a tree diagram that shows all the types, classes, properties, methods, global variables, and global routines defined in your unit. It also shows the other units listed in the **uses** clause. You can expand or collapse the nodes on the tree.

The Code Explorer uses the following icons:

 Classes

 Interfaces

 Units

 Constants or variables (including fields)

 Methods or routines: Procedures (green)

 Methods or routines: Functions (yellow)

 Properties

 Types

Whichever unit file is open in the Code editor is also open in the Code Explorer.

- To toggle between the Code Explorer and the Code editor, press Ctrl+Shift+E (or right-click and choose View Editor).
- The Code Explorer supports incremental searching. To search for a class, property, method, variable, or routine, just type its name.
- When you select an item in the Code Explorer, the cursor moves to that item's implementation in the Code editor. When you move the cursor in the Code editor, the highlight moves to the appropriate item in the Code Explorer.
- To add or rename an item, right-click the appropriate node in the Code Explorer and choose New or Rename from the menu.

To adjust Code Explorer settings, choose Tools|Environment Options|Explorer.

Use the Code Explorer with class completion and module navigation to automate repetitive coding tasks.

Module navigation

See also

Navigate quickly through your unit files by pressing Ctrl+Shift and the arrow keys.

- Place the cursor on the prototype of any method or global procedure in the **interface** section of a unit. Then press Ctrl+Shift+Up Arrow or Ctrl+Shift+Down Arrow to move to the procedure's implementation.
- Press Ctrl+Shift+Up Arrow or Ctrl+Shift+Down Arrow to toggle between the **interface** and **implementation** sections.

You can also set your own bookmarks by right-clicking in the Code editor and choosing Toggle Bookmarks. To jump to a bookmark, right-click and choose Goto Bookmarks.

Code browser

See also

In the Code editor, hold down the Ctrl key while passing the mouse over the name of any class, variable, property, method, or other identifier. The mouse pointer turns into a hand and the identifier appears highlighted and underlined; click on it, and the Code editor jumps to the declaration of the identifier, opening the source file if necessary. You can do the same thing by right-clicking on an identifier and choosing Find Declaration.

Use Tooltip Symbol Insight to see where an identifier is declared before you jump to it.

Use the navigation buttons in the upper right corner of the Code editor to browse forward and backward through the files you've opened.

The Code browser can find and open only units in the project's Search path or Debug Source path, or in Delphi Browsing or Library path. Directories are searched in the following order:

1. The project Search path (Project|Options|Directories/Conditionals).
2. The project Source path (Project|Options|Directories/Conditionals).
3. The global Browsing path (Tools|Environment Options|Library).
4. The global Library path (Tools|Environment Options|Library). The Library path is searched only if there is no project open in the IDE.

The Code browser cannot find identifiers declared in new, unsaved unit files.

Note: Code browsing does not work in package projects.

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.

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 property, event, method, property, procedure or type for which you want Help, then press *F1*.
- Right-click the Code editor, then choose Topic Search from the context menu.

If Help is not available for the specific topic you selected, Help displays a message reading, "Help Topic Not Found." 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.

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, 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

Code editor context menu

See also

The Code editor context menu 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.

All possible context menu commands are listed below (some appear only at certain times). To view detailed information on a Code editor context menu command, click that command:

Enabled
Breakpoint Properties
Find Declaration
Close Page
Open File At Cursor
New Edit Window
Browse Symbol At Cursor
Topic Search
Add to Interface
Expose as CORBA Object
Complete Class at Cursor
Add To-Do Item
Cut
Copy
Paste
Toggle Bookmarks
Goto Bookmarks
Debug
View As Form
Read Only
Message View
View Explorer
Properties

To display the Code editor context menu, do one of the following:

- Right-click anywhere in the Code editor window.
- Press *Alt+F10* when the Code editor window is active.

Breakpoint properties **(Code editor or CPU Dissassembly Pane context menu)**

Breakpoint properties context menu

Choose Properties from the Code editor context menu to open the Source Breakpoint Properties dialog that allows you to modify source breakpoints.

Choose Properties from the CPU Dissassembly pane context menu to open an Address Breakpoint Properties or a Source Breakpoint Properties dialog box that allows you to modify address or source breakpoints. This menu option is only available by right-clicking in the gutter on an address which has a breakpoint.

Close Page **(Code editor context menu)**

Code editor context menu

Choose Close Page from the Code editor context menu 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 Project As dialog box, where you can enter a name for the project.

Open File At Cursor **(Code editor context menu)**

Code editor context menu

Choose Open File At Cursor from the Code editor context menu 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. Choose Options|Project from the main Delphi menu.
2. Click the Directories/Conditionals page.
3. Set unit and include directories as you want.
4. Choose OK to put your choices into effect.

New Edit Window **(Code editor context menu)**

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.

Browse Symbol At Cursor **(Code editor context menu)**

Choose Browse Symbol At Cursor from the Code editor context menu to open the Symbol Explorer.

Topic Search **(Code editor context menu)**

Choose Topic Search from the Code editor context menu to display a Help window for the word or token at the cursor in the Code editor.

If no Help topic exists, the Search dialog box is displayed, with the closest match highlighted.

Add To-Do Item **(Code editor context menu)**

See also

Right-click on the code editor and choose Add To-Do Item (or press **Ctrl+Shift+T**) to add a To-Do List item within the currently displayed code module. The Add To-Do List Item dialog box is displayed where you add the item. You can also specify the priority, owner, and category of the item. The item is added at the current cursor position in the source code and is shown in the to-do list (choose View|To-Do List).

Toggle Bookmarks **(Code editor context menu)**

Choose Toggle Bookmarks from the Code editor context menu to set or clear up to 10 bookmarked locations In each file you have open in the Code editor. Bookmarks let you save your place within a long text file. You can also press Ctrl+K and the number of the bookmark to set or change the location of a bookmark. When a bookmark is set, you see a gray box in the left margin of the Code editor with the bookmark number in it.

Goto Bookmarks **(Code editor context menu)**

Choose Goto Bookmarks from the Code editor context menu to display a list of bookmarked locations you can jump to in the Code editor. You can also jump to bookmarks by typing CTRL+the number of the bookmark.

Message View **(Code editor context menu)**

Choose Message View to toggle the message window at the bottom of the Code editor. When you compile an application, errors or warning messages are displayed in the message window; and when you conduct a search, search results are displayed in the message window.

The message window context menu commands are listed below.

View Source	Scrolls the Code editor to the location of the error message or search result that is selected in the message window.
Edit Source	Scrolls the Code editor to the location of the error message or search result that is selected in the message window, and makes the Code editor active.
Clear Compiler Messages	Clears only compiler messages from the message window.
Clear Search Results	Clears only search result messages from the message window.
Save messages	Lets you save the messages in a file.
View editor	Displays the editor (useful if the Message View is undocked).
Dockable	Lets you make the Message View either dockable or not. It's docked to the Code editor by default.

Debug **(Code editor context menu)**

Choose Debug from the Code editor context menu to select the following debugger commands:

Toggle Breakpoint
Run to Cursor
Inspect
Goto Address
Evaluate/Modify
Add Watch at Cursor
ViewCPU

Debug|Toggle Breakpoint (Code editor context menu)

See also Code editor context menu

Choose Toggle Breakpoint from the Code editor context menu 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 in the gutter (left margin of code editor) and choose Breakpoint Properties. You can also choose Properties from the Breakpoint List window context menu.

See also

Setting breakpoints

Using breakpoints

About the integrated debugger

Debug|Run To Cursor **(Code editor context menu)**

See also Code editor context menu

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 integrated debugger

Debug|Inspect (Code editor context menu)

Code editor context menu

Choose Inspect to open an Inspector window for the term highlighted (or at the insertion point) in the Code editor. If the insertion point is on a blank space when you choose this command, an Inspect input dialog displays where you can enter an expression you want to inspect.

This command is only available when the integrated debugger is paused in a program you are debugging, such as when

- You are stepping through code.
- Your program is stopped at a breakpoint.
- You first choose Run|Run and then choose Run|Pause.

An alternate way to open the Inspect input dialog is to choose Run|Inspect.

Debug|Goto Address **(Code editor context menu)**

The Go to Address command prompts you for a new area of memory to display in the Disassembly pane of the CPU window. Enter any expression that evaluates to a program memory location. Be sure to precede hexadecimal values with $. This command will only show the address in the editor if the source for the address entered can be found. It will open the CPU view, if source cannot be found. If the CPU view is already open, it becomes the active view.

Note: This command is available only while you run your program from the IDE.

Debug|Evaluate/Modify **(Code editor context menu)**

See also Code editor context menu

The Evaluate/Modify command opens the Evaluate/Modify dialog box, which lets you evaluate or change the value of an existing expression. From the code editor, this context menu will use highlighted text, or text at the cursor position, and automatically evaluate it.

An alternate way to perform this command is:

- Choose Run|Evaluate/Modify.

See also

Evaluating and modifying expressions

About the integrated debugger

Debug|Add Watch At Cursor **(Code editor context menu)**

See also Code editor context menu

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 to display and manage the current list of watches.

Watch Properties only open if the cursor is on whitespace. Otherwise, the expression highlighted, or at the cursor position, is automatically added as a watch.

Alternate ways to perform this command are:

- Choose Run|Add Watch from the Code editor context menu.
- Choose Add Watch from the Watch List context menu.
- Right-click an existing watch in the Watch List and choose Edit Watch from the Watch List context menu.

Debug|View CPU (Code editor context menu)

Code editor context menu

The View CPU command opens the CPU Window, for debugging a specific low-level aspect of an application such as a contents of the program stack, registers or CPU flags, memory dumps, or assembly instructions disassembled from the application's machine code.

Read Only **(Code editor context menu)**

Code editor context menu

Choose Read Only from the Code editor context menu 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 context menu and "Read only" is displayed in the Code editor status line.

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

Read File As Block dialog box

See also

This dialog box enables you to specify the filename and location of an operating system file containing a block of Object Pascal source code that 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*

Glossary

A

abstract
accelerator key
actual parameter
actual variable
ADO
alias
ancestor
application
array
ASCII

B

base type
batch operation
BDE
BDE Adminstrator
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
COM
compile
compiler directive
compile time
compile-time error
complete evaluation

component
conditional symbol
connection component
const parameter
constant
constant address expression
container application
container component
context menu
control
CORBA

D

data
data access component
data-aware
data control component
data module
data packet
data type
database
database server
dataset
DDE client
DDE conversation
DDE server
default ancestor
default event
default new form
default new project
delta packet
derive
descend
descendant
design time
design-time package
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
grandparent
grid

H
handling exceptions
header
heap
heap suballocator
help context
Hint
host type
HTML

IJK
IDAPI
identifier
IDL
implementation
include file
index
index type

OLE server
ORB
ordinal
override
owner

P

package
parameter
parent
pixel
pointer
power set
primary index
private
private part
procedure
procedure header
program
project
project directory
project file
project group
property
protected
protected block
public
published

Q

qualified identifier
qualified method identifier
qualifier
query

R

raise
real
record
record type
recursion
relational database
remote data module
report
root class
routine
row

S

T

U

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 runtime, as distinguished from the definition of that variable within the program. A location in memory used for storage purposes, as distinguished from an identifier.

ADO
ActiveX Data Objects: a set of ActiveX components for using Microsoft's OLEDB to access and modify database information.

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 class from which another class 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. Many components in Delphi use 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 Administrator to change the settings in this configuration file.

BDE Administrator
A program that enables you to change the settings in the BDE configuration file, usually called IDAPI.CFG. The executable file is named BDEADMIN.EXE. Formerly called the BDE Configuration Utility.

BDE Configuration Utility
See BDE Administrator.

BLOB

Binary large object. BLOB data is indeterminate in size and is not stored directly in the records of the database table.

Many database tables use specific field types to contain BLOB data. For example, 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
An 8-bit wide data type capable of holding a value from 0 to 255.

callback routines
Routines in your application that are passed to a procedure or function and called from within that procedure or function's body. For example, EnumFonts is a Windows routine that calls a 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."

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.

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.

In distributed applications, a client is an application that initiates communication with a server application on a remote system.

client area
In Windows, the area of a control which a program (that is, a client of Windows' services) is allowed to draw on. A client area might appear 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.

COM

Component Object Model. COM is Microsoft's client/server object-based model designed to enable interaction between software components and applications. The key aspect of COM is that it enables communication between clients and servers through interfaces. Information about these interfaces is usually included in a type library.

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 and data modules are components that can contain other components (forms and data modules 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 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.

connection component
Any descendant of TCustomRemoteServer. This family of components allow clients in a multitier database application to locate and establish connections to remote data modules on MIDAS servers.

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, TControlBar, and TGroupbox. A container component is the parent of the components it contains.

context menu
A local menu on an object which you access by right-clicking with a pointing device.

control
A visual component (one that appears at runtime). Specifically, any descendant of TControl.

CORBA

Common Object Request Broker Architecture. CORBA is a specification adopted by the Object Management Group (OMG) to address the complexity of developing object applications. It defines an IDL for defining object interfaces, the protocols for remote communication, and a number of standard service interfaces. The CORBA standard has been adopted on multiple platforms.

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 VCL 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 runtime.

data-aware
Able to display and update data stored in an underlying table. All VCL data control components are data-aware.

data control component

A VCL 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 module

A repository for non-visual components. You can place any non-visual components in the data module. At design time, the data module designer lets you view the non-visual components hierarchically or in a list-view format, and to create relationships between components. Use data modules to organize business logic separately from the UI of an application.

data packet

A transportable encoding of database information including metadata, records, and named values that describe other information about the data or its use. Data packets are used to transport database information in a multi-tiered database application.

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 logical view of the data from a database. A dataset is a collection of data determined by a TDataSet descendant such as TClientDataSet, TTable, TQuery, or TStoredProc.

A dataset defined by TTable includes every row in a database able. A dataset defined by a TQuery contains a selection of rows and columns from one or more tables.

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 Object Repository dialog box (Tools | Repository).

default new project

The Project Template that is used to open a new 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 Object Repository dialog box (Tools | Repository).

delta packet

A transportable encoding of a set of changes to the records in a database. Delta packets include insertions, deletions, and modifications. Delta packets to transport information about changes in a multi-tiered database application. See also data packet.

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 class derived from another class. A descendant is type compatible with all of its ancestors.

design time
Phase when you can use the IDE to design your application, using the form, the Object Inspector, Component palette, Code editor, and so forth; as opposed to runtime, when the application you design is actually running.

design time package
A special dynamic-link library used by the IDE to install components and to create special property editors for custom components.

detail table

1. 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.

2. In Oracle 8, a nested dataset. That is, a field that consists of an entire table. By extension, a nested dataset in TClientDataSet component.

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.

Dragging can be part of either a drag-and-drop or a drag-and-dock operation.

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 runtime 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 code 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 runtime), regardless of the intended runtime functionality of the window.

A form is a descendant of TForm.

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 likelihood 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.

globally unique identifier (GUID)
A GUID is a specific type of universally unique identifier (UUID). It is a 16 byte (128-bit) binary value that is guaranteed to be unique. GUIDs are used to identify COM interfaces.

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 runtime). Control through Tools | Environment Options | Preferences.
2. An object on a form that enables you to view and edit information in a spreadsheet-like format. You create a grid with a TDBGrid, TStringGrid, or TDrawGrid 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 and menu items 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 runtime. 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.

HTML

Hypertext Markup Language: A tagged language for creating Web pages. Each HTML document consists of text and embedded tags that modify the attributes or layout of the text or introduce non-text elements such as images or hypertext links.

IDAPI
See BDE.

identifier
A programmer-defined name for a specific item (a constant, type, variable, procedure, function, unit, program, or field).

IDL

Interface definition Language. An interface definition language is a syntax for defining the interfaces of objects or routines that are used in distributed applications. Although the parts of distributed applications may be written in different development languages (such as C++, Java, or Object Pascal), IDL provides a common language that all developers can use to describe the interfaces.

There are three separate dialects of IDL, each of which is specific to a communications protocol:

- Microsoft IDL (MIDL), used to describe COM interfaces.
- CORBA IDL, used to describe CORBA interfaces.
- DCE IDL, used for DCE-based remote procedure calls (such as those supported by Entera).

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 class. More generally, a variable of any type. Actual memory is allocated.

integer
A numeric variable type that is a whole number in the range -2,147,483,648 to +2,147,483,647.

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.

InterBase

Inprise's database server. InterBase has two types of database servers, a local version (local InterBase) and a remote version. Some versions of Delphi include components that access an InterBase server directly, without using the BDE or ADO.

interface

1. The first, public part of a unit that describes the constants, types, variables, procedures, and functions that are available within it.

2. The set of methods supported by a COM object. Applications obtain an instance of an interface by calling QueryInterface, and use this to interact with the COM object.

3. A set of property and method declarations. While Object Pascal classes do not support multiple inheritance, they can implement multiple interfaces to achieve a similar effect.

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 Administrator enables you to specify the default language driver for tables. Language drivers correspond to locales.

late binding

1. When the addess used to call virtual methods or dynamic methods is determined at runtime.

2. When a method call in a distributed application is resolved at runtime, for example by using QueryInterface (COM) or the dynamic invocation interface (CORBA).

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 (Project | Options). 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 (Project | Options | Forms). The main form is usually the first displayed at runtime, and usually the principal form displayed throughout the execution of the program.

marshaling
The mechanism by which remote method and procedure calls are executed in distributed applications. Marshaling transfers arguments from one process space to another and makes a method or function that is implemented in one process space available in another.

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.

MIDAS

Multi-tier Distributed Application Services Suite. MIDAS defines a mechanism by which client applications and application servers communicate database information. Data is encoded in special data packets that are passed between the client and server applications.

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 runtime state of a form designed as a dialog box which the user must close 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 runtime 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. nil pointers can’t be dereferenced: A pointer must be assigned a memory address in order to be meaningfully and safely used.

Note: Dereferencing 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 runtime until it is called (like TSaveDialog) or simply has no appearance at all at runtime (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 its own window.

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.

ORB

Object Request Broker. The runtime software that handles communication in a CORBA application. Client and server applications communicate by passing messages through the ORB. The ORB is not a single executable, but rather a coordinated set of utilities running on different machines.

ordinal (type)
Any Object Pascal type consisting of a closed set of ordered elements.

override
Redefine a virtual object method in a descendant object type.

owner
An object responsible for freeing the resources used by other (owned) objects.

package
A special dynamic-link library used by Delphi applications, the IDE, or both. *Runtime packages* provide functionality when a user runs an application. *Design-time packages* are used to install components in the IDE and to create special property and component editors for custom components.

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. A parent component is responsible for writing its child component to a stream when forms are saved.

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 a 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.

project group
A collection of related projects, such as an executable and its associated DLLs or a client application and its associated server application. The project manager organizes projects into project groups.

property
A feature that provides controlled access to methods or fields of a class. 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 qualifier (a period). 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.

remote data module

A special type of data module that supports an IAppServer interface in multi-tiered database applications. Remote data modules act as COM or CORBA servers that respond to requests from client applications.

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.

runtime
Period when the application you design is running.

runtime error
An error that occurs when the application runs, as opposed to a compile-time error.

runtime library
The standard procedures and functions available to all Object Pascal programs.

runtime only
Routines, properties, events, or components that can be modified, called, or seen only while your application is running (as opposed to design time).

runtime package
A special dynamic-link library used by Delphi applications to provide functionality when a user runs an application.

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. Comments are treated as spaces.

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.

service

1. A utility implemented as an NT service application. NT services are accessed via the Service Control Manager and can be started automatically at system boot, through the Services control panel, or from an application through the service API.

2. The use of an HTTP (Web server) application. The service of an application is usually associated with a specific port number so that client applications can initiate the service. Examples of predefined services include ftp, http, finger, and time.

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.

skeleton
An automatically generated class in a CORBA server application. The skeleton handles marshaling of incoming method calls. CORBA server applications implement objects that correspond to the automatically generated skeleton classes.

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 runtime 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. Also called a context menu.

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 that can be accessed using 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 routines 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.

stub

1. Under CORBA, an automatically generated class in the client application. The stub handles marshaling of outgoing method calls, acting as a proxy for the CORBA object on the server application.

2. A routine or method that has not been fully implemented. Stubs serve as placeholders that can be called by application code while it is under development.

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

1. 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 runtime. Normal typecasts are evaluate at compile time and are not validated at runtime.

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.

type library
Files that include information about data types, interfaces, member functions, and object classes exposed by an ActiveX control or server. Delphi lets you view and edit type libraries using the type library editor.

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 the VCL, 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 runtime.

virtual

Use the **virtual** keyword to allow derived classes to provide different versions of a base class function. Once you declare a function as **virtual**, you can redefine it in any derived class, even if the number and type of arguments are the same. The redefined function overrides the base class function.

visual component
A component that is visible, or can be made visible on a form at runtime.

warning

A message that appears in the Message window that does not stop your code from compiling, but indicates areas you might want to examine for problems.

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 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.

Web item
A component that generates HTML for part of an HTML document produced in a MIDAS Web application. Web items support the IWebComponent and IWebContent interfaces, among others.

Web module
A special type of data module that dispatches HTTP request messages to the objects that handle them.

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 contain other controls, and which has its own window.

wizard
A dialog or set of dialogs that obtain information about an object you want to create and then generate code to implement that object.

word
A location in memory occupying 2 adjacent bytes; the storage required for a variable of type shortint 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.

IDE command-line options

This topic lists and describes all of the options that you can use to start the IDE from the command line.

You must precede all options (unless otherwise noted) with either a dash (-) or a slash (/). The options are not case sensitive. Therefore, the following options are all identical: -d /d -D /D.

You use these options with the IDE startup command: `delphi32.exe`.

For example:

```
delphi32.exe /ns /hm
```

Starts the IDE with no splash screen and tracks memory allocation.

```
delphi32.exe–sdc:\test\source –d c:\test\myprog.exe -td
```

Starts the IDE and loads c:\test\myprog.exe into the debugger and used c:\test\source as the location for the source code while debugging. The –td and any other argument that appears after the –d*exename* debugger option is used as an argument to c:\test\myprog.exe.

General options

Option	Description
?	Displays help for IDE command-line options.
hm	Heap Monitor. Displays information in the IDE title bar regarding the amount of memory allocated using the memory manager. Displays the number of blocks and bytes allocated. Information gets updated when the IDE is idle.
hv	Heap Verify. Performs validation of memory allocated using the memory manager. Displays error information in the IDE title bar if errors are found in the heap.
ns	No splash screen. Suppresses display of the splash screen during IDE startup.
np	No Project. Supresses loading of any desktop files on IDE startup and suppresses creation of a default project.

Debugger options

Option	Description
d*exename*	Loads the specified executable into the debugger. Any parameters specified after the *exename* are used as parameters to the program being debugged and are ignored by the IDE. A space is allowed between the d and the *exename*.
attach:%1;%2	Performs a debug attach, using %1 as the process ID to attach to and %2 as the event ID for that process. It can be used manually, but is used mostly for Just in Time debugging.
td	TDGoodies. Implements several features found in the TurboDebugger, TD32. It must be used with the d option. It causes the CPU and FPU views to stay open when a process terminates. It causes Run|Program Reset to terminate the current process and reload it in the debugger. If there is no current process, Run|Program Reset reloads the last process that terminated. It also causes breakpoints and watches to be saved in the default desktop if desktop saving is on and no project is loaded.
sd*directories*	Source Directories. Must be used with the d option. The argument is either a single directory or a semicolon delimited list of directories which are used as the Debug Source Path setting (can also be set using the Project|Options|Directories/Conditionals option page). No space is allowed between sd and the directory list argument.
h*hostname*	Hostname. Must be used with the d option. When specified, a remote debug session is initiated using the specified host name as the remote host to debug on. The remote debug server program must be running on the remote host.

Project options

Option	Description
filename	(No preceding dash) The specified *filename* is loaded in the IDE. It can be a project, project group, or a single file.
b	AutoBuild. Must be used with the *filename* option. When specified, the project or project group is built automatically when the IDE starts. Any hints, errors, or warnings are then saved to a file. Then the IDE exits. This facilitates doing builds in batch mode from a batch file. The Error Level is set to 0 for successful builds and 1 for failed builds. By default, the output file has the same name as the *filename* specified with the file extension changed to .err. This can be overridden using the o option.
m	AutoMake. Same as AutoBuild, but a make is performed rather than a full build.
o*outputfile*	Output file. Must be used the b or m option. When specified, any hints, warnings, or errors are written to the file specified instead of the default file.

Dockable tool windows

Docking allows you to make full and efficient use of your screen space as you work on your project. From the View menu, you can bring up any tool window and then dock it directly onto the Code editor for use while coding and debugging. You can also dock two or more tool windows together to form tabbed tool windows to save screen space while retaining fast one-step access to these tools.

Changing the docking state

Most tool windows in the IDE are dockable. You can recognize dockable windows because they have

- A thinner title bar than the code editor.
- A Dockable property on the context menu. Uncheck this property to turn off the drag-and-dock capability of a tool window.
- A drag outline that appears when you move the tool window with the mouse.

Saving the docking state

The docking state is saved with a desktop configuration. To save this configuration, choose View| Desktops|Save Desktop (or click the Save current desktop icon on the Desktops toolbar).

Preventing a window from docking

When you see the drag rectangle snap to possible dock sites, you can prevent this imminent dock from happening by holding the Ctrl key down and keep dragging the window.

Docking tool windows onto the Code editor

See also

If you want to dock one or more tool windows onto the Code editor:

1. Choose the View menu and the name of the tool you want to dock. For example, if you want to dock the breakpoints tool choose View|Breakpoints.
2. When the tool window appears onscreen, drag it by clicking on the title bar. When the tool window is over a docking site, its drag outline narrows and changes shape to show how the window would dock.
3. Release the mouse to dock the tool window.

   For example, to dock the Breakpoint List window onto the Code editor, click on the Breakpoint List title bar and drag the rectangle around the Code editor window until the drag outline narrows.

To undock a tool window from the Code editor:

1. Drag the tool window away from the Code editor by clicking on the title bar. When the tool window is no longer over a docking site, its drag outline widens.
2. Release the mouse. The tool window becomes a floating window.

   For example, to undock the Breakpoint List window from the Code editor, drag the Breakpoint List window away from the Code editor using its title bar. The narrow drag outline widens when you are no longer over a docking.

Docking tool windows together to form tabbed tool windows

See also

If you want to dock two or more tool windows together:

1. Choose the View menu and the names of the tools you want to dock together. For example, to dock together the Call Stack tool and the Watches tool, choose View|Debug Windows|Call Stack then View|Debug Windows|Watches.
2. When the tool windows appear onscreen, drag one tool window by clicking on its title bar and move it onto the other tool window.
3. When the drag outline narrows, release the mouse. The two windows dock together.

   For example, to dock the Call Stack and Watch List windows together, drag the Watch List over the Call Stack window and drop it when the outline narrows.
4. To view the hidden windows, click the tab with its name on it. For example, to view the Call Stack window, click the Call Stack.
5. When dragging windows to a dock location, the drag rectangle snaps to possible dock sites. To prevent this imminent dock from happening hold the Ctrl key down and keep dragging the window.

To undock a tool window from the tabbed tools window:

1. Choose the window you want to undock by clicking its tab. For example, to undock the Watch list window, click the Watch List tab.
2. Drag the tool window away from tools window until the drag outline widens and then release the mouse. The tool window becomes a floating window.

   For example, to undock the Watch List window from the tools window, drag the Watch List window away from the tools window. The narrow drag outline widens when you are no longer over a tools window docking site.

System (Brief)

See also

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|Run
F10	Accesses the menu bar
F11	View|Object Inspector
F12	View|Toggle Form/Unit
Alt+F2	Zooms window
Alt+F7	Displays previous error in Message view
Alt+F8	Displays next error in Message view
Alt+F9	Displays a context menu
Alt+F10	Project|Compile
Alt+F11	File|Use Unit
Alt+F12	View|Toggle Form/Unit
Ctrl+F1	Topic search
Ctrl+F2	Run|Program Reset
Ctrl+F3	View|Call Stack
Ctrl+F7	Evaluate/modify
Ctrl+F8	Toggle breakpoint
Ctrl+F9	Project|Compile
Ctrl+F11	Run|Step over
Ctrl+D	Descends item (replaces Inspector window)
Ctrl+N	Opens a new Inspector window
Ctrl+S	Search| Incremental Search
Ctrl+T	Displays the Type Cast dialog
Shift+F3	View|Call Stack
Shift+F7	Run|Trace To Next Source Line
Shift+F8	Run|Trace Into
Shift+F10	Project|Add To Project
Shift+F11	View|CPU
Ctrl+Hyphen	File|Close
Ctrl+F12	View|Units

Shift+ F12	View|Forms
Alt+B	View|Window list
Alt+E	File|Open(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 As(Note: opens Save As dialog box, even when Code editor window does not have focus)
Alt+-	Displays the previous page
Alt+W	File|Save
Alt+X	File|Exit
Alt+Z	Accesses the File menu

Clipboard control (Brief)

See also

These Clipboard keyboard shortcuts apply to the Brief keystroke mapping scheme.

Shortcut	Command
Ins	Edit|Paste
Plus (+)	Edit|Copy
Minus (-)	Edit|Cut

Editor (Brief)

See also

These editor keyboard shortcuts apply to the Brief keystroke mapping scheme.

Shortcut	Action or command
F5	Search|Find (forward from cursor position)
F6	Search|Replace (forward from cursor position)
Alt+F5	Search|Find (backward from cursor position)
Alt+F6	Search|Replace (backward from cursor position)
Alt+F9	Displays the local menu
Shift+F4	Tiles windows horizontally
Shift+F5	Search|Search Again
Shift+F6	Repeats the last Search|Replace operation
Esc	Cancels a command at the prompt
Del	Deletes a character or block at the cursor
*****	Edit|Undo
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|Redo
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 number
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|Find
Alt+T	Search|Replace
Alt+U	Edit|Undo
Alt+Backspace	Deletes the word to the right of the cursor
Alt+Hyphen	Jumps to the previous page
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+O	Toggles the case of a selection
Ctrl+F1	Help keyword search
Ctrl+F5	Toggles case-sensitive searching
Ctrl+F6	Toggles regular expression searching

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

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

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

System (classic)

See also

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	Zooms window
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+F2	View|CPU
Alt+F3	File|Close
Alt+F7	Displays previous error in Message view
Alt+F8	Displays next error in Message view
Alt+F10	Displays a context menu
Alt+F11	File|Use Unit
Alt+F12	Displays the Code editor
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 project
Ctrl+F11	File|Open Project
Ctrl+F12	View|Units
Shift+F7	Run|Trace To Next Source Line
Shift+F11	Project|Add To Project
Shift+F12	View|Forms

Ctrl+D	Descends item (replaces Inspector window)
Ctrl+N	Opens a new Inspector window
Ctrl+S	Incremental search
Ctrl+T	Displays the Type Cast dialog
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)

See also

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
Plus (+)	Edit|Copy
Minus (-)	Edit|Cut
Start (*)	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+Right 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

System (default)

See also

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+0	View|Window List
Alt+F2	View|Debug Windows|CPU
Alt+F7	Displays previous error in Message view
Alt+F8	Displays next error in Message view
Alt+F10	Displays a context menu
Alt+F11	File|Use Unit
Alt+F12	Displays the Code editor
Ctrl+F1	Help|Topic Search
Ctrl+F2	Run|Program Reset
Ctrl+F3	View|Debug Windows|Call Stack
Ctrl+F4	Closes current file
Ctrl+F5	Add Watch at Cursor
Ctrl+F6	Displays header file in Code editor
Ctrl+F7	Evaluate/Modify
Ctrl+F9	Project|Compile project
Ctrl+F11	File|Open Project
Ctrl+F12	View|Units
Ctrl+D	Descends item (replaces Inspector window)
Ctrl+E	View|Code Explorer
Ctrl+N	Opens a new Inspector window
Ctrl+S	Incremental search
Ctrl+T	Displays the Type Cast dialog
Shift+F7	Run|Trace To Next Source Line
Shift+F11	Project|Add To Project

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)

See also

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+j	Templates pop-up menu
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+<space bar>	Code Completion pop-up window
Ctrl+Shift+g	Inserts a new Globally Unique Identifier (GUID)
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
Ctrl+Shift+<space bar>	Code Parameters pop-up window
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 code page (or file)
Ctrl+Shift+Tab	Moves to the previous code page (or file)
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+Right 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

System (Epsilon)

See also

These system keyboard shortcuts apply to the Epsilon keystroke mapping scheme.

Shortcut	Action or command
F1	Displays context-sensitive Help
F5	Toggle Breakpoint
F7	Run|Trace Into
F8	Run|Step Over
F9	Run|Run
F10	Edit|Redo
F11	View|Object Inspector
F12	View|Toggle Form/Unit
Alt+0	View|Window List
Alt+F3	View|CPU
Alt+F7	Displays previous error in Message view
Alt+F8	Displays next error in Message view
Alt+F9	Project|Compile project
Alt+F10	Displays a context menu
Alt+F11	File|Use Unit
Alt+F12	Displays the Code editor
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 project
Ctrl+F12	View|Units
Ctrl+D	Descends item (replaces Inspector window)
Ctrl+N	Opens a new Inspector window
Ctrl+S	Incremental search
Ctrl+T	Displays the Type Cast dialog
Shift+F3	View|Call Stack
Shift+F7	Run|Trace To Next Source Line
Shift+F11	Project|Add To Project
Shift+F12	View|Forms

Ctrl+X+(	Records a keyboard macro
Ctrl+X+)	Ends a keyboard macro recording
Ctrl+X+e	Plays back the last keyboard macro recorded
Ctrl+X+E	Plays back 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)

See also

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 first letter of the current word
Esc+@c	Capitalizes the first letter of 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 project
Ctrl+X+M	Project|Compile project
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	Incrementally 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

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

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, Ctrl+X	Toggles between bookmark and current position

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+RightArrow	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
UpArrow	Moves the cursor up a line
Ctrl+N	Moves the cursor down a line
DownArrow	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

System (Visual Studio)

See also

These system keyboard shortcuts apply to the Visual Studio keystroke mapping scheme.

Shortcut	Action or command
Ctrl+R	Records a keyboard macro
Ctrl+P	Plays back a keyboard macro
F1	Displays context-sensitive Help
F4	Run|Go to Cursor
F5	Run|Run
F7	Project|Build project
F9	Toggle breakpoint
F10	Run|Step over
F11	Run|Trace Into
F12	View|Toggle Form/Unit
Alt+0	View|Window list
Alt+3	View|Debug Windows|Watches
Alt+4	View|Debug Windows|Local Variables
Alt+7	View|Debug Windows|Call Stack
Alt+8	View|Debug Windows|CPU
Alt+F2	View|Debug Windows|CPU
Alt+F5	Run|Inspect
Alt+F7	Project|Options
Alt+F10	Displays a context menu
Alt+F11	File|Use Unit
Alt+F12	View|Toggle Form/Unit
Alt+Enter	View|Object Inspector
Shift+F5	Run|Program Reset
Shift+F7	Run|Trace To Next Source Line
Shift+F9	Run|Add Watch
Shift+ F11	Project|Add To Project
Shift+ F12	View|Forms
Ctrl+F1	Topic search
Ctrl+F2	Evaluate/modify
Ctrl+F3	View|Debug Windows|Call Stack
Ctrl+F7	Project|Compile

Ctrl+F10	Run|Go to Cursor
Ctrl+F12	View|Units
Ctrl+a	Edit|Select All
Ctrl+B	View|Breakpoints
Ctrl+E	View|Code Explorer
Ctrl+n	File|New
Ctrl+N	File|New Application
Ctrl+o	File|Open
Ctrl+O	File|Open Project
Ctrl+s	File|Save
Ctrl+S	File|Save All
Ctrl+<space bar>	Code Completion pop-up window
Ctrl+Shift+<space bar>	Code Completion pop-up window
Ctrl+Tab	Displays the next page
Ctrl+Shift+Tab	Displays the previous page
Ctrl+Q+W	Displays next error in Message view
Ctrl+Alt+E	View|Debug Windows|Event Log
Ctrl+Alt+M	View|Debug Windows|Modules

Clipboard control (Visual Studio)

See also

These Clipboard keyboard shortcuts apply to the Visual Studio 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 (Visual Studio)

See also

These editor keyboard shortcuts apply to the Visual Studio keystroke mapping scheme.

Shortcut	Action or command
F3	Search|Search Again
Ctrl+F	Search|Find
Ctrl+g	Search|Go to line number
Ctrl+G	Open file at cursor
Ctrl+h	Search|Replace
Ctrl+I	Search|Incremental Search
Ctrl+j	Templates pop-up menu
Ctrl+L	Deletes a line
Ctrl+P	Causes next character to be interpreted as an ASCII sequence
Ctrl+s	File|Save
Ctrl+T	Deletes the word to the left of the cursor
Ctrl+y	Deletes a line
Ctrl+Y	Deletes to the end of a line
Ctrl+z	Edit|Undo
Ctrl+Z	Edit|Redo
Ctrl+Tab	Displays the next window in the buffer list
Ctrl+Shift+Tab	Displays the previous window in the buffer list
Ctrl+F4	Closes the current page
Ctrl+K+E	Converts the word under the cursor to lower case
Ctrl+K+F	Converts the word under the cursor to upper case
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)
Alt+F3	Search|Find
Alt+F12	Browse symbol at cursor
Alt+[	Finds the matching delimiter (forward)
Alt+]	Finds the matching delimiter (backward)

Delete	Deletes a character or block at the cursor
Backspace	Deletes the character to the left of the cursor
Shift+Backspace	Deletes the character to the left of the cursor
Ctrl+Backspace	Deletes the word to the left of the cursor
Tab	Inserts a tab character
Enter	Inserts a new line character
Insert	Toggles insert mode

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 character

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

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+Right 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

Block commands (Visual Studio)

See also

These block command keyboard shortcuts apply to the Visual Studio keystroke mapping scheme.

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+P	Prints selected block
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+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+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+Del	Deletes a selected block
Ctrl+Q+B	Moves to the beginning of a block
Ctrl+Q+K	Moves to the end of a block

Cursor movement (Visual Studio)

See also

These cursor movement shortcuts apply to the Visual Studio keystroke mappings 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
Shift+Tab	Moves the cursor to the left one tab position
Ctrl+Left Arrow	Moves one word to the left
Ctrl+Right Arrow	Moves one word to the right
Ctrl+PgDn	Moves to the bottom of the screen
Ctrl+PgUp	Moves to the top of the screen
Ctrl+UpArrow	Scrolls the screen up one line.
Ctrl+DownArrow	Scrolls the screen down one line.
Ctrl+Home	Moves to the top of a file
Ctrl+End	Moves to the end of a file
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

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 large part of the Epsilon editor
Visual Studio	Key bindings that emulate a large part of the Visual Studio editor

To select a keymapping:

1. Choose 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. Choose 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.

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:

Classic keystroke mapping

The Classic keystroke mapping scheme provides key bindings that match the Delphi programming environment. 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:

Visual Studio keystroke mapping

The Visual Studio keystroke mapping scheme provides key bindings that emulate most of the Visual Studio editor. 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

System

Debugger (default, classic, Brief, Epsilon, and Visual Studio)

See also

The Debugger keyboard shortcuts apply to all keystroke mapping schemes:

Default

Classic

Brief

Epsilon

Visual Studio

Breakpoint view

Ctrl+V	View Source
Ctrl+S	Edit Source
Ctrl+E	Edit Breakpoint
Enter	Edit Breakpoint
Ctrl+D	Delete Breakpoint
Del	Delete Breakpoint
Ctrl+A	Add Breakpoint
Ins	Add Breakpoint
Ctrl+N	Enable Breakpoint

Call stack view

Ctrl+V	View Source
Ctrl+E	Edit Source
Space	View Source (Epsilon only)
Ctrl+Enter	Edit Source (Epsilon only)

Message view

Ctrl+V	View Source
Space	View Source
Ctrl+S	Edit Source
Ctrl+Enter	Edit Source

Watch view

Ctrl+E	Edit Watch
Enter	Edit Watch
Ctrl+A	Add Watch
Ins	Add Watch
Ctrl+D	Delete Watch
Del	Delete Watch

Block commands (default and classic)

See also

These block command shortcuts apply to the Default and 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+P	Prints selected block
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

Bookmark operations (default, classic, and Visual Studio)

See also

The following bookmark operations shortcuts apply to the Default, Classic, and Visual Studio keystroke mappings 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 and Visual Studio schemes:

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

Cursor movement (default and classic)

See also

These cursor movement shortcuts apply to the Default and Classic keystroke mappings 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

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 (default and classic only)
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

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

Keyboard shortcuts for the File menu

See also

The table below lists keyboard shortcuts for file commands.

Command	Shortcut	Mapping
File|New	**Ctrl+n**	Visual Studio
File|New Application	**Ctrl+N**	Visual Studio
File|Open	**F3**	Classic
	Alt+E	Brief
	Ctrl+X+Ctrl+F	Epsilon
	Ctrl+o	Visual Studio
File|Open Project	**Ctrl+O**	Visual Studio
Open File At Cursor	**Ctrl+O+A**	Default, Classic, Brief
	Ctrl+G	Visual Studio
File|Save	**Ctrl+K+S**	Default, Classic
	Ctrl+S	Default
	Ctrl+s	Visual Studio
	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|Save All	**Ctrl+S**	Visual Studio
File|Close	**Alt+F3**	Classic
	Ctrl+Hyphen	Brief
Close Active Window	**Alt+F4**	Default, Classic, Brief, Epsilon
File|Close	**Alt+F3**	Classic
	Ctrl+Hyphen	Brief
File|Use Unit	**Alt+F11**	Default, Classic, Epsilon, Brief, Visual Studio
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

Visual Studio keystroke mapping

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, Visual Studio
	Ctrl+X	Default, Visual Studio
	Minus (-)	Brief
Edit|Copy	**Ctrl+Ins**	Default, Classic, Visual Studio
	Ctrl+C	Default, Visual Studio
	Plus (+)	Brief
	Alt+w	Epsilon
	Esc+@w	Epsilon
	Ctrl+Alt+w	Epsilon
	Esc+Ctrl+w	Epsilon
Edit|Paste	**Shift+Ins**	Default, Classic, Visual Studio
	Ctrl+V	Default, Visual Studio
	Ins	Brief
	Ctrl+Y	Epsilon
Edit|Delete	**Ctrl+Del**	Default, Classic, Visual Studio
Edit|Redo	**Ctrl+Shift+Z**	Default, Visual Studio
	Alt+Shift+Backspace	Default, Classic, Visual Studio
	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, Visual Studio
	Ctrl+Z	Visual Studio
	Star	Brief
	Alt+U	Brief
	Ctrl+X+u	Epsilon
	Ctrl+X+U	Epsilon
	F9	Epsilon
	Ctrl+F9	Epsilon
	Ctrl+X+Ctrl+U	Epsilon
Edit|Select All	**Ctrl+A**	Visual Studio

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
Visual Studio keystroke mapping

Search commands keyboard shortcuts

See also

The table below lists the keyboard shortcuts for commands on the Search menu.

Command	Shortcut	Mapping
Find	**Ctrl+Q+F**	Default, Classic
	Ctrl+F	Default, Visual Studio
	F5	Brief
	Alt+F5	Brief
	Alt+S	Brief
	Ctrl+Alt+S	Epsilon
	Esc+Ctrl+S	Epsilon
	Ctrl+Alt+R	Epsilon
	Esc+Ctrl+R	Epsilon
	Alt+F3	Visual Studio
Replace	**Ctrl+Q+A**	Default, Classic
	Ctrl+R	Default
	Ctrl+H	Visual Studio
	Alt+T	Brief
	F6	Brief
	Alt+F6	Brief
	Alt+&	Epsilon
	Esc+&	Epsilon
	Alt+%	Epsilon
	Esc+%	Epsilon
	Alt+*	Epsilon
	Esc+*	Epsilon
Search Again	**F3**	Default, Visual Studio
	Ctrl+L	Classic
	Shift+F5	Brief
Incremental Search	**Ctrl+I**	Visual Studio
Go To Line Number	**Ctrl+O+G**	Default, Classic, Brief
	Alt+G	Brief
	Ctrl+X+g	Epsilon
	Ctrl+X+G	Epsilon
	Ctrl+g	Visual Studio

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

Visual Studio keystroke mapping

Debug commands keyboard shortcuts

See also

The table below lists the keyboard shortcuts for debug operations.

Command	Shortcut	Mapping
Run|Run	**F9**	Default, Classic, Brief, Epsilon
	F5	Visual Studio
Run|Go to Cursor	**F4**	Default, Classic, Visual Studio
	Ctrl+F10	Visual Studio
	Alt+F7	Brief
Run|Add Breakpoint	*F5*	Default
Run|Trace Into	**F7**	Default, Classic
	F11	Visual Studio
Run|Step Over	**F8**	Default, Classic, Epsilon
	F10	Visual Studio
Run|Program Reset	**Ctrl+F2**	Default, Classic, Brief, Epsilon
	Shift+F5	Visual Studio
Run|Add Watch	**Ctrl+F5**	Epsilon
	Shift+F9	Visual Studio
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
	Ctrl+F2	Visual Studio
Toggle Breakpoint	**Ctrl+F8**	Classic, Brief
	F5	Epsilon
	F9	Visual Studio
Inspect	**Alt+F5**	Visual Studio

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
Visual Studio keystroke mapping

Build commands keyboard shortcuts

See also

This table lists the keyboard shortcuts for build operations:

Command	Shortcut	Mapping
Project|Compile project	**Ctrl+F9**	Default, Classic, Brief
	Alt+F9	Default, Classic, Epsilon
	Alt+F10	Brief
	Ctrl+X+m	Epsilon
	Ctrl+X+M	Epsilon
	Ctrl+F7	Visual Studio
Project|Build project	**F7**	Visual Studio

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

Visual Studio keystroke mapping

Keyboard support in the IDE

See also

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, 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

Package editor keyboard shortcuts

CPU window keyboard shortcuts

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 the next component
Shift+Tab	Selects the previous component
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.

Project Manager keyboard shortcuts

See also

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

See also

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 event 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.

Package editor keyboard shortcuts

See also

Listed below are keyboard shortcuts for working in the package editor.

Keyboard command	Description
Enter	Lets you view the selected unit's source code.
Ins	Adds a unit to the current folder (Contains or Requires).
Del	Removes the selected item from the package.
Ctrl+B	Compiles the current package. If changes to the package are required, a dialog box appears that lists the changes that will be made to the package before it is compiled.
Ctrl+I	Installs the current package as a design time package. If changes to the package are required, a dialog box appears that lists the changes that will be made to the package before it is compiled.

CPU Window keyboard shortcuts

See also

Listed below are keyboard shortcuts for navigating in the CPU window.

Keyboard command	Description
Shift+Left Arrow	Move left one pane.
Shift+Right Arrow	Move right one pane
Shift+Up Arrow	Move up one pane.
Shift+Down Arrow	Move down one pane.

Object Inspector context menu

See also

The Object Inspector context menu provides you with commands for closing the Object Inspector, displaying Help, and for keeping the Object Inspector the top-most window.

The commands on the Object Inspector context menu are

- View
- Arrange
- Revert To Inherited
- Expand
- Collapse
- Stay On Top
- Status Bar
- Hide
- Help
- Dockable

View (Object Inspector context menu)

See also

Right-click in the Object Inspector and choose VIew|(category) to filter the display of properties or events. The categories of properties or events that are currently displayed are listed with toggle check marks. Make sure there are checkmarks by all categories you want to display.

Categories you see depend on which object is selected and whether you are in the Events or Properties tab. Common categories are Drag Drop Docking, Help and Hints, and Visual. Properties are listed by category or alphabetically; see Arrange.

Note: Some properties or events logically occur in multiple categories.

The following commands appear on the bottom of the View submenu:

Value	Description
View|All	Display all properties (if in the Properties tab) or events (if in the Events tab).
View|Toggle	Display all properties currently unchecked; hide all checked properties.
View|None	Display no properties or events.

Arrange (Object Inspector context menu)

See also

Right-click in the Object Inspector and choose Arrange to change the ordering of listed properties or events. You can arrange:

Value	Description
by Category	Displays properties or events by category. The categories are listed alphabetically. You can collapse or expand the categories by clicking the **+** or **–** collapse icon and the state is persistent until you change it.
by Name	Displays visible properties or events alphabetically. The categories are no longer visible in the Object Inspector.

You use View|(category) to specify which categories of properties or events are displayed.

Revert To Inherited **(Object Inspector context menu)**

Right-click in the Object Inspector and choose Revert to Inherited when you want to change an object that has had its properties overwritten back to the original inherited behavior.

This option is only available when the object has properties.

For example, if a form inherits a certain button placement from another form and you then move the button, Revert To Inherited returns the button to its original position.

Expand **(Object Inspector context menu)**

Right-click in the Object Inspector and choose Expand 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.

Collapse **(Object Inspector context menu)**

Right-click in the Object Inspector and choose Collapse 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.

Stay On Top **(Object Inspector context menu)**

Right-click in the Object Inspector and choose Stay On Top to keep the Object Inspector in front of all other Delphi windows and dialog boxes.

Status Bar **(Object Inspector context menu)**

Right-click in the Object Inspector and choose Status Bar to show or hide the status bar at the bottom of the Object Inspector. The status bar states how many properties or events are not shown as a result of using the View command. If all properties or events are visible in the Object Inspector, it says “All shown.”

Hide **(Object Inspector context menu)**

Right-click in the Object Inspector and choose Hide to close the display.

When closed, an item can be redisplayed from the View menu.

Hide is available from the following context menus:

- Alignment Palette context menu
- Component palette context menu
- Object Inspector context menu
- Toolbar context menu

Help **(Object Inspector or Component Palette context menu)**

Right-click in the Object Inspector and choose Help to display Help for the item.

Help is available from the following context menus :

- Alignment Palette context menu
- Component palette context menu
- Object Inspector context menu
- Toolbar context menu

Dockable

Check Dockable to allow a tool window to be docked (connected) to other windows, such as the code editor.

UnCheck Dockable to prevent the tool window from being docked to other windows. If the tool window is currently docked, unchecking Dockable will cause the tool window to undock and become a floating window.

To-Do List context menu

See also

The To-Do List context menu provides you with commands for adding, deleting, and editing items on a to-do list. It also lets you change the format of the to-do list. Right-click on a to-do list or select an item and right-click to display the To-Do List context menu.

The commands that may appear on the To-Do List context menu are

- Add
- Delete
- Edit
- Sort
- Filter
- Show Completed Items
- Show ToolTips When Clipped
- Copy As
- Table Properties
- Dockable

Add **(To-Do List context menu)**

See also

Right-click on the to-do list and choose Add to add an item to the project's to-do list. Type the item text and optional Priority, Owner, and Category, then click OK.

Delete **(To-Do List context menu)**

See also

Select an item in the to-do list and press Delete. You can also right-click on the selected to-do list item and choose Delete.

If an item's text is grayed out in the to-do list, it comes from a source file in the project that is not currently open. It can't be edited or deleted until it is open in the editor. Double-click the item to open the source file containing the item in the editor.

Sort (To-Do List context menu)

See also

Right-click on the to-do list and choose Sort to change the order in which items are listed in the to-do list. Select one of the sort options to determine how to sort the list. The current sort option becomes bulleted.

Sort provides the following sort orders:

Sort option	Description
Action Item	Sort alphabetically by action item.
Status	Sort alphabetically with incomplete items first, then completed items.
Type	Sort alphabetically with global items first, then show to-do items within specific modules.
Priority	Sort highest priority items first (with 1 being the highest priority, 2 the second highest, and so on); items with no set priority are included last.
Module	Sort items alphabetically according to which module they're in.
Owner	Sort items alphabetically by owner.
Category	Sort items alphabetically by category.

You can also click on the column headings in the to-do list to sort the fields. Clicking on Action item sorts alphabetically, then by status, and then by type. Clicking on the other column headings sorts the list by that column in ascending or descending order.

Filter (To-Do List context menu)

See also

Right-click on the to-do list and choose Filter to choose which to-do list items are displayed. You can filter the list by categories, owners, or item types.

Filter option	Description
Categories	Display items in checked categories.
Owners	Display items belonging to owners checked.
Item types	Lets you filter items by origin using the Filter To-Do List dialog.

Filter To-Do List dialog box

See also

The Filter To-Do List dialog box is displayed when you right-click on a to-do list and choose Filter| Categories, Owners, or Item Types. Three slightly different dialogs appear depending on which option you selected.

You use the dialog to filter which items are displayed in a to-do list:

Filtering by Categories

Any categories that you have used within the to-do list are shown in a list. You can check categories of items you want to display and uncheck them to exclude them from being displayed in the to-do list.

Filtering by Owners

All owner names that you have used within the to-do list are shown in a list. You can check owners whose to-do list items you want to display and uncheck them to exclude them from being displayed in the to-do list.

Filtering by Item types

Three options let you filter items by origin:

Current project source files	Displays to-do list items that were added directly in the current project's source files.
Open source files	Displays to-do list items that were added directly in any source files you have open.
Project To-Do file	Displays to-do list items that were added directly to the to-do list and which apply to the whole project.

Copy As (To-Do List context menu)

See also

Copies the contents of the to-do list in one of the following formats:

Copy As option	Description
Text	copies the current contents of the to-do list (as it appears in the window, including the header titles) to the clipboard in a tab-delimited format. Columns are separated by tabs and each row is on a separate line.
HTML Table	copies the current contents of the to-do list to an HTML table, which can be pasted into an HTML document. You format the table by right-clicking and choosing Table Properties to display the Table Properties dialog box.

Table Properties **(To-Do List context menu)**

This command displays the Table Properties dialog box. The dialog allows you to specify basic HTML formatting options for the to-do list when using the Copy As|HTML Table option.

Table Properties dialog box

See also

The Table Properties dialog box is displayed when you right-click on a to-do list and choose Table Properties. You use the dialog to set the basic HTML formatting options for the table, including table and column properties if you plan to use the Copy As|HTML Table option. The options are stored in the registry.

Table options

Lets you specify properties that apply to the whole table.

Option	Description
Caption	Lets you add a caption for the table.
Border Width	Specifies the width (in pixels only) of the frame around a table.
Width (Percent)	Specifies a value for how wide the table will appear on the page. The value is relative to the amount of available horizontal space.
Cell Spacing	Specifies how much space to leave between the left side of the table and the left side of the leftmost column, the top of the table and the top-side attribute. Also specifies the amount of space to leave between cells.
Cell Padding	Specifies the amount of space between the border of the cell and its contents.
Background Color	Lets you explicitly code a background color for the HTML table cells.
Alignment	Indicates the location (left, right, or center) of the table on the HTML page.

Column options

Lets you specify properties for each of the columns in the to-do list.

Option	Description
Column	Lets you choose the column for which you want to specify properties.
Alignment	Specifies the alignment of the text within the column.
Vertical alignment	Specifies the alignment of the text within the cell.
Title	Indicates the column heading.
Width	Specifies the width of the column in a percentage of the whole table width.
Height	Specifies a recommended cell height in pixels.
Wrap text	Allows text within cells to wrap.
Visible	Determines whether or not this column will be included in the table or not.
Font Size	Specifies the point size of the text in the column.
Face	Lets you change the typeface of the text in this column.
Color	Lets you change the color of this column.
Bold	Makes the text in this column bold.
Italic	Makes the text in the column italic.

Edit **(To-Do List context menu)**

See also

Select an item in the to-do list, right-click, and choose Edit to modify an item in the to-do list. An Edit To-Do Item dialog box is displayed where you can edit the item.

If an item's text is grayed out in the to-do list, it comes from a source file in the project that is not currently open. It can't be edited or deleted until it is open in the editor. Double-click the item to open the source file containing the item in the editor.

Show ToolTips When Clipped **(To-Do List context menu)**

See also

Check Show ToolTips When Clipped if you want to be able to point to a cut-off field in the to-do list and see a tooltip that shows the entire contents of the field. Although you can resize the fields to see more information, some fields, such as the module pathname, can be quite long. Displaying the tootips can be helpful in this case.

Show Completed Items **(To-Do List context menu)**

See also

Check Show Completed Items if you want to include completed to-do list items in the to-do list. The completed items are shown in strike-through font and the status box is checked.

Dockable

See also

Check Dockable to allow a to-do list to be docked (connected) to other windows, such as the code editor.

Uncheck Dockable to prevent the to-do list from being docked to other windows. If the to-do list is currently docked, unchecking Dockable will cause the to-do list to undock and become a floating window.

Edit To-Do Item dialog box

See also

The Edit To-Do Item dialog box lets you edit an item on the to-do list. Select the item you want to edit, right-click, and choose Edit. Change the fields you want and click OK.

The dialog box includes the following fields:

Column	Description
Text	Specifies the to-do list item text. Enter the text here.
Priority	Specifies the importance of the item using a decimal number from 1 to 5. You can type the number or select one using the spin control.
Owner	Says who's responsible for completing the task. You can type the name or select one if others are listed in the spin control.
Category	Indicates a type of task (for example, user interface or UI, or Interface implementation).You can type the category or select one if others are listed in the spin control.

The Edit To-Do Item dialog box also includes a Done check box that specifies whether or not the item has been completed. The status is indicated in the to-do list itself by a box with or without a checkmark. A check means it is done. Done items are shown as crossed out. If Show Completed Items is unchecked, completed items will not appear in the list.

Add To-Do Item dialog box

See also

The Add To-Do Item dialog box lets you add a task to the to-do list directly. Right-click on the to-do list and choose Add. Fill in the fields listed below and click OK to add an item to the list.

The dialog box includes the following fields:

Column	Description
Text	Specifies the to-do list item text. Enter the text here.
Priority	Specifies the importance of the item using a decimal number from 1 to 5. You can type the number or select one using the spin control.
Owner	Says who's responsible for completing the task. You can type the name or select one if others are listed in the spin control.
Category	Indicates a type of task (for example, user interface or UI, or Interface implementation).You can type the category or select one if others are listed in the spin control.

Working with projects

See also

When you're working in Delphi, you're working on a project. These topics describe the files that make up a project and then provide information on working with projects. The topics covered here include

- What is a project?
- Viewing a project's contents
- Saving projects and individual project files
- Managing projects
- Sharing objects
- Creating a project group
- Specifying a default project, new form, and main form
- Managing multiple project versions and team development
- Compiling, building, and running projects

What is a project?

See also

A project is a collection of files that make up an application or dynamic-link library. Some of these files are created at design time. Others are generated when you compile the project source code.

You can combine projects into a project group. Project groups let you organize and work on related projects, such as applications and DLLs that function together or parts of a multi-tiered application.

You can view the files that make up a project in the Project Manager (see Viewing a project's contents). Although you can edit many of these files directly, it is often easier and more reliable to use the visual tools in Delphi. You should, however, understand the files and file types that make up a project.

Single project files, which describe individual projects, have a .DPR extension. Project files contain directions for building an application or library. When you add and remove files using the Project Manager, Delphi updates the project file.

Delphi reads the **uses** clause of the project (.DPR) file to determine which units are part of a project. Only units that appear in the **uses** clause followed by the keyword **in** and a file name are considered part of the current project. For example, here is the default project file for new applications:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

The project defined above uses two units: Forms and Unit1. Only Unit1, however, is actually part of the project.

The project group file contains make commands to build the projects in the project group, has a .BPG extension. Any time you add a project to the project group, a reference to that project is added to the .BPG file.

You can also add additional types of files to your project (using drag and drop or Project|Add to Project) and view them in the editor as text files. You can also add resource files, and they are compiled into .RES files and linked when you compile the project.

Package files

See also

Delphi packages are specially compiled dynamic-link libraries. You can create runtime packages to allow code sharing among applications. You can create design-time packages to easily manipulate components in the IDE. You use design-time packages to create special property editors for custom components.

Packages have file extensions of .BPL and package source files have the extension .DPK. When you rebuild any project that contains a package, the package is implicitly recompiled, if necessary.

Packages are described in detail in About packages.

Desktop file

See also

Delphi can generate a desktop file that maintains the state of your desktop, such as which windows are open and in what positions. This allows you to restore your project's workspace whenever you reopen the Delphi project.

The desktop-settings file has the same name as the project file, but with the extension .DSK.

To generate and automatically save a desktop file:

1. Choose Tools|Environment Options…
2. On the Preferences page, look for Autosave options, and check the Desktop box.

   Delphi generates and saves a project .DSK file whenever you close the project. The file is stored in your main project directory.

When you create a desktop file for your projects, Delphi opens the project with the same window setup that you had when you last closed the project.

Project file

See also

Every Delphi project contains Object Pascal source code that Delphi compiles into the finished application or dynamic-link library. The central point for the project's source code is called the project file. Delphi updates this file throughout the development of the project.

The project file contains references to all the forms and units used by the project. When you load, save, or compile a project, Delphi knows which other files to act on by looking at the project file.

By default, Delphi project files have the extension .DPR. (Delphi project). When you compile or run the project, the compiler produces an executable file, a dynamic-link library, a package, etc. on disk with the same name as the project file, but with the extension .EXE, .DLL, .BPL, etc. as appropriate.

Viewing the project file

Caution: Because Delphi maintains the project file, you should not modify it manually. You can change the project file by using the Project Manager. Doing so ensures that Delphi keeps all the project's files synchronized.

The main reason to view the project file is so you can see the units and forms that make up the project, and which form is specified as the application's main form. As you add forms and units to the project, you can see that Delphi updates the project source code.

To display the project file, use either of these methods:

- Choose Project|View Source.
- Right-click in the Project Manager (with a file or part of a project selected), and choose View Source.

To display the project file, use either of these methods:

- Choose Project|View Makefile.
- Right-click in the Project Manager (with a file or part of a project selected), and choose View Project Makefile.

The contents of the project file appear in a page in the Code editor. (When you're finished viewing the project file, close it in the Code editor.)

Delphi generates the following source code for a default, blank project:

```
program Project1; { declares project identifier }
uses  { indicates units used by project... }
 Forms,  { ...including non-form units... }
 Unit1 in 'UNIT1.PAS' {Form1};      { ...and form units }
{$R *.RES} { links in resource file }
begin  { start of main program block }
 Application.Initialize;
 Application.CreateForm(TForm1, Form1);   { auto-creates first form }
 Application.Run; { runs the application }
end. { end of main program block }
```

- Project1 is the identifier for the project. Delphi also uses this as the default name for the project file. When you save a project, you can name it
  The reserved word **program** indicates that this project is an application. If the project were a dynamic-link library, the reserved word **library** would appear instead.
- The **uses** clause tells the compiler which units to link into the project. *Forms* is the identifier of a standard unit used by all Delphi projects that use forms.
- Unit1 is the unit identifier for another unit, which contains a form. UNIT1.PASCPP that represents the name of the file that contains the unit's source code. These names are identical and must remain so in order for your project to compile correctly.
- The reserved word **in** tells the compiler where to find the source-code file for each unit. The comment {Form1} specifies the instance identifier for the form associated with this unit (this would not

appear in the clause if this were not a form-associated unit.) This is the same as the Name property of the form. You use the Object Inspector to name the form, and Delphi maintains the name in the project file.

- The **$R** compiler directive specifies that the file with the same base name as the project and the extension .RES should be linked into the project. The project's resource file contains such items as the project's icon image. For more information, search online Help for the Resource File Directive topic.
- The **begin..end** block is the main source-code block for the project.
- The Application.CreateForm statement loads the form specified in its argument. Delphi adds an Application.CreateForm statement to the project file for each form you add to the project. The statements are listed in the order the forms are added to the project. This is the order that the forms will be created in memory at runtime. If you want to change this order, do not edit the project source code. Use the Project| Options menu command. (For more information see Setting project options.)
- The Application.Run statement starts your application.

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. For more information, see Adding existing forms and units to a project.

Form files

See also

Forms are a very visible part of most Delphi projects. Normally, you design forms using Delphi's visual tools, and Delphi stores a description of the designed forms in form files. Form files (extension .DFM) describe each component in your form, including the values of all persistent properties. You do not specify the form file programmatically; you simply create the form by selecting components from the Component palette and customizing them to suit your needs by setting properties and events with the Object Inspector.

Each form in a Delphi project also has an associated unit. The unit contains the source code for any event handlers attached to the events of the form or the components it contains. A unit associated with a form is sometimes called a *form unit*. When you save a form unit or a project containing unsaved forms, Delphi prompts you to enter a name for each unit, which it uses as the name of the unit file, appending the extension .PAS. The form file gets the same name, but with the extension .DFM (Delphi form). You can use any extension you want on your unit files, but Delphi expects the .DFM extension on the corresponding form file.

Warning: You can't define more than one form in a single unit. This is because each DFM file can only describe a single form (or data module).

Form files can be saved in either binary or text format. The Environment Options dialog lets you indicate which format you want to use for newly created forms.

To view the text version of .DFM files in the Code editor,

1. Select the form.
2. Right-click and choose View As Text.

To return to viewing the form graphically, follow the above steps and choose View As Form.

To change the format (text or binary) in which the form file is saved,

1. Select the form.
2. Right-click and check or uncheck Text DFM.

Tip: You may want to archive your forms as text as they are less susceptible to data corruption.

Unit files

See also

Delphi's Object Pascal language supports separately compiled modules of code called units. Using units promotes structured, reusable code across projects. The most common units in Delphi projects are form units, which contain the event handlers and other code for the forms used in Delphi projects. But units don't have to have forms associated with them. You can create and save a unit as a standalone file that any project can use. For example, you can write your own procedures, functions, DLLs, and components, and put their source code in a separate unit file that has no associated form.

If you open and save a default new project, the project directory initially contains one unit source-code file (UNIT1.PAS) and its associated form file (UNIT1.DFM).

When you compile or run the project or perform a syntax check on the project, Delphi's compiler produces an intermediate output file on disk from each unit's source code. By default the compiled version of each unit is stored in a separate binary-format file with the same name as the unit file, but with the extension .DCU (Delphi compiled unit). You should never need to open these binary files, and you do not need to distribute them with the completed project. The compiled-unit format is specific to the Delphi compiler, and enables rapid compiling and linking.

Note: As an option, you can choose to have the compiler generate standard Intel object files (with the extension .OBJ) for greater compatibility with other compilers, but this greatly reduces the speed of compiling and linking your project. It should have no effect on the quality of the final generated code, however.

Unit files for forms

Most unit files you'll work with will probably be associated with forms. Whenever you create a new form, Delphi creates the corresponding unit file with the following code. The default unit identifier is incremented (Unit2, Unit3, and so on) for each new form.

```
unit Unit1;{ unit identifier }
interface
uses { uses clause }
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;
type
 TForm1 = class(TForm)        { class declaration }
 private
   { Private declarations }
 public
   { Public declarations }
 end;
var
 Form1: TForm1;       { instance declaration }
implementation
{$R *.DFM} { compiler directive to link form file }
end.
```

The **type** declaration (or class declaration) part introduces the form as a *class*. A class is simply an object, which you will recognize if you are familiar with previous versions of Borland Pascal products, or another object-oriented programming language.

The default type declaration makes the new form a descendant of the generic form class, *TForm*. This means it contains all the behaviors and characteristics of a *TForm* object.

The variable declaration declares your form as an instance of the class TForm1.

The **$R** compiler directive links the *TForm*'s binary form file. This adds the form file(s) in your project to the compiled executable.

Caution: Do not remove the {**$R *.DFM}** directive from a form unit file. Doing so will result in code that will never work correctly.

Caution: Do not add more than one form into a single unit file. The associated form file (.DFM) can only describe a single form.

Unit files for procedures and functions

You can write custom procedures or functions within a unit that's associated with a form. However, if you want to reuse the routines that you write, it's better to create a separate unit to contain those routines. By creating standalone units that have no associated forms, you can easily make your procedures and functions available to other projects.

To create a unit file not associated with a form:

1. Choose File|New.
2. Choose Unit in the New Items dialog box.
3. Choose OK.

It is not necessary to have a project open unless you want the new unit to be part of a project.

See Programs and units for more information about units. Units are also used when you create new components, as described in the online book called Creating custom components.

Generated files

See also

Delphi generates other files in conjunction with maintaining and compiling your project, most of which you never need to consider. However, you should not delete these files. These include the following:

File Extension	Description

CFG	Project configuration file. Stores project configuration settings. It has the same name as the project file, but with the extension .CFG.
	The compiler searches for a dcc32.cfg in the compiler exe directory, then for dcc32.cfg in the current directory, and then finally for project name.cfg in the project directory. You can therefore type
	`dcc32 project1`
	on the command line and compile the project with all the same options as specified in the IDE. You can also type
	`make -f projectgroup1.bpg`
	to make all the targets in the project group.

DCI	Holds Code Insight changes you make in the IDE.
DCT	Holds component template changes you make in the IDE.
DMT	Holds changes you make to menu templates in the IDE (may cause a "Stream Read Error" if corrupt; deleting it will lose your changes to menu templates but resolve the error).
DOF	Delphi options file. Contains the current settings for project options, such as compiler and linker settings, directories, conditional directives, and command-line parameters. Set these options using the Project Options dialog box (Project| Options), but Delphi saves them in text form for easy maintenance, version control, and sharing.
DRO	Holds changes when things are added to the repository. Can be deleted but your additions to the repository will lost. Can be restored to default by copying the default ..\bin\delphi32.dro from the product CD.
DSK	Desktop settings. Saves the current state of the desktop, such as which windows are open, and in what positions. Used when Tools|Environment Options| Preferences Autosave Desktop is checked. Depending on where you're working, you'll save desktop settings for the project (Project.dsk), project group (Group.dsk), or IDE (..\bin\bcb.dsk). Delete these if you do not want to save the desktop settings.
RES	Contains the version info resource (if required) and the application's main icon. This file may also contain other resources used within the application but these are preserved as is. Do not delete this file if your application contains any references to it.
TDS	Holds the external debug symbol table.
TODO	To-do list file. Includes the current to-do list for the project. It has the same name as the project file, but with the extension .TODO.

Naming unit and project source code files

See also

As you open new units in a project, Delphi gives them default names: UNIT1.PAS UNIT2.PAS, UNIT3.PAS, and so on. You can change a unit's default name to a meaningful (and unique) name when you save the project.

Delphi also supplies a default name for the project file (PROJECT1.DPR) which you can rename when you save the project.

All unit and project file names must be legal Object Pascal identifiers. When the compiler looks for a unit or project file, it first searches for a file with the full name of the unit or project identifier. If it does not find that file, it will then search for a version of the identifier name, truncated to eight characters. This is for backward compatibility and for compatibility with file servers that store only short file names. You should not manually truncate your file names.

Storing a project

See also

You should store each project in its own directory. By default, projects are stored in the Projects directory. Projects can share forms, files, and resources located in almost any directory, but it's best to keep the central project file and any other files specific to the project in a dedicated directory. See Sharing objects for more information about project templates and shared forms.

Viewing a project's contents

See also

The Project Manager displays the contents of your current project group and any project it contains. It allows you to navigate among various projects and the projects' constituent files. You can perform project management tasks using its toolbar and context (right-click) menus.

To display the Project Manager:

1. Open a project.
2. Choose View|Project Manager.

To view a unit, form or other file, double-click it. It is displayed in the Code editor.

The Project Manager gives you a high-level view of the projects contained in a project group, and of the form, unit files, resource, object, and library files contained in the project file. You can use the Project Manager to open, add, save, and remove project files. You can also use the Project Manager to access the Project Options dialog box, which lets you configure your default project settings.

You can also add additional types of files to your project (using drag and drop or Project|Add to Project) and view them in the editor as text files. You can also add resource files, and they are compiled into RES files and linked when you compile the project.

The Project Manager is an invaluable tool if you share files among different projects because it lets you quickly find each file in the project (see Adding existing forms and units to a project and Sharing objects). It is also useful when backing up all the files in your project (see Backing up a project).

Certain operations, such as commands available on context menus, operate on the active project. The active project is the one that is highlighted in bold in the Project Manager and is the project you are currently working on. The active project is also shown in the project selector which is the top left of the Project Manager. See Selecting a project to work on.

To make a project the active project:

1. Display the Project Manager.
2. Select the project you want to make active and click Activate.

When you view a project item, such as a form or source code, the Project Manager automatically makes the project it belongs to the active project. Projects you select using the project selector automatically become the active project.

Managing projects

See also

You can use the Project Manager to manage multiple projects within a project group. The Project Manager displays information about the status and file content of the current project and provides a convenient way to open, add, save, and remove project files (you can do some of these tasks from the File menu as well). You can also create new projects to add to the current project group.

You can perform the following tasks from the Project Manager:

- Selecting a project to work on
- Searching for files
- Removing items in the Project Manager
- Copying in the Project Manager
- Getting project path and unit information
- Adding projects to the project group
- Adding existing forms and units to a project
- Removing forms and units from a project
- Copying a project

Using the Project Manager

See also

The Project Manager window displays information about the status and file content of the currently open project. If the project is part of a project group, it displays information about all projects within the project group.

With the Project Manager, you can easily visualize how all your project files are related. Also, you can select any file displayed, right-click, and perform various project management tasks, such as opening, adding or removing files, and compiling your projects.

Also, with the Project Manager you can add related projects to a project group. This way, you can compile multiple executables at the same time.

Use the Project Manager to perform project-related tasks such as adding and removing projects rather than editing the project file because Delphi tracks and updates the affected files in your project.

If you save your desktop settings, you can have the Project Manager window opened by default when you open any project. The Project Manager is dockable (right-click and choose Dockable) so it can be docked or placed alongside other dockable windows.

The main elements of the Project Manager window are the

- Project Manager file view
- Project Selector
- Project Manager toolbar
- Project Manager status bar
- Project Manager context menu

Project Manager file view

See also

The main area of the Project Manager provides a tree view of all the files in your project or project group. The file view displays all the unit files in your project and the paths to the files. You can also add other file types to your Delphi project (*.HTM, *.HTML, and *.TXT files, for example). However, Delphi will not handle these files in any special way.

As you add and remove files from a project using the Project Manager, you can see that Delphi updates the project file (.DPR).

If you have multiple projects, you can easily see all the related projects contained in the project group.

The Project Manager displays the currently active project in **bold** and displays its name in the project selector. See Viewing a project's contents for how to make a different project the active project.

Caution: Delphi has mechanisms for automatically tracking the files that make up a project and for keeping the project file updated. Avoid editing project files manually unless you have a thorough understanding of this process and its ramifications. By editing a project file, you circumvent Delphi's automated project management and risk maintaining inaccurate information about project components. Compilation failures and other problems can result. If you edit the project file manually, you must close the file and reopen it to update the Project Manager display.

Project Manager toolbar

See also

You can display the toolbar (if it is not already displayed) by right-clicking in the Project Manager and choosing Toolbar. The toolbar displays buttons that provide quick access to common project tasks. You can also perform these tasks by using a context menu or choosing a menu command.

Button	Context menu command	Menu command	Function	Comment
New	Add New Project	File|New or Project|Add New Project	Displays the New Items dialog so that you can add a new project to the current project group.	To add an existing project to this project group, select the project group, right-click, and choose Add Existing Project.
Remove	Remove Project	Project|Remove from Project	Removes the selected project from the current project group.	To remove individual files from a project, select the file, right-click, and choose Remove From Project. You can also use the Delete key. See Removing items in the Project Manager.
Activate	Activate	NA	Makes the selected project active so that you can make changes to the project.	You can activate a project in the Project Manager by double-clicking on it. You can also choose a project from the current project group using the project selector in the Project Manager.

Project Manager status bar

See also

You can display the status bar (if it is not already displayed) by right-clicking in the Project Manager and choosing Status Bar. The area at the bottom of the Project Manager window displays the full path name of the selected file.

The project file path name can be a useful reference if you are bringing many forms and units that reside in locations other than the main project directory into the current project.

Project Manager context menus

See also

The Project Manager allows you quick access to commands by right-clicking on any selected item in the file view. You get a different context menu depending on what type of file you select. Note that all of the context menus include the choices Toolbar, Status Bar, and Dockable. These options apply to the Project Manager.

File Type	Provides commands that	Available Commands
Project Group	Act on the project group as a whole.	Add new project, Add existing project, Save Project Group, Save Project Group As, View Project Group Source.
Project	Act on the current project.	Add, Remove File, Save, Options, Activate, Make, Build, View Source, View Project Makefile, Close, Remove Project, Build Sooner, Build Later.
File	Act on the current file. Which commands are available depends on the type of file selected.	Can be all or some of the following: Open, Remove from Project, Save, Save As, Compile.

Creating a project group

See also

Create project groups to handle related projects at once. For example, you can create a project group that contains multiple executable files such as a .DLL and an .EXE. By organizing them into a group, you can compile them at the same time.

To create a project group:

1. Choose View|Project Manager to display the Project Manager, if necessary.
   If no project is currently loaded, the Project Manager lists <No Project Group>.
   If a project is currently loaded, the Project Manager lists <ProjectGroup1>.
2. Select the project group, right-click, and choose either:
   Add New Project to open the New Items dialog box to add a new project.
   Add Existing Project to add an existing project to this project group.
3. When you have completed adding projects, select ProjectGroup1, right-click, and choose Save to rename the project group to a meaningful name.

To manage a project group:

- Select the project group, right-click, and a context menu appears.
- Whenever you open a project that is not currently part of a project group, Delphi displays the project as ProjectGroup1. You may choose to save the Project Group, thereby creating a project group for this project. However, it is not necessary.

Selecting a project to work on

See also

The project selector is the list box at the top left of the Project Manager. You can select a project from the project group using the project selector. A drop-down list shows all projects in the current project group. The project you select becomes the active project. The active project is the one that is highlighted in bold in the Project Manager and is the project you are currently working on.

Searching for files

See also

You can locate files in large projects using an incremental search within the Project Manager. With the Project Manager displayed, start typing the name of a file and the Project Manager moves to the nearest match. Press the Spacebar to repeat the last search.

If you share files among different projects, using the Project Manager is highly recommended because you can quickly and easily tell the location of each file in the project. This is especially helpful to know when creating backups that include all files the project uses. (See Backing up a project for more information.)

Removing items in the Project Manager

See also

You can remove selected items in the Project Manager in the following ways:

- Click the Remove button in the Project Manager.
- Press the Delete key on the keyboard.
- Choose Project|Remove from Project from the main menu.

The Project|Remove from Project dialog allows you to multiselect and remove multiple items from the project.

If a project is selected, the whole project including all it contains is deleted from the current project group. If one file is selected, only that file is deleted. Realize that the files are not deleted from disk, they are only disassociated from the current project or group. The Project Manager verifies that you want to remove the project or item before doing so.

Note If you copy a file from one project to another then remove the first project without saving the second one, the copied item is removed from both projects. The Project Manager prompts you save the project before removing the item. If you save the project when prompted to do so, the copied item is retained.

The section Removing forms and units from a project provides more details about removing forms from projects.

Copying in the Project Manager

See also

You can addy items into projects in the Project Manager from other Windows folders using drag and drop. You can also copy and paste items from one project to another within a project group.

For information on copying an entire project, see Copying a project.

Note: When you copy items in the Project Manager, you are not making a physical copy on the disk. You are including the file or item as part of the active project.

Drag and drop

You can drag one or more selected items from any Windows folder and drop them into a project in the Project Manager:

1. Activate the project where you want the item to be added. The active project appears in bold in the Project Manager.
2. In any Windows folder (such as the Windows Explorer or My Computer), select one or more items to copy.
3. Drag the item or items onto the name of a project in the Project Manager.
4. Drop the items.

   You are asked to verify that you want to add the item or items, and if you click Yes, the items are added to the active project.

1. Save the project where you added the items

Copying files between projects

You can copy any project item from one project to another:

1. Select the project item you want to copy.
2. Choose Edit|Copy (or type Ctrl+C).
3. Select the project where you want to place the copy (or move the cursor where you want to place the copy).
4. Paste the copy using Edit|Paste (or Ctrl+V).

1. Save the project where you placed the copy.

You can also use drag and drop to copy items from one project to another.

Note If you copy a file from one project to another then remove the first project without saving the second one, the copied item is removed from both projects. The Project Manager prompts you save the project before removing the item. If you save the project when prompted to do so, the copied item is retained.

Getting project path and unit information

See also

The status bar at the bottom of the Project Manager window displays the full path name of the project file. You can display and hide the status bar by right-clicking and choosing Status Bar.

The project file path name can be a useful reference if you are bringing many forms and units that reside in locations other than the main project directory into the current project.

Adding projects to the project group

See also

To add a new project to this project group, select the project group, right-click, and choose New. You can also use the New button on the Project Manager toolbar. Types of projects you can add are shown in the Object Repository. You can choose Project|Add Existing Project to add a project that was already created to the current project group.

Adding existing forms and units to a project

See also

A project can share any existing form and unit file including those which reside outside the project directory tree. This includes custom Object Pascal procedures and functions that have been written as standalone routines.

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

Note: The path that Delphi uses for the shared file is either absolute or relative, depending on where the file is located. If the shared file is located on the same disk drive as your project, the Delphi uses a relative path for the file; otherwise, it uses an absolute path.

When you compile your project, it does not matter whether the files that make up the project reside in the project directory, a subdirectory of the project directory, or any other location. The compiler treats shared files the same as those created by the project itself.

To add a shared file to the current project, do one of the following:

- Choose Project|Add to Project.
- Choose the Add File to Project button on the toolbar.
- Choose Add from the Project Manager context menu.

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 Manager's file list displays the path to the shared file.

Using Borland Pascal source code units

If you have existing source code units for custom procedures or functions written in Borland Pascal or Turbo Pascal, you can use these units in a Delphi project. You add these files in the same way as files created in Delphi.

Removing forms and units 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 from the **uses** clause of the .DPR file. Using the procedures in Removing items in the Project Manager to remove a unit that has an associated form also removes the form from the project.

You can remove multiple items from a project with the dialog that displays after selecting Project| Remove from Project.

Removing a file from the project ends its association with the project; it does not delete the file from disk.

Caution: Do not use Windows file management programs to delete Delphi project files from disk until you have performed the preceding removal process in every project that uses the files. Otherwise, the project file of each project using the deleted files retains references to them. When you open the project again, Delphi will attempt to locate the deleted files and display error messages for each file it cannot find. When the project opens, the information about its constituent files in the Project Manager is inaccurate.

Toggling between form image and unit source code

To switch between viewing the current form and its unit source code, use any of the following methods:

- Press F12.
- Choose View|Toggle Form/Unit.
- Click the Toggle Form/Unit button on the toolbar.
- In the Project Manager, double-click either the form or the source file.

Bringing a window to the front

If you have a number of windows open, and you want to bring one of them to the front:

1. Choose View|Window List or press Alt+0 (zero).

1. Double-click the name of the window you want to bring to the front.

You can also use command on the View menu to bring the Project Manager, Code Explorer, Object Inspector, and other windows to the top. Additionally, you can use docking to keep preferred windows on top.

Viewing forms and units

See also

To display a list of forms or units associated with the current project, choose View|Forms or View|Units. Select a form or unit from the dialog box that appears, and choose OK.

The Project Manager also provides commands that let you quickly navigate to the source code and form images contained in your projects.

To view a specific form, double-click on the form listing in the Project Manager; Delphi gives focus to that form image. Double-clicking a unit listing in the Project Manager opens the Code editor and displays the selected unit source file. If the file is not currently open, Delphi opens it for you.

Saving projects and individual project files

See also

At any time during project development, you can save an open project in its current state to the project directory. You can optionally save a copy of the project in a different directory under the same or a different name.

You can also add your project to the Object Repository so that you or others can use it as a template. For more information, see Adding items to the Object Repository.

You are not limited to saving a project as a whole; you can save individual constituent files of a project, including saving a copy of a file to a different directory or with a different file name.

Saving a project

See also

Note: If the project was begun from a project template, the Object Repository selection process creates the project directory. Otherwise, Delphi saves projects by default to the \Projects directory if you start from the Start menu (Windows 95 and Windows NT). If you start Delphi by clicking a shortcut icon, the default is \BIN which you can change to your preferred default location by right-clicking on the icon and choosing Properties. On Windows 98, you cannot customize the default location.

To save all open project files to the project directory, use one of the following methods:

- Choose File|Save All.
- Choose the Save All button on the toolbar.
- Right-click in the Project Manager with a project selected, and choose Save.

From here, the save process for projects varies somewhat depending on whether you have previously saved the project:

- If you have not previously saved the project, Delphi displays the Save Unit As dialog box. This dialog box prompts you to supply a name for each open unit file that has been created in the current project. (You are not prompted for any shared files you might have added to the project. See Sharing objects.)

  After you name the unit file, Delphi prompts you to name the project file before saving it to disk. This processing order ensures that the unit and form file names you just specified are correctly registered in the project file's source code.

- If you have previously saved the project, all open files that reside in the project directory are saved to disk if they have been modified.

  If you have created any new forms or units in the project since the last save, the Save Unit As dialog box appears and prompts you to name those unit files before saving them.

  The project file is then updated to reflect any new units and any newly shared files that you have specified for the project to use.

To save the project group file, select the project group in the Project Manager, right-click and choose Save Project Group or Save Project Group As. The group is saved in the project group file *projectgroup*.BPG.

Copying a project

See also

You can save a separate version of an open project in a directory other than the project directory by choosing File|Save Project As. However, because the open project might use shared files in addition to files that were created as part of the current project, the Save Project As command saves a copy of just the project file, project options file, and the project resource file to the new location.

Important: No unit files are saved to the new location. When you open the copied version, the Project Manager displays all units in the copied project as shared files; that is, none of them reside in the directory you copied the project to.

To copy an open project:

1. Choose File|Save Project As to display the Save *projectname* As dialog box.

   You are prompted for the new name and the location of the project file.
2. Select the directory where you want to copy the project file.
3. To save the project file under a different name, enter the new name in the File Name edit box.

   If a project file with the same name exists in the directory you specify, you're asked if you want to overwrite the existing file.
4. Choose OK.

   The open project is now the project you just saved.

In addition to saving the project file, project options file, and project resource file to the new name/location, Delphi also saves any modified unit files (in their current location). Therefore, you won't be prompted to save these changes again when you close the project. When you open either version of the project, all changes saved with the Save As operation are reflected in both places.

If you check the 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 want separate copies of any of those files in the new project directory, you need to save them individually to the new location using File|Save As. (See Backing up a project for more information.)

If you leave the new project unchanged, it continues to use the files in their present (that is, old) location as shared files, which may or may not be what you want. If you don't understand how the new project is using its files, you can run into problems later.

Caution: Do not use file management tools other than those in Delphi to save a copy of a project to a new location.

Saving individual files

See also

You can save individual files in a project, or non-project files (such as text files) that you might have open in the Code editor.

To save an individual file:

1. Bring the file to the front of the Code editor by selecting its tab.
2. Choose File|Save As. If this is the first time you've saved the file, you're prompted to name it.
3. If necessary, name the file and choose OK.

   Delphi saves the file.

Note: Each file must have a unique prefix, even if their extensions are different. For example, if you specify About as the file name for both the form and project file, Delphi displays the following error message:

```
The project already contains a module named About.
```

If you see this error message, Delphi won't save the project file (or whichever file you named secondly). Save the file again with a different name.

To save a file under a different name or location:

1. Bring the file to the topmost level of the Code editor by selecting its tab.
2. Choose File|Save As.

   The Save File As dialog box appears.
3. Specify the new file name, or 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 file still exists, but isn't included in the project any longer.

Caution: If using shortened windows long file names, do so consistently. Delphi views the shortened name and the full long file name as two different files.

Backing up a project

See also

Backing up a project can be a simple matter of copying directories or can involve some additional steps. This depends on how your project directories are structured and whether the project uses files from outside its own directory tree.

The project directory isn't encoded into the project file. The project file does, however, record the location of all the other files in the project. If these files reside in subdirectories of the main project directory, all path information is relative, which makes backup easy. You could back up such a project by copying (not moving) 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 the project directory 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 other backup processes already preserve these outside files, then there is probably no need to make separate copies of them in the backup project directory.

Sharing objects

See also

The Object Repository (Tools|Repository) is a versatile tool that makes it possible to easily share (or copy) forms, dialog boxes, and data modules across projects and within a single project. It also provides project templates as starting points for new projects. By adding forms, dialog boxes, and data modules to the Object Repository, you make them available to other projects. In fact, you can add an entire project to the Object Repository as a template for future projects.

You'll also see wizards in the Object Repository. Wizards are small applications that lead you through a series of dialog boxes to create a form or project. Delphi provides a number of wizards, and you can also create and add your own customized wizards to simplify and standardize your work.

The repository stores settings in a text file named DELPHI32.DRO (Delphi repository objects) in the \BIN directory that contains references to the items that appear in the Object Repository dialog box and the New Items dialog box. You can open this file in any text editor.

Sharing items within a project

See also

It's also easy for you to share items *within* a project without having to add them to the Object Repository: when you open the New Items dialog box (File|New), you'll see a page tab with the name of 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 an existing item, and customize it as needed.

Sharing objects in a team environment

See also

To share objects in a team environment, you need to specify a directory that's available to team members. After you do this, another DELPHI32.DRO file is created in the specified directory as soon as you add an item to the Object Repository. The new DELPHI32.DRO text file contains pointers to the objects you want to share.

To specify a shared repository directory:

1. Choose Tools|Environment Options.
2. On the Preferences page, locate the Shared Repository panel.
3. In the Directory edit box, enter the name of the directory where you want to locate the shared repository.

The location of your shared directory is stored in the Windows registry. Changing the location in the Environment Options dialog box changes it in the registry as well.

To share Object Repository items among team members, every member's Directory setting in the Environment Options dialog box must point to the same location.

Copy, Inherit, or Use?

See also

To gain access to items in the Object Repository, choose File|New. The New Items dialog box appears, showing you all the items in the Object Repository. You have three options for adding an item to your project:

- Copy
- Inherit
- Use

Copy option

Select Copy to make an exact copy of the selected item and add the copy to your project. Future changes made to the item in the Object Repository will not be reflected in your copy, and alterations made to your copy will not affect the original Object Repository item.

Copy is the only option available for using project templates.

Inherit option

Select Inherit to derive a new class from the selected item in the Object Repository and add the new class to your project. The Inherit option creates a link to the ancestor item in the repository. When you recompile your project, any changes that have been made to the item in the Object Repository are reflected in your derived class. These changes apply in addition to any changes or additions you make to the item in your project. Changes made to your derived class do not affect the shared item in the Object Repository.

Inherit is available 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.

Use option

Select Use when you want the selected item itself to become part of your project. In this case, you are not making a copy of the item; you are using the item itself, "live." Using an item is like reverse inheritance: instead of inheriting changes others make to an item, they inherit your changes when they use the item in the repository. Changes to the item appear in *all* projects that have added the item with the Inherit or Use options selected.

Caution: The Use option is available for forms, dialog boxes, and data modules, but you should use it carefully. Make sure the changes you make to an item are thoroughly tested before letting others copy it into their applications from the repository.

Note: The Use option is the only option available in wizards, whether form wizards or project wizards. Using a wizard doesn't actually add shared code, but rather runs a process that generates its own code.

Using project templates

See also

Project templates are predesigned projects you can use as starting points for your own projects.

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. If you specify a directory that doesn't exist, Delphi creates it for you.

Delphi copies the template files to the project directory. You can then modify the project, adding new forms and units, or use it by 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.

Adding items to the Object Repository

See also

You can add your own projects, forms, and data modules to those already available in the Object Repository.

To add an item to the Object Repository:

1. If the item is a project or is in a project, open the project.
2. For a project, choose Project|Add To Repository. For a form or data module, right-click the item and choose Add To Repository from the context menu.
3. Type a description, title, and author.

   The title will appear in the Object Repository window and in the New Items dialog box (File|New).
4. Decide where you want this item to appear in the New Items dialog box, and select that page from the Page combo box. Or, type the name of the page.

   If you type the name of a page that doesn't exist, Delphi creates a new page for you, and your new page name will appear on a tab of the New Items dialog box.
5. Choose Browse to select an icon to represent the object in the Object Repository.
6. Choose OK.

Modifying a shared form

See also

If several projects share a form in the Object Repository, then modifications you make to the form can affect all projects, depending on how the form is imported into each project. If a project copies a form from the Object Repository, then later modifications to the form in the Object Repository have no effect on the project.

If a project inherits a form from the Object Repository, then each time the project is compiled, it inherits the latest version of the form from the Object Repository, including any changes made since the project was last compiled. If a project "uses" a form from the Object Repository, then any time you make changes to the form in the project, the changes are stored in the Object Repository directly where other applications can copy or inherit them.

If you know that other projects inherit a form in the Object Repository, but you do not want to replicate your changes to those projects, there are several ways to prevent inheritance:

- Save the form under a different name and use the renamed form in your project instead of the original.
- Make the changes to the form at runtime instead of at design time.
- Make the shared form a component that can be installed onto the Component palette. This has the added advantage of enabling users to customize the form at design time.

If you expect to be the only user of the form, and you don't plan extensive or frequent changes, runtime customization is probably acceptable. If you plan on using the form in many different applications, runtime customization involves more coding for you and other developers. In this case, it's usually more convenient, whenever possible, to make the form a component that other users or developers can install onto their Component palette.

Specifying a default project, new form, and main form

See also

You can specify defaults for a new project, a new form, and a main form. You always have the option to override the defaults by choosing File|New and selecting from the New Items dialog box.

Specifying the default new project

See also

The default new project opens whenever you choose File|New Application. If you haven't specified a default project, Delphi creates a blank project with an empty form. You might want to specify a project you're using as a template to be the default new project.

You can also designate a project wizard to run by default when you start a new project. A project wizard 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 you want as the default new project from the Objects list.
4. With the project you want selected, check New Project.
5. Choose OK to register the new default setting.

Specifying the default new form

See also

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 as the default new form. Or you can designate a form wizard 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 you want as the default new form from the Objects list.
4. With the form you want selected, check New Form.
5. Choose OK to register the new default setting.

Specifying the default main form

See also

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 you want to use 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 you want as the default main form from the Objects list.
4. With the form you want selected, check Main Form.
5. Choose OK to register the new default setting.

Setting project options

See also

You can change project settings in the Project Options dialog box. To open this dialog box, do one of the following:

- Choose Project|Options.
- With a project selected in the Project Manager right-click and choose Options.

The settings you change affect only the current open project, unless you check the Default check box (see the next topic). If you change any of the default settings, an *options file* with a file extension .DOF (Delphi options file) is created in the project directory the next time you save the project. So when you reopen the project in future work sessions, the project options you set are in effect.

These topics discuss only those project options that pertain to project management. For detailed information about the options on any given page of the dialog box, click the Help button on that page.

Setting options that affect all new projects

See also

The Project Options dialog box contains a check box labeled Default. Checking this control writes the current settings from the Compiler, Linker, Directories/Conditionals, Packages, and VersionInfo pages of the Project Options dialog box 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 learn about information on all of the Project Options pages, choose Project|Options and display any page. Click Help or press F1 for Help on setting the options.

Restoring Delphi's original default settings

See also

To restore Delphi's original default project settings, delete, or rename the DEFPROJ.DOF file.

Setting environment preferences

See also

In addition to changing project settings you can customize the Delphi environment (editor, designer, debugger, and compiler). Preferences you set in the Environment Options dialog box affect all Delphi projects.

To specify environment settings, choose Tools|Environment Options. Click the Help button on any page of the Environment Options dialog box for help with that page.

Note: If you share your installation of Delphi with other users, it's possible that another user has modified the default option settings. In a shared-installation situation, it's a good idea to check environment options before creating a new project.

Managing multiple project versions and team development

See also

When you are developing a complex programming project in a team setting, or managing several development projects, you might soon develop the need for a version control system (VCS). A version control system can archive files, control access to project files, and track multiple versions of your projects.

Some versions of Delphi ship with TeamSource, a tool specificly designed to manage the complexities of developing in a team environment. Not only does TeamSource use a version control system to archive files, it provides a mechanism for reconciling changes made by individual developers with the changes to the overall project.

Compiling, building, and running projects

See also

All projects have as a target a single distributable executable file, either an .EXE or a .DLL file. You can view or test your application at various stages of development by compiling, building, or running it. You can also test the validity of your source code without attempting to compile the project.

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

Compiling a project

See also

To compile all the source-code files that have changed since the last time you compiled them, Choose Project|Compile *projectname*.

When you choose this command, this is what happens:

- 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 (Delphi compiled unit) extension for each unit.
  If the compiler can't locate the source-code file for a unit, the unit isn't recompiled.
- If the interface part of a unit's source code has changed, all the other units that depend on it are recompiled.
  To learn about the **interface** section of a unit, see The Interface section.
- 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 executable file (or dynamic-link library). This file is given an .EXE (or .DLL) file extension and the same file name as the project source code file. This file now contains all the compiled code and forms found in the individual units, and the program is ready to run.

You can choose to compile only portions of your code if you use **{$IFDEF}** conditional directives and predefined symbols in your code. For information about conditional compilation, see "Compiler directives" of this book or "conditional directives" in online Help.

Obtaining compile status information

You can get information about the compile status of your project by displaying the Information dialog box (Project|Information). 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 file 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 Progress box in the Environment Options dialog box, Preferences page.

Building a project

See also

To compile all the source-code files in your project, regardless of when they were last compiled, Choose Project|Build *projectname*.

The result of this command is similar to that of the Project|Compile command, except that all units in the project are compiled, regardless of whether or not they have changed since the last compile. This technique is useful when you are unsure of exactly which files have or have not been changed, or when you simply want to ensure that all files are current and synchronized. It's also important to use Build when you've changed global compiler directives, to ensure that all code compiles in the proper state.

Running a project

See also

You can test run a project from within Delphi, or you can run the compiled .EXE file from the Windows operating environment without having to run Delphi.

To compile and then run your application from within Delphi, either:

- Choose Run|Run.
- Choose the Run button on the toolbar.

These actions are identical to choosing the Project|Compile command, except that Delphi runs your application immediately if the compile operation succeeds.

Executing a project from Windows

Because the compiler always creates a fully compiled standalone executable file (.EXE), you can run your application from the Windows operating environment using the same techniques as you would for any other Windows application.

If you have specified an icon for your project, it will appear beside the file name in the Windows Explorer, in a shortcut on the desktop, on the Windows Start menu, and on the taskbar when you minimize the application while it is running.

File not found projectname.res

When you create an application, Delphi creates a project.RES file. Either you deleted (or moved) the file or it is corrupted. It is required to open the project without displaying an error. Your source file refers to the resource file which contains version info resources plus the application's main icon. Try locating the file and placing in the same directory as the rest of the project files. If you cannot locate the RES file, delete all references in your source to the resource file (or create a dummy RES file), then try recompiling the project.

Database menu

The Database menu commands enable you to create, modify, track, and view databases.

- Explore
- SQL Monitor
- Form wizard

Database|Explore

Choose Database|Explore to open the Database Explorer or SQL Explorer, depending on your version of Delphi. Both tools let you create, view, and edit data and BDE aliases. In addition, the SQL Explorer lets you query local and remote databases.

Database|SQL Monitor

Choose Database|SQL Monitor to open the SQL Monitor.

This tool, available only in some versions of Delphi, lets you monitor SQL resource allocation and see the actual statement calls made through SQL Links to a remote server or through the ODBC socket to an ODBC data source.

Database|Form Wizard

See also

Choose Database|Form Wizard to use the Database Form wizard to create a form that displays data from a local or remote database.

Using the Database Form wizard

See also

Use the Database Form wizard to easily generate a form that displays data from any database that has a valid BDE alias.

Select the type of database forms to create:

- Simple database form
- Master/detail form

Select the DataSet option:

- TTableObject
- TQueryObject

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

Creating a form using the Form wizard

You can use the Form wizard to create a simple database form.

To build a database form by using the Form wizard:

1. Open the Form wizard by choosing Database|Form Wizard.
2. Select a Form Option.
3. Select a DataSet and click Next.
4. From the Drive or Alias Name list, select an alias.

   Note: If you have not created an alias, you can still enter a local database name by specifying the path to a database in the Form Wizard dialog box.
5. 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. To change the field's position in the list, choose the Up or Down button.

   For the purposes of this exercise, use all the fields from the Available Fields list. Choose Next to proceed.
3. The next Form wizard screen presents options for displaying the selected fields on the form. The wizard explains and illustrates each of your choices.

   For the purposes of this exercise, choose the Vertical option.
4. The Form wizard 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.
5. Choose the Create button to generate the form.

   The Form wizard generates a database form based on your choices.

Update SQL editor

Use the Update SQL editor to create SQL statements for updating a dataset.

The TUpdateSQL object must be associated with a TQuery object by setting the TQuery property UpdateObject to the name of the TUpdateSQL object used to contain the SQL statements. A datasource, and database name must be selected for the TQuery object. In addition, the SQL property must include an SQL statement defining a table.

To open the SQL editor:

1. Select the TUpdateSQL object in the form.
2. Right-click and choose 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	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.
Key Fields	The Key Fields list box 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.
Update Fields	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.
Get Table Fields	Read the table fields for the table name entered and list the fields.
Dataset Defaults	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.
Select Primary Keys	Click the Primary Key button to select key fields based on the primary index for a table.
Generate SQL	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.
Quote Field Names	Check the box labeled Quote Field Names to specify that all field names in generated SQL be enclosed by quotation marks.

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.

Note: 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.

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.

Field Link designer

The Field Link Designer provides a visual way to link (join) master and detail tables.

At design time, drop a TDataSource object on the form and define a datasource. Select the TTable component and double-click the **MasterFields** property in the Object Inspector to invoke the Field Link designer.

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.

Edit menu

Edit commands keyboard shortcuts

Use the Edit menu commands to manipulate text and components at design time.

Undo/Undelete	Undoes your last action or last deletion
Redo	Reverses an undelete or 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
Flip Children	Invert the layout controls into a right-to left mirror image.
Lock Controls	Secures all components on the form in their current position
Add to interface	Define a new method, event, or property for an ActiveX component.

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 removes your changes by "stepping back" through your actions and reverting to their previous state. You can specify an undo limit on the Editor Options page of the Tools|Environment Options dialog box.

If you undo a block operation, your file appears as it was before you executed the block operation.

Note: The Undo command does not change an option setting that affects more than one window.

To undo a group of actions,

1. Choose Tools|Environment Options|Editor
2. Check Group Undo.

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.

Redo is not available for reversing the effects of the Undelete command.

Edit|Cut **or Code editor context menu**

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. (You can also right-click in the Code editor and choose Cut.)
- Components from the active 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.

Edit|Copy **or Code editor context menu**

See also

Choose Edit|Copy (or right-click in the Code editor and choose 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 **or Code editor context menu**

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, only components onto the form, and only menu items into the Menu designer.

When pasting into the Code editor window, the text is inserted at the current cursor position. You can also right-click in the Code editor and choose Paste from the context menu.

When pasting onto a 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.

Edit|Delete

See also

Choose Edit|Delete to remove the selected text or component without placing a copy on the Clipboard.

Even though you cannot paste the deleted item, you can restore it by immediately choosing Edit| 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

See also

Edit|Select All selects every item (where appropriate) in the active window.

In the Form Designer, choose Edit|Select All to select every component on the active form. When you select multiple components, only those properties which the components have in common appear in the Object Inspector.

In the Code editor, choose Edit|Select All to select all the text in the currently displayed file.

Edit|Align to Grid

See also

Choose Edit|Align to Grid to align the selected components to the closest grid point.

To select more than one component, hold down Shift while clicking each one.

You can specify the grid size on the Preferences page of the Tools|Environment Options dialog box.

You can also invoke Align to Grid by right clicking in an active form.

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 controls. For example, you cannot change the z-order of a label in relation to a button.

You can also invoke Bring to Front by right clicking in an active form.

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 controls. For example, you cannot change the z-order of a label in relation to a button.

You can also invoke Send to Back by right clicking in an active form.

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 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 Equal	Lines up the selected components equidistant from each other
Center In Window	Lines up the selected components with the center of the window

You can also invoke Align by right clicking in an active form.

Edit|Size

See also

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.

You can also invoke Size by right clicking in an active form.

Edit|Scale

See also

Choose Edit|Scale to open the Scale dialog box. You can also invoke Scale by right clicking in an active form.

Scale dialog box

Use this dialog box to proportionally resize all the components on the current form.

Scaling Factor, In Percent

Enter a percentage to which you want to resize the form's contents. The scaling factor must be between 25 and 400.

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. You can also invoke Tab Order dialog box by right clicking in an active form.

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.

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. The default tab order is determined by the order in which you placed the components on the form.
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.

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 arrow to move it 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.

Edit|Creation Order

See also

Choose Edit|Creation Order to open the Creation Order dialog box. You can also invoke the Creation Order dialog box by right clicking in an active form.

Creation Order dialog box

Use this dialog box to specify the order in which your application creates nonvisual components. when you load the form at design time or runtime.

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 arrow 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.

Edit|Flip Children

See also

Edit|Flip Children allows you to reverse the layout of controls in the current form to a right-to-left mirror image. This lets developers quickly change a form created for an audience that reads left to right so that it appears natural in environments where users read from right to left.

All	Reverses the position of all children of the form. Also flips the alignment of any controls aligned to the left or right of the form.
Selected	Reverses the position of all children of the selected controls. Also flips the alignment of any controls aligned to the left or right of the selected controls.

Edit|Lock Controls

See also

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 control. However, you can use the Object Inspector to edit the Height, Left, Top, and Width properties for a selected control.

When this command is checked, controls are locked. When controls 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|Add to Interface

See also

Choose Edit|Add to Interface to define a new procedure, function, or property for an ActiveX component. These elements will be added to the ActiveX component's interfaces, making them available to other applications. This command is a short-cut for declaring an interface member to be used by an ActiveX component.

Alternatively, you can right-click in an ActiveX implementation file and choose Add to Interface from the context menu.

The Edit|Add to Interface command displays the Add to Interface dialog box that lets you choose an interface member type (such as property, method, or event) and then quickly enter the declaration for the selected type.

To automatically check the syntax of what you type in the declaration box,
Click the Syntax Helper checkbox.

When you click OK, the declaration you entered is automatically stored in the three required locations:

- The current ActiveX Implementation unit
- The ActiveX Type Library (TLB file)
- The Delphi Type Library (DTL file)

Now you can simply write the actual Pascal code for the method or set the property in your implementation unit.

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.

File commands	Description
New	Opens the New Items dialog box, which contains objects that are stored in the Object Repository and wizards for creating new objects.
New Application	Creates a new project containing an empty form, a unit, and a project file.
New Form	Creates a blank form and adds it to the current project.
New Frame	Creates a blank frame and adds it to the current project.
Open	Displays the Open dialog box for loading an existing project, form, unit, or text file into the Code editor.
Open Project	Displays the Open Project dialog box for loading an existing project (.BPR or .BPK file).
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.
Print	Sends the active file to the printer.
Exit	Closes the open project and exits Delphi.

File|New

See also

The New Items dialog box provides access to the templates and the Forms, Dialogs, Projects, and Business pages of the Object Repository. The New Items dialog box contains pages for each category of template or Object Repository page.

The Object Repository contains forms, projects, and wizards. For information about including these objects in your projects, see Including objects from the Object Repository.

Including objects from the Object Repository

See also

To include an object from the Object Repository, you can:

- 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 or project wizards. Using a wizard does not add shared code; it runs a process that generates its own code.

Inheriting Items

Inheriting items is the most flexible and powerful way to use a Repository object. Inheriting lets you reuse items within the same project.

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 are 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: You can inherit forms, dialog boxes, and data modules but not project templates. This is the *only* option available for reusing items from within the same project.

Using Items Directly

You use the Using Items Directly option primarily 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. Modify these items only at runtime to avoid making changes that affect other modules.

File|New Application

See also

Choose File|New Application to create a new Delphi project group with a single application in it.

Instead of the standard blank project, you can specify a custom project template as the default project.

If a project is open when you choose File|New Project, Delphi prompts you to save any changes to the project, closes the current project group, and creates a new project group. You can use Project|Add New Project to add a new project to the current project group.

A new project consists of:

- A new project file (PROJECT1.DPR).
- A new form file (FORM1.DFM), and its associated form unit (UNIT1.PAS).

Tip: Change the names of the project and unit files to more meaningful names before continuing.

To redefine the default project:

1. Choose Tools|Repository to open the Object Repository dialog box.
2. In the Pages list box, click Projects.

   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.

File|New Form

See also

Choose File|New Form to create a default form and a new unit and add them to the project.

When you create a new form, Delphi 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 did not redefine the default form (or if you selected a blank form from the New Items dialog), the new form is titled FormXX and the new unit is UnitXX.PAS, where XX represents the form/unit number, that is, the first form is Form1, the second Form2.

To change the name of the form:
Edit the Name property from the Object Inspector.

To change the unit name:
Save the file by using File|Save File As or save the entire project by 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.

You can specify a custom form as the default form.

To redefine the default form:
1. Choose Tools|Repository to open the Object Repository dialog box.
2. In the Pages list box, click Forms or Dialogs.

   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.

File|New Frame

See also

Choose File|New Frame to create a blank frame and add it to the current project. The new frame will not appear at runtime until it is dropped onto a form.

File|Open

See also

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.

Opening a file does not 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 have only one project open at any time. If a project is open when you select File|Open, 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 type wildcards to use as filters in the Files list box.
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.
Up One Level	Click this button to move up one directory level from the current directory.
Create New Folder	Click this button to create a new subdirectory in the current directory.
List	Click this button to view a list of files and directories in the current directory.
Details	Click this button to view a list of files and directories along with time stamp, size, and attribute information.

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.

You can create a:

- 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.

If you are creating a file that you want to include in the current project, use one of the following methods:

- Choose Project|Add to Project.
- Choose View|Project, then right-click on the Project Manager window and choose Add File.

File|Open Project

See also

Choose File|Open Project to open an existing project.

If a project currently open, you are prompted to save your changes and the currently open project is closed before you open another project.

File|Reopen

See also

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. The Reopen list can contain up to five projects and ten files.

To Reopen a project or module:

1. From the File menu, choose 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 do not appear in the list.

File|Save

See also

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 are not saved. To save these files, select each file in the Code editor and choose File|Save.

File|Save As

See also

Choose File|Save As to save the active file with a different name or in a different location.

Save As dialog box

Use the Save As dialog box to change a unit's file name or to save the unit in a new location. If the file name already exists, Delphi asks if you want to replace the existing file.

Save In	Lists the current directory. Use the drop down list to select a different drive or directory.
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. 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.
Up One Level	Click this button to move up one directory level from the current directory.
Create New Folder	Click this button to create a new subdirectory in the current directory.
List	Click this button to view a list of files and directories in the current directory.
Details	Click this button to view a list of files and directories along with time stamp, size, and attribute information.

File|Save Project As

See also

Choose File|Save Project As to save the project file (.DPR file) to a new name or location. In addition to copying and/or renaming the .DPR file and associated project files, this command saves each associated file using its current location and name.

Tip: 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, use File|Save 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.

Look In	Lists the current directory. Use the drop down list to select a different drive or directory.
File Name	Enter a name for the project 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.
Up One Level	Click this button to move up one directory level from the current directory.
Create New Folder	Click this button to create a new subdirectory in the current directory.
List	Click this button to view a list of files and directories in the current directory.
Details	Click this button to view a list of files and directories along with time stamp, size, and attribute information.

File|Save All

See also

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.

   If you are saving the files for the first time, the Sae All dialog box appears with a default name for the item to be saved.

1. Type in a new file name if you do not want to use the default name.
2. Click Save.

   The Save As dialog appears again with a default name for the next item to be saved.

3. Repeat previous steps until all modules are saved.

File|Close

See also

Choose File|Close to close the active window.

Note: File|Close typically closes only a single file. However, if the file is a form, it closes the associated unit file. If the Project Manager is the active window, it closes all files in the project.

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.

You can close the entire project by choosing File|Close when the Project Manager is the active window. If you close the project file (.DPR) from the Code editor, you also close the entire project.

File|Close All

See also

Choose File|Close all to close all open files. The project file and all modules are closed.

Tip: Another way to close all files is to close the project file (with the .DPR extension) in the Code editor.

File|Use Unit

See also

Choose File|Use Unit to add an existing a unit to the uses clause of the current unit in the Code editor. This command lets you access public objects, methods, functions, and procedures in the chosen unit from the current unit.

Use Unit dialog box

Use this dialog box to make the contents of the specified unit available to the current unit.

To use a unit:

1. From the File menu, select Use Unit.

   The Use Unit dialog box appears. It displays a list of all units in the project that are not currently used by the current unit. You can only use units when they are part of the current project. If the current project contains no more units, a message box appears instead.
2. In the Use Unit list, click the name of the unit you want to add.
3. Click OK to add the unit to the current unit.

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

See also

Use this dialog box to specify any scaling options when printing a form. The scaling options depend 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 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 onscreen size. If you choose this option, your form might print on more than one page.

Setup

To display the Printer Setup dialog box, click the Setup button.

Print Selection dialog box

See also

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

See also

Changes printer options and selects a printer from a list. To display this dialog box, click Setup from the Print Selection dialog box or the Print Form dialog box.

For more information about setting printer options, see your Windows documentation.

Name	Selects a printer from the list box.
Portrait	Prints text across the narrowest side of the paper (such as, 8.5" x 11").
Landscape	Prints text along the widest side of the paper (such as, 11" x 8.5").
Paper Size	Specifies the paper size.
Paper Source	Specifies the paper tray or paper feeding method your printer uses.
Properties	Displays the Printer properties page for the currently selected printer.

File|Exit

Choose File|Exit to close the open project and then close Delphi.

If you exit before saving your changes, Delphi asks you if you want to save them.

New Items dialog box

See also

Use the New Items dialog box to select a form, project template, or wizard that you can use as a starting point for your application. The New Items dialog box provides a view into the Object Repository. The Object Repository contains forms, projects, and wizards. You can use the objects directly, copy them into your projects, or inherit items from existing objects.

The New Items dialog box tabs

Each tabbed page in the New Items dialog box contains items that you can include in your project. Four of these pages are fixed, providing standard Delphi components:

- New
- ActiveX
- Multitier
- Your project

The remaining pages are user-defined pages containing forms, projects, data modules, or wizards from the Object Repository. You can create your own objects and store them in these pages to use as templates for your projects.

When shipped, the New Items dialog box contains these additional pages:

- Forms
- Dialogs
- Data Modules
- Projects
- Business

Right-click the page and choose View Details to read a description of each item.

To add a new form from the New Items dialog box:

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. Or select the Application wizard to define a custom template.
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. Or the Application wizard will prompt you to enter a directory name.

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 context menu.

   The item description appears in the Description column.

To edit, add pages to, or rename items in the Object Repository:

Right-click the New Items dialog box to display the speed menu. The speed menu allows you to customize the display.

Note: To open the Object Repository dialog box, select **Properties** from the context menu. You can use this dialog box to edit, add pages to, and rename items in the Object Repository.

Usage Options

There are three ways to include a Repository Object in your project. The options are dimmed if unavailable for a specific object.

- Copy the item
- Inherit from the item
- Use the item directly

For more information about usage options, see Including objects from the Object 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 File|New, select the Projects tab, and then select a non-blank project template.

Directory Name	Displays the current directory. If you enter a directory that does not exist, Delphi creates 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.

New page

See also

The New page of the New Items dialog box contains many pre-built components that you can use in your application development.

New Item	Description
Application	Creates a new project containing a form, a unit , and a .DPR, or provides a way for you to select a template.
Batch file	Creates a new batch file project with a .BAT extension that allows you to specify batch commands. The project is not associated with any forms or code editor.
Component	Creates a new component using the Component wizard.
Console Wizard	Creates a new console application project.
Control Panel Application	Creates a new applet for the Windows Control Panel.
Control Panel Module	Creates a new module for a control panel application.
Data Module	Creates a new data module that can be used as a repository for nonvisual components and business rules.
DLL	Creates a new DLL project.
Form	Creates and adds a blank form to the current project, or lets you select a form template.
Frame	Creates a new frame.
Package	Creates a new package. The new package appears in the Package editor.
Project Group	Creates a new Project Group to contain related projects. By adding associated projects to a Project Group, you can build all the projects with one command. The Project Group has the extension .BPG.
Report	Creates a Quick report that helps you create visually design effective reports for your database applications. **Note:** you can also use the QuickReport wizard on the Business page.
Resource DLL wizard	Starts up a wizard to help you generate resource DLLs that contain localized versions of your application forms.
Service	Adds a new service to an existing NT service application. Do not add services to an application that is not a service application. While a TService object can be added, the application will not generate the requisite events or make the appropriate Windows calls on behalf of the service.
Service Application	Creates a new NT service application. Once you have created a service application, You will see a window in the designer that corresponds to a service (TService). Implement the service by setting its properties and event handlers in the Object Inspector.
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.
Web Server Application	Creates a new Web server application DLL or EXE.

New Thread Object dialog box

See also

Use the New Thread Object dialog box to define a thread class that encapsulates a single execution thread in a multi-threaded application. Type the name of the class you wish to define in the Class Name edit control. Click OK to create a new unit that defines a thread class with the name supplied in the dialog. You must then supply the code that executes when the thread is run by writing the Execute method in the **implementation** section of the new unit.

Note: Unlike some other dialogs in Delphi, the New Thread Object dialog does not prepend a T to the supplied class name. You will want to type a class name such as TMyThread, rather than typing MyThread and expecting the T to be added implicitly.

To bring up the New Thread Object dialog box:

1. Choose File|New to open the New Items dialog box.
2. Choose the tab labeled New.
3. Select the Thread Object item in the list view.

Batch file projects

With batch file, you can create a project with the .BAT extension to run batch files. Right-click on the batch file project and choose Edit/Options to enter the commands in your batch file project and to indicate how commands are invoked.

The Command Execution radio buttons indicate how the batch file is executed.

Use Command Interpreter
: Delphi invokes the command-line interpreter as specified. Typically, this is $(COMSPEC), which evaluates to the command-line interpreter defined in the environment variable (such as windows\command.com or 4dos\4dos.com). You can specify another command interpreter by typing its name in the edit box or browsing for it by clicking on the ellipsis button.

Use Windows Shell
: Delphi executes each command in the batch file (using CreateProcess). It waits until each command terminates before executing the next, therefore, it handles executable programs only.

Note: If you have specified that the batch file uses the command-line interpreter, Delphi adds a line to the top of the file, “REM CommandInterpreter: $(COMSPEC),” to tell the IDE to start the command-line interpreter upon invoking this batch file. Do not remove this line from the batch file.

Note: When using a command interpreter, Delphi automatically passes a “/c” command-line parameter to the command interpreter. This switch is valid for most popular shells. If, however, your shell requires a different switch, specify that switch in the Windows registry. Under HKEY_CURRENT_USER\Software\Borland\Delphi\5.0\Compiling, add a string key called “InterpreterOptions,” and set the value of that key to the switches you want passed to the command interpreter.

Note: If your batch file requires a command-line interpreter, you must set this option before invoking the batch file.

To create a new batch file target,

1. Choose File|New and select the Batch file icon from the New page.

   Delphi creates a new project with no source code editor.
2. In the Project Manager, select the project, right-click and choose Edit/Options. The Batch file options dialog box appears.
3. In the Commands edit box, type the commands to include in the batch file.
4. Choose the method for invoking the batch file commands.

   If you want to specify a different command interpreter, type its name in the edit box.
5. Click OK.

   Delphi saves the file using the batch file extension, .BAT. It also adds code to the project group file (*project*.BPG).

To load an existing batch file,

1. Choose File|Open.
2. In Files of type, select Batch file (*.bat) to display batch files.
3. Double-click the desired .BAT file to open it.
4. Right-click and choose Edit/Options to choose the method for invoking the batch file commands.

ActiveX page

Click the objects in this page to create new COM objects, Active Forms, ActiveX controls, property pages for ActiveX controls, and type libraries for Active X controls or Automation objects.

New Item	Description
Active Server Object	Create an Active Server Page from an existing application. The Active Server Object dialog appears, where you can specify the coClass name, threading model, and so on.
Active Form	Create a new Active form, which is a simpler ActiveX control (descended from TActiveForm) preconfigured to run on a Web browser. The ActiveX Control wizard appears to guide you through the creation process, allowing you to add controls to the form. The wizard creates an ActiveX Library project (if needed), a type library, a form, an implementation unit, and a unit containing corresponding type library declarations. **Note:** Unlike other ActiveX controls, you cannot modify the properties of a built Active form in a development environment unless you add code to publish the properties.
ActiveX Control	Create a new ActiveX control. The ActiveX Control wizard guides you through the creation process, choosing the VCL object on which you want to base the new control. Note that ActiveX controls need an ActiveX library to expose their interfaces and method arguments to client applications. If an ActiveX Library project is not open before you try to create an ActiveX control, Delphi opens one.
ActiveX Library	Create a new ActiveX library. A template file named Project1.dpr is created as a starting point for you. If an ActiveX Library project is not open before you try to create an ActiveX control, Delphi opens one.
Automation Object	Create a new Automation object. The Automation Object wizard allows you to enter a class name for the new Automation object, set the type of instancing, and threading model. If an ActiveX Library project is not open before you try to create an ActiveX control, Delphi opens one.
COM Object	Create a new COM object for simple, lightweight objects such as a shell extension. The COM Object wizard allows you to specify the properties of a new COM server.
Property Page	Create a file that sets up an ActiveX property page. The property page appears in design mode, ready for you to add private and public declarations. You can design a dialog box in the form window, grouping properties to make it easy for developers to modify the control when implementing it in an application.
Type Library	Create or edit a library of type information for an ActiveX control or an Automation object. The Type Library editor appears.

Multitier page

See also

Click the objects in this page to create servers that are part of a multi-tiered application.

New Item	Description
CORBA Data Module	Create a CORBA Data Module to act as a server in a multi-tiered database application that uses CORBA as a communications protocol. The CORBA Data Module wizard appears to generate the initial implementation, defining an implementation class for the data module with the specified threading and instancing options.
CORBA Object	Create a CORBA server object. The CORBA Object wizard appears to generate the initial implementation of an interface, defining an implementation class, stub and skeleton classes, and using the specified threading and instancing options.
MTS Data Module	Create a new MTS Data Module to act as an application server in a multi-tiered database application. The MTS Data Module wizard appears to generate the initial implementation, defining an implementation class for the data module with the specified threading and transaction options.
MTS Object	Create a new MTS object. The MTS wizard allows you to create an Automation object that runs in the MTS runtime environment.
Remote Data Module	Create an application server in a COM-based multi-tiered database application. The Remote Data Module wizard appears to generate the initial implementation, defining an implementation class for the data module with the specified instancing and threading options.

Project page

If a project is open, the New Items dialog box includes a project page with the same name as the current project. The current project page contains all the forms of the project. You can create an inherited form from any existing project forms.

For information about including these forms in your projects, see Including objects from the Object Repository.

Dialog wizard

Use the Dialog wizard to design a dialog box for your application.

To display this wizard, choose File|New to display the Object Repository. Click the Dialogs tab and select Dialog wizard.

Follow the instructions in the wizard and click Next. Once you get to the last screen, click Finish. The wizard then creates the dialog box form. You can modify the form to customize it further if needed.

The wizard prompts you to select the type of dialog box (single or multi-page) and the button placement for the standard OK, Cancel, and Help buttons.

Control Panel Application wizard

See also

Control panel applications are special-purpose dynamic link libraries (DLLs) that provide a way to configure the Microsoft Windows environment. Applets on the control panel typically let users examine and modify the settings and operational modes of specific hardware and software.

To create a new control panel application:

- Choose File|New and select Control Panel Application from the New page of the New Items dialog.

Delphi creates a new applet application and a default applet module. The $E compiler directive is inserted in the project's source file, which will change the output file extension to .cpl.

Control Panel Module wizard

See also

Control panel applications are special-purpose dynamic link libraries (DLLs) that provide a way to configure the Microsoft Windows environment. When you create a control panel application, a default control panel module is created. You can use the Control Panel Module wizard to create additional modules for a control panel application.

To create a new control panel module:

- Choose File|New and select Control Panel Module from the New page of the New Items dialog.

Delphi creates a new applet module design-time form and source file.

Note: You can only add applet modules to projects that are control panel applications.

You can right-click on an applet module to display a context menu that has helpful options such as Installing a Control Panel Applet, Uninstalling a Control Panel Applet, and Launching the Control Panel.

Debugging control panel applets

See also

Control panel applications are special-purpose dynamic link libraries (DLLs) that provide a way to configure the Microsoft Windows environment. You can debug them by choosing Run|Parameters and entering the following settings:

Host Application: `c:\windows\Rundll32.exe`
or
`c:\winnt\system32\rundll32.exe`

Run Parameters: `shell32.dll,Control_RunDLL <AppletName>`

where <AppletName> is a fully-qualified path name for the applet application (including .cpl extension). Note that on Windows 95 or Windows 98, the value of <AppletName> may need to be surrounded by quotes.

Because Applet applications are really a special type of DLL, you can also write your own driver application that calls the cplApplet() function from the .cpl file. The driver application can use this function to send messages to the applet modules (for example to test specific event handlers).

Help menu

Use the Help menu to access the online Help system, which is displayed in a special Help window.

The Help system provides information on all aspects of the Delphi environment and libraries, the Object Pascal language, and so on.

Help options	Description
Delphi Help	Opens the Delphi Help Topics dialog to the tab (Contents, Index or Find) that you last used or viewed.
Delphi Tools	Opens the Help Topics dialog for the Delphi Productivity Tools Help file. The dialog opens to the tab (Contents, Index or Find) that you last used or viewed.
Windows API/SDK Help	Opens the Help Topics dialog for the Windows Programmer's Reference Help system. The dialog opens to the tab (Contents, Index or Find) that you last used or viewed.
Borland Home Page	Opens your Web browser and points it to Borland’s World Wide Web site.
Delphi Home Page	Opens your Web browser and points it to the Delphi Web page where you can find information about Delphi, including news and announcements, feature descriptions, and product downloads.
Delphi Developer Support	A direct link to the Developer support page on Borland’s World Wide Web site. Provides information and technical support, the most recent downloads for Delphi, and other services.
Delphi Direct	A direct link to the Delphi web page where you can find out more about downloading software that will automatically inform you of Delphi and Inprise news and announcements.
Customize	Launches OpenHelp, a utility that lets you configure which help topics you want available in the help contents and index.
About	Shows copyright and version information for Delphi.

Help|Delphi Help

Choose Help|Contents 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.
- You can also start a keyword search in the Code editor: Place the insertion point on or next to a term (such as a class, function, member, or property) or highlight one or more terms and press F1.

Help|Delphi Tools

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.
- You can also start a keyword search in the Code editor: Place the insertion point on or next to a term (such as a class, function, member, or property) or highlight one or more terms and press F1.

Help|Windows API/SDK Help

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.
- You can also start a keyword search in the Code editor: Place the insertion point on or next to a term (such as a class, function, member, or property) or highlight one or more terms and press F1.

Help|Borland Home Page

Choose Help|Borland Home Page to open your web browser and point it to Borland's World Wide Web page.

Help|Delphi Home Page

Choose Help|Delphi Home Page to open your web browser and display Borland's Delphi home page.

Help|Developer Support

Choose Help|Developer Support to open your Web browser and point it to Delphi's Developer Support Web page. It provides links for downloads, bug lists, frequently asked questions (FAQ's), logging bugs and suggestions, and so on.

Help|Delphi Direct

A direct link to the Delphi web page where you can find out more about downloading software that will automatically inform you of Delphi and Borland news and announcements.

Help|Customize

Choose Help|Customize to launch the OpenHelp utility.

With OpenHelp, you can add or remove help files from Delphi. This allows you to limit the number of files listed in the Contents and Index.

By default, Delphi is configured with all the help files contained in its help directory.

- If you configure OpenHelp with additional help files, you may exceed the capacity of the Index.

Help|About

Choose Help|About to display the About Delphi dialog box that shows copyright and version information.

Project menu

Use the Project menu to compile or build your application. You must have a project open.

Add to Project	Add a file to a project.
Remove from Project	Remove a file from a project.
Import Type Library	Import a type library to a project.
Add to Repository	Add a project to the Object Repository.
View Source	Display the project file in the Code editor.
Languages	Lets you add, remove, and update resource DLLs, or select a language for testing.
Add New Project	Open the New Items dialog box, which contains wizards and objects that are stored in the Object Repository. You can either generate a new object or start with any preexisting object stored in the Object Repository.
Add Existing Project	Use the Open Project dialog box to add an existing project to the project manager.
Compile project	Compile only those files in the current project that have changed since it was last built.
Build project	Compile everything in the project, regardless of whether any source has changed.
Syntax Check project	Compiles your project but does not link it.
Information for project	Displays all the build information and build status for your project.
Compile All Projects	Compile any source code that has changed since the last compile in all projects in the project group.
Build All Projects	Compile everything in the project group, regardless of whether the source has changed.
Web Deployment Options	Make necessary settings to deploy a finished ActiveX control or ActiveForm.
Web Deploy	After setting the web deployment options and compiling the project, deploy your finished ActiveX control or ActiveForm.
Options	Display the Project Options dialog box, where you set options for compiling, linking, default forms, version information, and so forth.

Project|Add to Project

See also

Choose Project|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.

Look In	Select the drive or directory in which to look for files to add to the project. The File window displays the subdirectories of the current drive or directory, as well as all the files that match the filter specified in the Files of Type combo box.
File Window	Displays the subdirectories of the current directory as well as the files in the current directory that match the current filter. Select a file to add to the project by clicking with the mouse or using the arrow keys. Right-click on the files to bring up a context menu.
File Name	Enter the name of the file you want to add to the project, or a filter expression to limit the files that appear in the file window.
Files of Type	See "Look In" above.
Up One Level	Click this button to move up a level from the current directory.
Create New Folder	Click this button to create a new subdirectory in the current directory.
List	Click this button to display the files as a list of files with associated icons.
Details	Click this button to display files in a tabular format that includes detailed information about each file's size, type, time stamp, and other attributes.

Project|Remove from Project

See also

Choose File|Remove from Project remove a unit from the current project.

Remove From Project dialog box

Use this dialog box to select one or more units to remove from the current project. When you select a unit and click OK, Delphi removes the selected unit from the **uses** clause of the current project file but does not delete any files from your disk. If you have manually-coded references to the unit in your source code, you must manually remove those references. Use the *Ctrl* or *Shift* key to select more than one unit.

If you have modified the file you are removing during this editing session, Delphi prompts you to save your changes, to preserve the unit in another project. If you have not modified the file, Delphi removes that file from the project without prompting you.

Caution: Remove the file from your project first before deleting the file from disk so that Delphi can update project file accordingly.

Project|Import Type Library

See also

The Import Type Library dialog box displays the type libraries registered on your system so you can add them to your projects. If the registered type libraries contain creatable coclasses, this dialog allows you to install VCL components representing them on the component palette. You can generate Pascal declarations in a .PAS file that let you use these types as though they were native VCL objects.

The top part of the dialog is a list of type libraries that are currently registered and thus available to be imported. This list lets you extract the declarations from an existing control or free-standing type library. You can also conveniently register a new type library from this dialog box so that it is available to be imported.

To add and register a new type library:

1. Click Add. The Register OLE Control dialog box appears.
2. In the Register OLE Control dialog box, navigate to the disk or network location of the library file you want to add.
3. Select the new type library. It is automatically registered on your system for Delphi and immediately appears in the list of available libraries in the Import Type Library dialog.

Add button	Locate a new type library and register it in the Windows Registry, so that it will appear in the list of registered objects available to be imported into Delphi.
Remove button	Remove a registered type library. The library is removed from the Windows Registry and from this list. Warning: Removing type libraries can disable the associated applications.
Class names	Shows all creatable coclasses in the selected type library.
Palette page	Determines on which page of the component palette to add the component(s) listed under Class names when you click Install.
Unit dir name	Shows the name of the directory that contains the unit using this library. Only the path root is shown; no file name appears. The unit name is derived from the internal type library name. Click the Browse button to move up the directory tree.
Search path	Specifies where to look for dependencies when creating a package.
Install button	Creates a new file and adds it to a new or existing package. When you choose Install, an Install dialog lets you specify a new or existing package to be created and installed. This button is grayed out if no component can be created for that type library.
Create Unit button	Creates a file for the type library and adds it to your project.

The Import type library is one way to create an Automation controller.

Project|Add to Repository

See also

Choose Project|Add to Repository to open the Add to Repository dialog box. Use this command to add projects and forms to the Object Repository.

By adding your own projects and forms to those already available in the Object Repository, you can share objects across your organization. This is helpful in situations where you want to enforce a standard framework for programming projects.

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 icon.

Dialog box options

Title	Enter the name of the template. This is the full path of the object you are adding. 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) on which you want the template to appear.
Author	Enter text identifying the author of the application. Author information appears only when you select View Details form the context menu.
Template Icon	Click the Browse button to open the Select Icon dialog. You can use a bitmap of any size, but it will be cropped to 60 x 40 pixels.

Project|View Source

See also

Use Project|View Source to display the project file for the current project and make it the active page in the Code editor. If the project source file is not currently open when you choose this command, Delphi opens it for you.

Project|Languages

See also

Choose Project|Languages and one of the following options to change translation (resource DLL) settings for your project.

- Add
- Remove
- Set Active
- Update Resource DLLs

Project|Languages|Add

See also

Choose Project|Languages|Add to add a resource DLL.

Project|Languages|Remove

See also

Choose Project|Languages|Remove to remove a resource DLL.

Project|Languages|Set Active

See also

Choose Project|Languages|Set Active to select a resource DLL for testing.

Project|Languages|Update Resource DLLs

See also

Choose Project|Languages|Update Resource DLLs to update the resource DLLs associated with your project.

Project|Add New Project

Use Project|Add New Project to add a new item (such as a new application, DLL, or package) to the project group. This command opens the New Items dialog box to create a new target from the templates provided in the Object Repository.

Typically, a project group consists of at least one project, which contains the source of your application. You can choose to add additional projects to a project group to contain other targets associated with your application. For example, you may have one project for your .EXE, another for your .DLL, and another for your application resources.

This command works the same as in the Project Manager, selecting a project group, right clicking, and choosing Add New Project.

For details on project groups, see creating a project group.

Project|Add Existing Project

Choose Project|Add Existing Project to add an existing project to the current project group. This command opens the Open dialog box for you to specify the path to the project that you want to add to this project group.

Typically, a project group consists of at least one project, which contains the source of your application. You can choose to add additional projects to a project group to contain other targets associated with your application. For example, you may have one project for your .EXE, another for your .DLL, and another for your application resources.

This command works the same as in the Project Manager, selecting a project group, right clicking, and choosing Project|Add Existing Project.

For details on project groups, see creating a project group.

Project|Compile project

See also

Use Project|Compile project to compile all files in the current project that have changed since the last build into a new executable file (.EXE),. dynamic link library (.DLL), resource file (.RES), or so on. This command is similar to the Build command, except that Project|Compile builds only those files that have changed since the last compile, whereas Build rebuilds all files whether they have changed or not.

If you checked Show Compiler Progress from the Preferences page on the Tools|Environment Options 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 cannot 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 may choose to compile only portions of your code if you use conditional directives and predefined symbols in your code.

Project|Build project

See also

Choose Project|Build *project* to rebuild all the components of a project regardless of whether they have changed. This command is useful when you've changed global compiler directives or compiler options, to ensure that all code compiles in the proper state.

This option is identical to Project|Compile project except that it rebuilds everything, whereas Project|Compile rebuilds only those files that have changed since the last build.

You can also invoke this command from the Project Manager. Right click and choose Build.

If you have multiple projects within a Project Group, you can build all projects within a Project Group by using the Project|Build All Projects command.

Project|Syntax check Project

Choose Project|Syntax check Project 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.

Using Project|Syntax check Project is faster than using Project|Compile project because Delphi does not have to create the object code for the units.

Project|Information for project

Choose Project|Information for project 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
File size	Size of final output file

Status Information

The Status Information line displays whether or not your last compile succeeded or failed.

Package Used

The Package Used group lists all runtime packages included in the project. You can add runtime packages to the project using the Packages page of the Project Options dialog.

Project|Compile All Projects

See also

Use Project|Compile All Projects to compile all files in the current project group that have changed since the last build. This command is similar to Project|Build All Projects, except that Project|Compile All Projects builds only those files that have changed since the last compile, whereas Project|Build All Projects rebuilds all files.

The Project|Compile All Projects command recompiles changed files from top to bottom as they are listed in the Project Manager. (For example, if a project group includes a DLL on which the executable file depends, list the DLL first to compile most effectively.)

If you checked Show Compiler Progress from the Preferences page of the Tools|Environment Options 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 cannot 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 may choose to compile only portions of your code if you use conditional directives and predefined symbols in your code.

Project|Build All Projects

See also

Choose Project|Build All Projects to rebuild all the projects in your Project Group regardless of whether they have changed. This command is useful when you've changed global compiler directives or compiler options, to ensure that all code compiles in the proper state.

This option is identical to Project|Compile All Projects except that it rebuilds everything, whereas Project|Compile All Projects rebuilds only those files that have changed since the last build.

The Project|Build All Projects command recompiles all files included in the project group from top to bottom as they are listed in the Project Manager. Be sure to list projects in the order you want them compiled. You can reorder projects within a project group by right-clicking in the Project Manager with a project selected and choosing Build Sooner or Build Later.

To build a single project within a Project Group, choose Project|Build.

Project|Web Deployment Options

Choose Project| Web Deployment Options to configure a finished ActiveX control or ActiveForm for deployment to your Web server. First, set web deployment options, compile the project, then choose Project|Web Deploy to deploy the current ActiveX project.

See Deploying ActiveX controls or ActiveForms on the Web

Default checkbox If checked, saves the current settings from the dialog box's Project, Packages, and Additional Files pages as the default options. To restore the default properties to the original state, delete or rename the DEFPROJ.DOF file.

The Web Deployment Options dialog box contains four tabbed pages of settings:

Project page

Specifies locations of files and the URL. Also, allows you to set configuration settings for CAB file compression.

Packages Page Specifies packages used by this project

Additional Files Page Specifies other files associated with this project.

Project|Web Deploy

Choose Project|Web Deploy to deploy a finished ActiveX control or ActiveForm to your web server. Use this command only after setting the Web Deployment Options and then compiling your project.

The ActiveX library (.OCX) is placed in the target directory specified in the Web deployment options. An HTML file (.HTM) that contains a URL reference to the ActiveX library in the target directory is created in the HTML directory specified in the Web Deployment options. When this HTML file is viewed in a Web browser, your ActiveX control or ActiveForm runs as an embedded application within the browser.

See also Deploying ActiveX controls or ActiveForms on the Web.

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. You can change the options of the current project, or the default properties for new projects. If there is no project currently open, you can only change the default properties.

The pages of the Project Options dialog box are:

Forms	Controls which forms are created automatically
Application	Specifies the title, help file name, and icon name associated with the application
Compiler	Specifies compiler switches that detemine how code is compiled
Linker	Manages how your program files are linked
Directories/Conditionals	Specifies the location of files needed to compile and link your program
VersionInfo	Specifies the types of product identification information
Packages	Specifies the design-time and runtime packages to install for your project

Tabs

You can change the page displayed by clicking the tabs at the top of the dialog box.

Default check box	saves the current settings as the default for each new project.

Default checkbox **(Project|Options) or (Project|Web Deployment Options)**

See also

Both the Project Options and the Web Deployment Options dialog boxes have a checkbox in the lower left- corner labeled Default. Checking this box saves the settings selected in the dialog as the default settings for every new project you create.

This check box is disabled if there is no current project open, because in that case you can only change the default properties.

To restore the default properties to the original state, delete or rename the DEFPROJ.DOF file.

Forms (Project|Options)

See also

Use the Forms page of the Project Options dialog box to select the main form for your current project and to choose which of the available forms are automatically created when your application begins.

Main form	Displays the form users 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 created at runtime. These forms are automatically created and displayed when you first run your application. 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 one of these forms, you must call its Create method.
Arrow buttons	Use the arrow buttons to move 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 box

Check Default to save the current settings in all the Project Options pages as the default options. Delphi will use the default options for each new project you create.

Application (Project|Options)

See also

Use the Application page of the Project Options dialog box to specify a title, a Help file, an icon, and an extension for your application.

Application settings

Title Specify a title to 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 automatically calls when invoking 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. To change the icon, click Load Icon and Delphi displays the Application Icon dialog box, where you can select an icon.

Output settings

Target file extension Specify the file extension to be used for the target executable file. If the project is an ActiveX application or DLL the standard file extension can be specified, such as .ocx for an ActiveX file.

Default check box

Check Default to save the current settings in all the Project Options pages as the default options. Every new project will use the default options.

Application Icon dialog box

Use the Application Icon dialog box to select an icon that represents your application in the Program Manager and when your application is minimized.

To display this dialog box:

Select Project|Options, select the Application page of the Project Options dialog box, and click Load Icon.

File Name	Enter the name of the file you want to use, or enter or wildcards to use as filters in the Files window.
Files	Displays subdirectories of the current directory as well as all files in the current directory that match the filter in the File Name edit box or the file type in the Files of Type combo box. Select an icon file for your application's icon using the mouse or arrow keys. Right-click on a file to bring up a context menu.
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.
Look in	Select the drive or directory in which to look for the icon file. The Files window displays the subdirectories of the current drive or directory as well as all files that match the current filter.

Application Help File dialog box

Use the Application Help File dialog box to select a Help file (.HLP) 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:
Select Project|Options, select the Application page and click Browse

File Name	Enter the name of the file you want to use, or enter or wildcards to use as filters in the Files window.
Files Window	Displays all subdirectories in the current directory as well as files in the current directory that match the filter in the File Name edit box or the file type in the Files Of Type combo box. Select a Help file for your application using the mouse or arrow keys. Right-click on a file to bring up a context menu.
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.
Look In	Select the drive or directory in which to look for the Help file. The Files window displays the subdirectories of the current drive or directory as well as all files that match the current filter.

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}.
Stack frames	Forces compiler to generate stack frames on all procedures and functions. Corresponds to {$W}.
Pentium-safe FDIV	Generates code that detects a faulty floating-point division instruction. Corresponds to {$U}.

Runtime errors	Effect
Range checking	Checks that array and string subscripts are within bounds. Corresponds to {$R}.
I/O checking	Checks for I/O errors after every I/O call. Corresponds to {$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 compatibility with Delphi 1.0. When enabled, the compiler allows assignments to typed constants. Corresponds to {$J}.

Debugging	Effect
Debug information	Puts debug information into the unit (.DCU) file. Corresponds to {$D}.
Local symbols	Generates local symbol information. Corresponds to {$L}..
Reference info/Definitions only	Generates symbol reference information used by the Code Browser, Code Explorer, and Project Browser.. Corresponds to {$Y}. If Reference Info and Definitions Only are both checked (**{$YD}**), the compiler records information about where identifiers are defined. If Reference Info is checked but Definitions Only is unchecked (**{$Y+}**), the compiler records information about where each identifier is defined and where it is used. These options have no effect unless Debug

	Information and Local Symbols (see above) are selected.
Assertions	Generates code for assertions placed in code. Corresponds to {$C}. Unlike exceptions, assertions can be removed for the final build. After disabling the option, rebuild the code base to eliminate assertions.
Use Debug DCUs	Allows you to link in debug versions of the VCL. When checked, Delphi prepends the Debug DCU path (specified in Tools|Debugger Options on the General page) to the unit Search path specified in Project|Options ont he Directories/Conditionals page.

Messages	**Effect**
Show Hints	Causes the compiler to generate hint messages.
Show Warnings	Causes the compiler to generate warning messages.

Default check box

Check Default to save the current project options so that every new project you create will use those options.

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

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.

- Default = Off

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

Specify the output from the linker.

Option	Effect
Generate DCUs	Output standard Delphi DCU format files.
Generate C object files	Create a C object file for linking with a C program (no name mangling).
Generate C++ object files	Create a C++ object file for linking with C++Builder (uses C++ name mangling).
Include namespaces	Puts information into namespaces and mangles the namespace (the name of the unit) into the symbol. This must be checked if sharing code with C++Builder. (This option is enabled if Generate C++Object files is checked.)
Export all symbols	Creates DLL exports for exported functions in the project. It is for use when generating OBJs for packages that will be linked into C++Builder applications. (This option is enabled if Generate C++Object files is checked.)

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 TD32 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. Use this option only if you are using an external debugger. Use of this option causes an increase in the length of time required to compile a project.
Include remote debug symbols	Check this if you are using remote debugging.

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 committed size of the stack (only applicable to .EXE projects – disabled for DLLs)
Max stack size	Total reserved size of the stack (only applicable to .EXE projects – disabled for DLLs)
Image base	Specifies the preferred load address of the compiled image. This value is typically only changed when compiling to a DLL.

Description

EXE Descriptor

This field can contain a string of up to 255 characters. The string will be linked to $D and included in the executable file. It is most often used to insert copyright information into the application. Copyright information can also be included as part of the VersionInfo file.

Default

Check Default to save the current project options so that every new project you create will use those options.

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. In addition, you can specify compiler defines on this page. Click the down arrow next to any edit box to choose from a list of previously entered directories or symbols.

Directories

Output directory	Specifies where the compiler should put the compiled units and the executable file.
Unit output directory	Specifies a separate directory to contain the .dcu files. Note, .DCP files can be relocated by setting the DCP output directory path on the library page of the Tools|Environment Options dialog box.
Search path	Specifies the location of your source files. 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. If you check Use Debug DCUs on the Compiler page of the Project|Options, the Debug DCU path (Tools|Debugger options|General page) is prepended to this search path.
Debug source path	Search path for the debugger. The debugger searches paths defined by the compiler by default. If the directory structure has changed since the last compile, a path can be entered here to include a file in the debugging session.
BPL output directory	Specifies where the compiler puts generated package files (BPL files).
DCP output directory	Specifies where your DCP file is placed at compilation time. If left blank, the global DCP output directory specified in the Tools|Environment Options Library page is used instead.

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.
- Relative and absolute path names are allowed, including path names relative to the logged position in drives other than the current one.

Conditionals

Conditional defines	Symbols referenced in conditional compiler directives. You can separate multiple defines with semicolons.

Aliases

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`.

Default

Check to save the current project options so that every new project you create will use those options.

List entry dialogs

These dialog boxes help you manage what can be lengthy lists of paths, conditionals, names, or unit aliases. They appear when you click the ellipses next to a listing edit control on another dialog such as the Directories/Conditionals page of the Project Options dialog.

Use these dialogs to add and remove strings from a semicolon-delimited list.

The list appears in the listbox at the top. As you scroll through the list, an edit control beneath the list displays the selected entry. Use this edit control to modify the current entry or enter a new entry. After editing the value in the edit control, click Replace to modify the current entry or Add to add the value as a new entry to the list. Click the Delete button to remove the current entry from the list.

Use the arrow buttons at the right side of the dialog to rearrange the entries in the list.

Version Info **(Project|Options)**

See also

Use this page to enable the version information option and to specify version information for the project.

Include version information in project determines whether the user can view product identification information.

Module version number sets hierarchical nested version, release, and build identification.

Module attributes indicates the intent of this version: whether for debugging, pre-release, or other purposes.

Language indicates the natural language (locale) the application displays.

Key/Value list box sets typical product identification properties.

Default check box

Check Default to save the current settings in all the Project Options pages as the default options. Each new project begins with these default options.

Include version information (Version info options)

See also

The include version information in projects checkbox enables version information to be entered. This information is then included in the compiled code. When version information is included, a user can right-click the program icon and select properties to display the version information.

Module version number (Version info options)

See also

Major, Minor, Release, and Build each specify an unsigned integer between 0 and 65,535. The combined string defines a version number for the application, for example 2.1.3.5.

Check Auto-increment build number to have the build number incremented each time the Project | Build <Project> menu is selected. (Other compilations do not change the build number.)

Module attributes (Version info options)

See also

Module attributes are flags that can be included in the version information for informational use only. If a project is compiled in debug mode, the debug flag will be included in the version information. You can select each of the remaining flags as needed.

Attribute	Effect
Debug build	Included to indicate that the project was compiled in debug mode.
Pre-release	Include to indicate the version is not the commercially released product.
DLL	Include to indicate that the project includes a dynamic-link library.
Special build	Include to indicate that the version is a variation of the standard release.
Private build	Include to indicate that the version was not built using standard release procedures.

Key/Value list box (Version info options)

See also

The Key/Value list box options provide appropriate values for typical product identification attributes. A default set of keys are included.

Key	Value indicates
CompanyName	The company that produced the file. Required.
FileDescription	File description. You can display this string in a list box during installation. Required
FileVersion	File version number. Required.
InternalName	File internal name. If file does not have internal name, use original filename, without extension. Required.
LegalCopyright	File copyright notices. Optional.
LegalTrademarks	Trademarks and registered trademarks that apply to file. Optional.
OriginalFilename	Original file name, not including path. Required.
ProductName	Name of product that file is distributed with. Required.
ProductVersion	Version of product that file is distributed with. Required.
Comments	Additional information for diagnostic purposes. Optional.

Key entries can be edited by selecting the key and re-entering the name. Key entries can be added by right-clicking within the Key/Value table and selecting Add Key.

Language (Version info options)

See also

The Language indicates which Code Page the users system will require to run the application, that is, it indicates which language the application displays. Choose the desired language from the drop-down list. The hex value of the selected locale appears above the drop-down box.

Note: You can only choose a language that is listed in the Control Panel Regional Settings dialog of your computer. Windows 95, Windows 98, and Windows NT do not include support for some languages (such as Far Eastern languages) and you may need to install the appropriate Language Pack before you can specify some languages.

Packages page (Project|Options, Component|Install packages)

See also

Use this page to specify the design-time packages installed in the IDE and the runtime packages required by your project.

Design packages	Lists the design time packages available to the IDE and to all projects.
Runtime packages	Determines which run-time packages to use when the executable file is created.

Default

Check the "Default" box in the lower left corner of the dialog to turn the current package configuration into the default configuration for all new projects.

Design packages (Packages options)

Design packages lists the design time packages available to the IDE. Items with a check mark are installed in the current project. When a package is installed, it may register components that appear on the Component palette, experts that appear on the menu bar and New Items (File|New) dialog, and property editors for custom components.

Warning: Take care when uninstalling packages, whether by using the Remove button or by unchecking the package's check box. When a package is removed, any components registered by it become unavailable in the IDE. If a project contains forms that use unavailable components, you will not be able to load the forms; if this happens, reinstall the package and reload the unit.

The following buttons manipulate entries in the list.

Button	Description
Add	Installs a design time package. The package will be available in all projects.
Remove	Deletes the selected package. The package becomes unavailable in all projects.
Edit	Opens the selected package in the Package editor if source code is available.
Components	Displays a list of the components included in the selected package.

As packages are installed and uninstalled, you may notice that the runtime package list is updated. Delphi automatically adds runtime packages that are required by installed design-time packages.

Runtime packages (Packages options)

The Runtime Packages option determines which run-time packages to use when the executable file is created. A runtime package is a special dynamic-link library used by Delphi to provide functionality when a user runs an application.

Build with runtime packages check box

Check this to include runtime packages in your project and to enable the runtime packages edit box.

Runtime packages edit box

Use this to change the packages included in your project. you can add a package to your project by

- Clicking Add and specifying a package to add to the list of runtime packages in the Add runtime package dialog box.

Or,

- Typing a list of packages to use as runtime packages, separated by semicolons (VCL50;VCLDB50;VCLDBX50), into the edit box.

When a project uses a package, Delphi must find the package's .DCP file in order to compile. When Delphi cannot find the .DCP, it is often because the .DCP's directory is not included in the global Library Search Path. To edit the Library Search Path, choose Tools|Environment Options and select the Library tab.

Add Design Package

See also

Use the Add Design Package dialog box to specify the name of a design time package to add to the Design packages list. This is a standard Windows Open dialog.

Add Runtime Package

See also

Use the Add Runtime Package dialog box to specify the name of a runtime package to add to the Design packages list.

Package Name Type the name of the package to add to the Runtime Packages list, or click the Browse button to search for the package using the Package File Name dialog. If the package is in the Search Path, a full path name is not required. (If the package directory is not in the Search Path, it will be added at the end.)

Search Path If you haven't included a full directory path in the Package Name edit box (see above), make sure the directory where your package resides is in this list. If you add a directory in the Search Path edit box, you are changing the global Library Search Path. You can also change this path on the Tools|Environment Options Library page.

Close this dialog box and open *package*?

When editing a package selected in the Design packages list, Delphi closes the Project Options dialog box and displays the selected package in the package editor. Click Yes to edit the package or No to return to the Project Options dialog box.

Another file with the same base name <file> is already on the search path

This error message appears when a file is found on the search path but not at the location specified by the user when creating a new component, installing a component, importing an ActiveX control, or adding a required package to a package. Because the Library Search Path is used to locate files when building a package, this conflict prevents the compiler from finding the intended file.

The problem can be solved by modifying the Search Path so that the directory of the intended file precedes the directory of the conflicting file.

Package *package* will be built then installed. Continue?

This message appears when adding a component or ActiveX control to a package. Click Yes to rebuild the package to reflect the new component or control.

Package(s) <package list> will also be uninstalled because they require package <package name>. Continue uninstall?

Appears when installed packages rely on packages that you're trying to uninstall. Click Yes to uninstall the dependent packages.

Components

See also

Lists components in the selected package, along with the icons for those components that appear on the Component palette if the package is installed.

Package editor

See also

The Package editor lists the units in a package, and the other packages it requires. You can save your changes with File|Save or Save As.

Contains list The Contains list shows the units included in the package. To add a unit to the package, click the Add button. To edit a unit's source code, double-click it.

Requires list The Requires list shows the other packages required by the current package. To add a package, click Add. To display a package in its own Package editor, double-click it.

Package editor SpeedBar

Button	Description
Compile	Compiles the current package. If changes to the package are required, a dialog box appears that lists the changes that will be made to the package before it is compiled.
Add	Adds an item (see the Contains list and Requires list for details).
Remove	Removes the selected item from the package.
Options	Displays the Project Options Dialog Box.
Install	Installs the current package as a design time package. If changes to the package are required, a dialog box appears that lists the changes that will be made to the package before it is compiled.

Package editor context menu

To display the Package editor context menu, right-click an item within either Package editor page.

Command	Description
Add	Equivalent to Add button on SpeedBar.
Remove File	Equivalent to Remove button on SpeedBar.
Open	If a unit is selected, loads the unit in the code editor. If a package is selected, opens the Opens the package in a new Package Editor window.
Save	Saves the current package.
Remove From Project	Equivalent to Remove button on SpeedBar.
View Unit	Lets you view the selected unit’s source code.
View Source	Lets you view the package source code (.DPK file)
Options	Equivalent to Options button on SpeedBar.
Install	Equivalent to Install button on SpeedBar.
Make	Check all units which the file being compiled contains and only compile those units that have been modified or any compiler directives specifies that the file be compiled.
Build	Compile all units which the package contains.
Add to Project Group	Add the package to the current project group.
Toolbar	Displays the Package Editor toolbar when checked.
Status Bar	Displays the Package Editor status bar when checked.
Dockable	Allows the Package Editor to be docked onto other windows (and vice versa) when checked.

Add Unit

Enter the name of the unit file to add in Unit File Name (or select it using Browse). If you enter a name, Delphi searches the paths specified in Search path.

Unit File Name The name of the unit to add. If the unit is in the Search Path, a full path name is not required. If the unit directory is not in the Search Path, it will be added to the end.

Search Path If you haven't included a full directory path in the Unit File Name edit box (see above), make sure the directory where the unit resides is in this list. If you add a directory in the Search Path edit box, you will be changing the global Library Search Path.

Requires

Enter the name of the package to add in Package name (or select it using Browse). If you enter a name, Delphi searches the paths specified in Search path.

Package Name Enter the name of the package to add. If the package is in the Search Path, a full path name is not required. (If the package directory is not in the Search Path, it will be added to the end.)

Search Path If you haven't included a full directory path in the Package Name edit box (see above), make sure the directory where your package resides is in this list. If you add a directory in the Search Path edit box, you will be changing the global Library Search Path.

When a package is required by another package, Delphi must find the package's .DCP file in order to compile.

Description page (Project Options dialog box)

See also

The Description page lets you specify a description for the package, the uses of the package, and how the package is built.

Description A brief description that appears when the package is installed.

Usage options Select Design Package if you want the package to be installable on the Component palette.

Select Runtime Package if you want the package to be deployable with an application.

Select both Design Package and Runtime Package if you want the package to be both installable and deployable.

If neither Design Package nor Runtime Package is checked, the package cannot be installed on the Component palette or deployed with applications. Use this option for packages that exist only to be referenced (required) by other (design-time) packages.

Note: If your package uses custom property editors, it's a good idea to compile separate design-time and runtime versions of it. The deployed runtime package will be smaller than the design-time version, since it won't contain code for the property editors.

Build control If the package is low-level and does not change often, click Explicit Rebuild. It is built only when you display it in the Package editor and click Build. For automatic compilation, click Rebuild As Needed.

Linker page (Package Options dialog box)

See also

Use the Linker page of the Project Options dialog box to specify how your program files are linked.

Memory sizes

Use this edit box to specify the heap image base for the compiled executable. You can also specify memory-size settings in your source code by using the $M compiler directive.

Option	Specifies
Image Base	Specifies the preferred load address of the compiled image.

Conditionals page **(Package Options dialog box)**

See also

Use the Conditionals page to specify compiler directives. Click the down arrow next to the edit box to choose from a list of previously entered symbols.

Conditional Defines Symbols referenced in conditional compiler directives. You can separate multiple defines with semicolons.

Library page (Tools|Environment Options)

See also

Use this page to specify directories, compiler, and linker options for all packages. Click the down arrow next to the edit boxes to choose from a list of previously entered symbols.

Map file

Select the type of map file produced, if any. The map file is placed in the directory specified by DCP Directory, 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.

Messages	Effect
Show Hints	Causes the compiler to generate hint messages.
Show Warnings	Causes the compiler to generate warning messages.

Options	Effect
Compile with debug info	Check to compile the package file with debug information. This will make the resulting file larger, but it does not affect memory requirements or performance. When you compile a package using debug information, you can use the integrated debugger or Turbo Debugger for Windows to debug the library file.

Directories	
BPL output directory	Where the compiler should put the compiled package (.BPL) file.
DCP output directory	Specifies a separate directory to contain the .dcp files.
Library path	Specify search paths where compiler can find the source files for the package. The compiler can find only those files listed in Library Path. If you try to build your package with a file not on the library path, you will receive a compiler error.

Aliases	
Unit Aliases	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.

Change Package dialog box (Package editor)

See also

The Change dialog appears when the Package editor tries to compile a package and detects that the package cannot be built or is incompatible with another package currently loaded by the IDE. This occurs because the package uses a unit or units that are found in another package; select View Details to see which units are causing the problem.

The solution is to add the other package to the "requires" clause of the package you are editing, or simply not use the problem units at all. Click OK to let the Package editor make the proposed changes and continue compiling. Click Cancel to leave the package as it is.

Duplicate file name error (Package editor)

See also

This error message appears when a file in the directory of the package has the same name as a file in a different location specified by the user. Because the directory of the package .DPK file is searched first when building a package, the naming conflict prevents the compiler from finding the intended file.

This problem can arise when creating a component, installing a component, importing an ActiveX control, or adding a contained unit or required package to a package.

The problem can be solved by saving the package .DPK file to a different directory.

Run menu

The Run menu commands help you debug your program from within Delphi. The following commands form the core functionality of the integrated debugger.

Run	Compiles and executes your application
Attach to Process	Provides a list of currently running processes that you can debug
Parameters	Specifies startup parameters for your application, a host executable for DLLs, or a computer for remote debugging
Register ActiveX Server	Adds a Windows registry entry for your ActiveX control Available when the current project is an ActiveX project
Unregister ActiveX Server	Removes the project from the Windows registry. Available when the current project is an ActiveX project
Install MTS Objects	Installs MTS objects in the current project into an MTS package. Available when the current project is an MTS object
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
Run Until Return	Runs the process until execution returns from the current function
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
Inspect	Open an Inspector window, where you can enter an item you want to inspect
Evaluate/Modify	Displays the Evaluate/Modify dialog box, where you can evaluate or change the value of an existing expression
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

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.

Run|Attach to Process

Choose Run|Attach to Process to debug a process that is currently running. A list of processes running on the local computer is displayed. Select a process from the list and either double-click it or click on Attach to start debugging. The debugger is ‘attached’ to the process.

You will not be allowed to attach to a process you are already debugging, nor will you be able to attach to the IDE itself.

You can also list processes on a remote machine by entering the remote computer name in the Run|Attach to Process dialog box and clicking Refresh. The remote debug server must be running on the remote computer. For more information, see Remote debugging.

Run|Parameters

See also

Choose Run|Parameters to open the Run Parameters dialog.

Use this dialog to pass command-line parameters to your application when you run it (just as if you were starting the application from the Program Manager File|Run menu), specify a host executable for testing a DLL, run your program on a remote machine, or load any executable into the debugger.

Click OK to save the settings you've entered and close the Run Parameters dialog.

Local tab

Use this page to run and debug projects on the computer you are working at.

Host application

Enter the path to an .EXE file. (Click Browse to bring up a file-selection dialog.)

If the current project is a DLL, use this edit box to specify a host application that calls the DLL.

You can also enter the name of any executable that you want to run in the debugger. Then press Load to load the executable. The executable will be paused at its entry point. If there is no debug information at the entry point, the CPU window will be opened. Select Run|Run (F9) to run the executable.

If you want to run the project that you have open in Delphi, there is no need to enter anything in the Host Application edit box.

Parameters

Enter the command-line arguments you want to pass to your application (or the Host application) when it starts. You can use the drop-down button to choose from a history of previously specified parameters.

Do not enter the application name in this edit box.

Remote tab

Use this page to run an application on a remote computer.

Remote path

Enter the path to an .EXE file *as the remote host will see it*. If you are debugging a DLL, specify an application that calls the DLL.

You can also enter the name of any executable that you want to run in the debugger. Then press Load to load the executable. The executable will be paused at its entry point. If there is no debug information at the entry point, the CPU window will be opened. Select Run|Run (*F9*) to run the executable.

Remote host

Enter the name or IP address of the computer on which you want to run the application. The remote host must have the debug server running on it.

Parameters

Enter the parameters you want to pass to the application when it starts.

You can use these parameters with the ParamCount and ParamStr() functions.

Do not enter the application name in this edit box.

Select the "Debug project on remote machine" check box to enable remote debugging.

The Load button loads the application, meaning that the process is loaded and stopped. The OK button accepts the settings, but does not load the application. On the Remote page, the Load button loads a source view of the remote application, but does not load the project (that is, the Project menus do not reflect that project).

Run|Register ActiveX server

See also

Choose Run|Register ActiveX Server to add a Windows registry entry for your ActiveX control.

Building the project creates an .OCX file that contains the ActiveX control. You must register the control so that it can be used by other applications such as Visual Basic, C++Builder applications, other Delphi applications, or Paradox for Windows. Registering the control adds an entry for it in the Windows registry.

To register the ActiveX control from the IDE, choose Run|Register ActiveX Server.

Run|Unregister ActiveX Server

Choose Run|Unregister ActiveX Server to remove the Windows registry entry for your ActiveX control.

To remove an ActiveX control from your system, it is recommended that you first remove its entry from the Windows registry. To unregister the ActiveX control from the IDE, choose Run|Unregister ActiveX Server.

Run|Install MTS Objects

See also

Choose Run|Install MTS Objects to allow the MTS objects in your application to be run in the MTS runtime environment. Before choosing this command,

- Your project must be an MTS object.
- The Microsoft transaction server (MTS) must installed on your machine.

Install MTS Objects displays a dialog that allows you to install your components into an MTS package.

Note: Packages can contain components from multiple DLLs and components from a single DLL can be installed into different packages. However, a single component cannot be distributed among multiple packages.

Install Object Into New Package

See also

Choose Into New Package to create a new package in which to install your MTS object.

Package Name Supply a name for the new MTS package.

Description Provide a description of the MTS object.

Click OK to update the MTS catalog which makes the MTS objects available at runtime.

Install Object Into Existing Package

See also

Choose Into Existing Package to install your MTS object into an existing MTS package.

Package Name Choose the MTS package from the list.

Description Provide a description of the MTS object.

Click OK to update the MTS catalog which makes the MTS objects available at runtime.

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.

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 on the toolbar.

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 Trace Into 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 Stepping through 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 on the toolbar.

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.

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.

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 alternate way to perform this command is:

Right-click the Code editor and choose Debug|Run to Cursor.

Run|Run Until Return

Choose Run|Run Until Return to run the loaded program until execution returns from the current function. The process stops on the instruction immediately following the instruction that called the current function.

Run Until Return is only available when your process is stopped in the debugger, and can be directed at a thread anytime it is stopped. When it is issued, the thread's stack is examined and the call-site of the current function is determined. Execution then resumes and stops when the thread attempts to return to the call site of the current function.

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.

If the execution point is at a location where there is no source code, the CPU window will be opened at the execution point and show the machine instructions.

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.

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.

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 result of resetting your program, you should exit Delphi before restarting your debugging session.

Run|Inspect

See also

Choose Inspect to open an Inspector window for the term highlighted (or at the insertion point) in the Code editor. If the insertion point is on a blank space when you choose this command, an empty Inspector window displays where you can enter an item you want to inspect.

After you enter a valid expression, choosing OK opens an Inspector window.

This command is only available when the integrated debugger is paused in a program you are debugging, such as when

- you are stepping through code.
- your program is stopped at a breakpoint.
- you first choose Run|Run and then choose Run|Pause.

An alternate way to perform this command is:

Right-click, not on an expression, the Code editor and choose Debug|Inspect.

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:

Right-click in the Code editor and choose Debug|Evaluate/Modify.

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 to display and manage the current list of watches.

Alternate ways to perform this command are:

- Choose Debug|Add Watch at Cursor from the Code editor context menu.
- Choose Add Watch from the Watch List context menu.
- Right-click an existing watch in the Watch List and choose Edit Watch.

Run|Add Breakpoint

See also

Use the Run|Add Breakpoint menu commands to add breakpoints:

Source Breakpoint	Opens the Add Source Breakpoint dialog box where you can set a breakpoint on a specific line location in your source code. When you run your program, the execution point in the Code editor indicates the breakpoint location.
Address Breakpoint	Opens the Add Address Breakpoint dialog box where you can set a breakpoint on a specific machine instruction. When you run your program, the execution point in the CPU window Disassembly pane indicates the breakpoint location.
Data Breakpoint	Opens the Add Data Breakpoint dialog box where you can set a breakpoint on a specific address that halts execution when that address is written to.
Module Load Breakpoint	Opens the Add/Edit Module dialog box where you can halt execution on a module when it is loaded.

To add breakpoints from the the debugging views:

- Right-click and choose Add Breakpoint from the Breakpoint List context menu to bring up the Source, Address, and Data Breakpoint menus.
- Choose View|Debug Windows|Modules and right-click in the Modules List Window. Right-click and choose Add Module from the Module window context menu.

To associate actions with the breakpoints, see Associating actions with breakpoints.

Search menu

Search commands keyboard shortcuts See also

Use the Search menu commands to locate text, errors, objects, units, variables, and symbols in the Code editor.

Find	Searches for specific text, and highlights first occurrence in the code editor.
Find in Files	Searches for specific text, displays each occurrence in a window at the bottom of the code editor.
Replace	Searches for specific text and replace it with new text.
Search Again	Repeats the last search.
Incremental Search	Searches for text as you type.
Go to Line Number	Moves cursor to specific line number
Find Error	Searches for most recent runtime error.
Browse Symbol	Searches for specified symbol.

Search|Find

See also Find in Files Tab

Choose Search|Find to display the Find Text dialog box.

Find Text dialog box

Select the Find tab to specify text you want to locate and to set options that affect the search. Find locates the line of code containing the first occurrence of the string and highlights it.

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	Specified 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 expressions	Recognizes regular expressions in the search string.

Direction	Direction to search, starting from the current cursor position
Forward	From the current position to the end of the file. Forward is the default.
Backward	From the current position to the beginning of the file.

Scope	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	Where the search starts
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.

Search|Find in Files

See also Find Tab

Choose Search|Find in files to list occurrences of a specified string.

Find Text dialog box

Select the Find in Files tab to specify text you want to locate and to set options that affect the search. Each occurrence of the string is listed in a box at the bottom of the Code editor. Double-click a list entry to move to that line in the code file.

Text to find

Enter a search string. To select from a list of previously entered search strings, click the down arrow next to the input box.

Options	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 expressions	Recognizes regular expressions in the search string.

Where	Which files to search:
Search all files in project	Searches all files in the open project.
Search all open files	Searches files that are currently open.
Search in directories	When selected, the Search Directories options are available. The search proceeds through all files indicated.

Search directory options	Defines the full path to file(s) to be searched:
File masks	Specify the path of the files to be searched. By default, only .PAS and .DPR files are searched. To search other files, use a wildcard entry (such as *.* or *.txt) at the end of the path. To enter multiple masks, separate the masks with semicolons.
Include subdirectories	If selected, subdirectories from the directory path specified are also searched.

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.

Text to find

Enter a search string. To select from a list of previously entered search strings, click the down arrow next to the input box

Replace with

Enter the replacement string. To select from a list of previously entered search strings, click the down arrow next to the input box. To replace the text with nothing, leave this input box blank.

Options	**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 expressions	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	**Which direction to search the file**
Forward	From the current cursor position, to the end of the file. Forward is the default Direction setting.
Backward	From the current cursor position, to the beginning of the file.

Scope	**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	**Where the search should start:**
From cursor	The search starts at the cursor's current position, and 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 check Prompt on replace, 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

See Also

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 a new occurrence of the string is not found, the editor loses focus, or you press *Enter* or *Esc*.

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

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 number of the line 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.

Enter New Line Number	Specify the line number of the code you want to go to. To select from a list of previously entered line numbers, click the down arrow next to the input box.

Search|Find Error

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 runtime error.

Error Address The address of the most recent runtime error and the error number appears in the runtime error report if it is available.

When you click OK, Delphi recompiles your program and stops at the address location you entered, highlighting the line that caused the runtime error.

Search|Browse Symbol

Choose Search|Browse Symbol to display the Browse Symbol dialog box.

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, Delphi shows you information about the specified symbol in the Symbol Explorer. The Symbol Explorer provides information about code references to the symbol. If the symbol is a class, it also provides information about the class's members and ancestry. The Symbol Explorer is the same as the right-hand portion of the Project Browser.

Regular expressions

See also

Regular expressions are characters that customize a search string. Delphi recognizes these regular expressions:

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. . For the Replace operation, the groups are referred to by a backslash and a number according to the position in the “Text to find” expression, beginning with 0. For example, given the text to find and replacement strings, Find: {[0-9]}{[a-c]*}, Replace: NUM\1, the string 3abcabc is changed to NUMabcabc.
****	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 keystroke mappings.

Brief regular expressions

See also

Use these symbols to produce Brief regular expressions:

<	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 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
- View and change debugger settings
- Modify items in the Object Repository.
- Modify the list of programs on the Tools menu
- Create and modify local database tables.
- Create and edit package collections.
- Create and edit images.

Default Tools menu commands:

Environment Options	Specifies configuration preferences, library pathnames, and customizes the appearance of the Component palette.
Editor Options	Specifies Editor configuration preferences.
Debugger Options	Displays the Debugger Options dialog box.
Repository	Displays the Object Repository dialog box.
Translation Repository	Displays the Translation Repository.
Configure Tools	Displays the Tools Options dialog box. Use this dialog box to add commands to, delete commands from, or edit commands on the Tools menu.

Optional tools:

Database Desktop	Displays the database desktop where you can create, view, sort, modify, and query tables in Paradox, dBASE, and SQL formats.
Package Collection Editor	Create and edit package collections. Package collections are a convenient way to bundle packages and associated files for distribution to other developers.
Image Editor	Create and edit resource files, icons, bitmaps, and cursor files for use in applications.

Items representing other installed or custom tools may also appear on this menu.

TeamSource	Starts a workflow management tool that helps application development teams manage their daily tasks in a shared development environment. **Note:** *The TeamSource Tool is a separate product and requires a separate installation, and is not available in all versions of Delphi.*

Tools|Environment Options

Choose Tools|Environment Options to display the Environment Options dialog box. Use the pages of this dialog box to specify IDE 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
- Palette
- Explorer
- Type Library
- Delphi Direct
- Translation Tools

To change pages in the dialog box:
Click the tab at the top of the dialog box that represents the page you want to use.

Note: Program arguments are specified in the Tools|Environment Options dialog box and are passed when the program is invoked. If a program requires arguments to be entered at runtime, you can supply them on the Tool Properties dialog box.

Tools|Editor Options

Choose Tools|Editor Options to display the Editor Options dialog box. You can also right-click in the editor and choose Properties. Use the pages of this dialog box to specify editor preferences.

The pages of the Editor Options dialog box are:

- General
- Display
- Key Bindings
- Color
- Code Insight

Preferences (Tools|Environment Options)

See also Environment options

Use the Preferences page of the Environment Options dialog box to specify your configuration preferences.

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.
Project 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 used by the project. For more control over desktop arrangement, see the Desktops toolbar.

Desktop contents

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

Compiling and Running

Check box	When checked
Show Compiler Progress	Check to see progress reports while your program compiles.
Warn on package rebuild	Display warning when packages are rebuilt during compile.
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.

Form designer

Set grid preferences that make it easier to design forms.

Option	Effect
Display Grid	Displays dots on the form to make the grid visible.
Snap To Grid	Automatically aligns components on the form with the nearest gridline. You cannot place a component "in between" gridlines.
Show Component Captions	Select this option to display component captions.
Show Designer Hints	Toggles help hints on the surface of the form designer. Note that this option only affects hints that appear when you pause the mouse over a component in a form or data module: help hints are always enabled in the component palette.
New Forms as Text	Toggles the format in which form files are saved. The form files in your project can be saved in one of two formats: binary or text. Text files can be modified more easily by other tools and managed by a version control system. Binary files are backward compatible with earlier versions of

	Delphi. (You can override this setting on individual forms by right-clicking and checking or unchecking the Text DFM command.)
AutoCreate Forms	Toggles whether or not to automatically create forms. When unchecked, forms added to the project after the first one are put into the Available Forms list rather than the Auto Create list. (You can change where individual forms are listed using the Forms tab of the Project|Options dialog box.)
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.

Shared Repository

Option	Effect
Directory	Specifies the location where Delphi looks for the Object Repository file (DELPHI32.DRO). If this field is empty, Delphi looks for the file in the BIN directory.

Package <name> is about to be compiled. Continue?

See also

This message is displayed when you are compiling a package or an application that implicitly calls a package that is already installed in the IDE. If you say Yes, the package is unloaded, compiled, then reloaded again into the IDE and may affect design time or runtime IDE operations.

If any forms are displayed, the message warns you that the forms will be closed.

The Don’t show this message again check box allows you to turn this notification off. Checking this box turns off the Warn on package rebuild option on the Preferences page of the Tools|Environment options.

Library page (Tools|Environment Options)

See also Environment options

Use this page to specify directories, compiler, and linker options for all packages. Click the down arrow next to the edit boxes to choose from a list of previously entered symbols.

Directory	Description
Library path	Specifies search paths where compiler can find the source files for the package. The compiler can find only those files listed in Library Path. If you try to build your package with a file not on the library path, you will receive a compiler error.
BPL output directory	Where the compiler should put compiled packages (.BPL) files.
DCP output directory	Specifies a separate directory to contain the .dcp files.
Browsing path	Specifies directories where the Project Browser looks for unit files when it cannot find an identifier on the project Search path or Source path.

Edit controls that permit multiple values have an ellipses button to the right. Click this button to add multiple values using a List Entry dialog. Alternately, you can specify multiple values by separating them with semicolons.

General (Tools|Editor Options)

See also

Use the General page of the Editor Options dialog box to customize the behavior of the code editor.

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 logical spaces within each tab character.
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 appended following 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 highlighting options, use the Color page.

Editor SpeedSetting

Use the Editor SpeedSettings to configure the editor. The drop-down list includes pre-configured default settings that can be customized.

Note: SpeedSettings are a quick way to set the Editor options. To set the keyboard mappings in the editor, use the Key Mappings page.

Option	Automatically sets
Default Keymapping	Auto Indent Mode, Insert Mode, Cursor through tabs, Group Undo, Overwrite Blocks
IDE Classic	Auto Indent Mode, Insert Mode, Cursor Through Tabs, Group Undo, Persistent Blocks
Brief Emulation	Auto Indent Mode, Insert Mode, Cursor Through Tabs, Cursor Beyond EOF, Keep Trailing Blanks, Brief Regular Expressions, Force Cut And Copy Enabled
Epsilon Emulation	Auto Indent Mode, Insert Mode, Cursor Through Tabs, Group Undo, Overwrite Blocks
Visual Studio Emulation	Auto Indent Mode, Insert Mode, Cursor Through Tabs, Group Undo, Overwrite Blocks

Other options	When selected
Block Indent	Specify the number of spaces to indent a marked block. The default is 2; 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.
Syntax Extensions	Specify, by extension, which files will display syntax highlighting information. The default extensions are .PAS, .DPR, .DPK, .INC, and DFM.

Display (Tools|Editor Options)

See also

Use the Display page of the Editor 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 options

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 main window when maximized.

Text settings

These settings allow you to change the font, size, and location of the text in the text editor.

Setting	Effect
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, an error message appears.
Visible gutter	Check to display the gutter on the left edge of the Code editor.
Gutter width	Set the width of the gutter, default is 30.
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	Displays a sample of the select editor font and size.

Key Mappings (Tools|Editor Options)

See also

Use the Key Mappings page of the Editor Options dialog box to specify key mapping modules and to enable or disable enhancement modules including what order to initialize them.

Key mapping modules

Enables you to quickly switch key bindings.

Mapping	Effect
Default	Uses key bindings that match CUA mappings (default)
IDE classic	Uses key bindings that match Borland Classic editor keystrokes
Brief emulation	Uses key bindings that emulate most of the standard Brief keystrokes
Epsilon emulation	Uses key bindings that emulate a large part of the Epsilon editor
Visual Studio emulation	Uses key bindings that emulate a large part of the Visual Studio editor
New IDE Classic	Uses the key bindings defined in the editor keybinding demo.

Enhancement modules

Enhancement modules are special packages that are installed and registered and use the keyboard binding features that can be developed using the Open Tools API. You can create enhancement modules that contain new keystrokes or apply new operations to existing keystrokes.

Once installed, the enhancement modules become visible in the Enhancement modules list box. Clicking the check box next to the enhancement module enables it and unchecking it disables it. Key mapping defined in an installed and enabled enhancement module overrides any existing key mapping defined for that key in the key mapping module which is currently in effect.

Color (Tools|Editor Options)

See also Environment options

Use the Color page of the Editor 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 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 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)
Integer	Error line
Float	Preprocessor
Octal	Illegal char
Hex	Plain text
Character	Right margin

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, as you will be unable to read any text).

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 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 the Windows Control Panel.

Using syntax highlighting

See also

Syntax highlighting changes the colors and attributes of your text in the Code editor, making it easier to quickly identify parts of your code.

To enable syntax highlighting:

On the Editor Options page of the Tools|Environment Options dialog box, check the Use Syntax Highlight option.

To change the syntax highlighting colors for elements of your code:

Use the Color page of the Tools|Editor Options dialog box.

Palette (Tools|Environment Options, Component|Configure Palette)

See also Environment options

Use the Palette page of the Environment Options dialog box to customize the way the component palette appears. You can rename, add, remove, or reorder pages and components.

Pages Lists the pages in the component palette, in the order in which they currently appear. You can rearrange these pages or view and rearrange their components in the Components list. The last item in the Pages list is [All]; when you select [All], the Components list shows components from every page as well as hidden components.

Components Lists the components on the currently selected Component-palette page in the Pages list. Components may come from installed packages or they may be component templates created with the Component|Create Component Template command. Components appear in their current order on the palette. You can rearrange components, or move them to a different page, by dragging them. When [All] is selected in the Pages list, you can sort by component name, package, or palette page by clicking on the appropriate column heading.

Use the following buttons when an item is selected in the Pages list.

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 move components from other pages onto it or add new components to it using Component|Install.

Delete To remove the selected page from the palette, click Delete. Before you can delete a page, it must be empty of components. If you accidentally delete a component, select [All] in the Pages list and press Default Pages, or use Component|Install to re-add it.

Rename Click Rename to display the Rename Page dialog box, where you can rename the selected page.

Default Pages Click Default Pages to restore pages to their default order and replace all components on their default pages. This button is available when [All] is selected on the Pages list.

Move Up / Move Down To change the position of the selected page, click Move Up or Move Down. You can also drag pages to a new position.

Use the following buttons when an item is selected in the Components list.

Hide / Show To prevent an installed component from appearing on the Component palette, click Hide. To redisplay a hidden component, select [All] on the Pages list, select the hidden component on the Components list, then click Show.

Delete To delete a component template, click Delete. This button is available only when a component template is selected.

Move Up / Move Down To change the position of a component on a page, click Move Up or Move Down. You can also drag components to a new position.

Rename Page dialog box

See also

Use this dialog box to specify a new name for a page on the Component Palette.

To open this dialog box:

On the Palette page of the Tools|Environment Options dialog box, select the page to rename and click the Rename button.

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.

Add Page dialog box

See also

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 Tools|Environment Options dialog box.

To open this dialog box:

Click the Add button on the Palette page of the Tools|Environment Options 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.

About Code Insight

See also

Code Insight is a set of tools that support you while you are writing code by doing the following:

- Display information in the Code editor to help you with code syntax and arguments
- Provide common programming statements for you to insert in your code
- Display classes, functions, methods, arguments, and events parameter lists
- Show you the value of a variable while debugging
- Display declaration information for identifiers

The information displayed by these tools is created dynamically from your code. This means that Code Insight can display information about a method or declaration that you just finished writing. Code Insight references compiled code in binary files and the non-compiled code that you are currently typing into the editor.

Some of the features can be set for automatic display and others are instigated by pressing a key combination. To enable and configure Code Insight features, select Tools|Editor Options and display the Code Insight page.

Code Insight provides five tools:

Code completion

Code completion displays a list box that includes the names of the methods and fields of an object. When enabled, the Code completion list box is automatically displayed after you enter

- The name of a class object followed by a period

then the list of members of the class is automatically displayed.

- For pointers to objects, the arrow

The list of properties, methods, events, and variables belonging to the class, function, or struct are displayed. Select the item to be entered in your code and press Enter.

You can also use code completion to list the variables that can be legally inserted after an assignment operator. Type the name of an object, and then Press Ctrl+Space to see a list of the methods it supports. Select an argument to be entered in your code. Similarly, you can display a list of arguments when typing a procedure, function, or method call and need to add an argument, or when you are typing an array property (not a genuine array) and you need to type an index expression. You can specify how the list of arguments is sorted by right clicking in the Code Completion popup list and choosing Sort by Name or Sort by Scope

Note: Code completion and code parameter features work best when you have already built your application and have created a precompiled header. Otherwise, you need to wait for the compiler to generate the required information.

Code parameters

This tool displays a dialog that tells you the names and types of the parameters for a function, method, or procedure. Therefore, you can view the required arguments for a function, method, or procedure as you enter it into your code. If automatic Code parameters is enabled, a list box appears when you enter a routine or method call followed by the opening parenthesis; the syntax for the arguments is displayed. Press Shift+Ctrl+Space to display the list box at any time, whether or not Code parameters is on automatic.

Code templates

A set of code templates is available to insert commonly used programming statements into your source code. Delphi comes with a set of templates for basic code constructs that can be easily inserted into your code. Some of the constructs covered by the default templates include class declarations, if statements, for statements, and while loops.

While working in the Code editor, press Ctrl+J to display the code templates defined. Double-click on any one of them to insert it into your code.

Many default templates with Delphi, and you can add as many as you like. For instance, if you type the letter p, then press Ctrl-J, the following code will be inserted automatically at the current cursor position:

```
procedure ();
    begin
    end;
```

Templates can be edited and added from the Code Insight page (Tools|Editor Options).

Tooltip expression evaluation

To make it easy to see the value of a variable at any point, enable Tooltip expression evaluation. (To enable Tooltip expression evaluation, use the Code Insight dialog box.) When you are debugging and you're stopped, you can point to a variable to display its value at that time.

When optimization is enabled for the compiler, you might sometimes see a blinking bubble that says "Evaluating" rather than the value. Disable optimization when debugging. (Choose Project|Options, select the Compiler page, and uncheck the Optimization box under Code Generation.)

Tooltip Symbol Insight

Tooltip Symbol Insight displays declaration information for any identifier when you pass the mouse over it in the Code editor. A pop-up window shows the kind identifier (procedure, function, type, constant, variable, unit, and so forth) and the unit file and line number of its declaration. You can use the Code Browser feature to jump directly to the declaration.

Code Insight (Tools|Editor Options)

See also

Use the Code Insight page to configure Code Insight options. Code Insight tools are available while you are working in the Code editor.

Automatic Feature	When Enabled
Code completion	When you enter a class name followed by a period in the Code editor, the list of properties, methods and events appropriate to the class or record is displayed. You can then select the item and press Enter to add it to your code. Enter an assignment statement and press *Ctrl*+*Spacebar*. A list of arguments that are valid for the variable is displayed. Select an argument to be entered in your code. **Note:** You can always invoke Code completion using *Ctrl*+*Spacebar,* even if the automatic feature is disabled.
Code parameters	View the syntax of a prototype method as you enter it into your code. **Note:** You can always invoke Code parameters using *Shift*+Ctrl+Spacebar*,* even if the automatic feature is disabled.
Tooltip expression evaluation	When the compiler is stopped while debugging, you can view the value of a variable by pointing to it with your cursor.
Tooltip symbol insight	Display declaration information (in a pop-up window) for any identifier by passing the mouse over it in the Code editor.
Delay	Set the duration of the pause before a Code Insight dialog box is displayed.
Code templates	Available code templates are listed by name with a short description. Click a template name to display the code that will be entered in your file when that template is selected. Code displayed in the code window can be edited.
Templates	The Templates box includes a name and short description of each template.
Code	The code box displays the code that will be inserted into a file when the template is selected. The code displayed can be edited.

Adding, editing, and deleting code templates

You can add, edit or delete code templates from the Code Insight page of the Editor Options dialog box:

To edit a template's name and description:
Select the name you want to edit. Click Edit. Edit the name and description fields as needed and click OK.

To edit a template:
When a name is selected, the code to be inserted in the file when the template is selected is displayed in the edit list box. Edit the text as needed.

To define the insertion point for a template:
Place a vertical bar in the code statement to define the point to begin insertion when the template is inserted in a code file. The cursor will be placed in the location defined by the vertical bar.

To add a template:
Click Add. After entering a Name and Description in the dialog box displayed, click OK. The cursor will move to the code window for you to define the code that will be entered in a file when the template is selected.

To delete a template:
Select the name of the template you want to delete. Click Delete.

Code completion

See also

The code completion list box lists the valid properties, methods, and events for the name of a class object that you entered. You can right-click on the list to sort the items by name or by scope.

Arguments that are valid for assignment to the variable entered are listed. Select an item from the list followed by the ellipsis (…) to open a second list of related arguments compatible with the variable entered in the assignment statement.

Select an item to be entered in your code file in either of two ways:

- Scroll through the list if necessary and double-click the item.
- Type until the characters entered refer to the entry in the list you want to include and press Enter.

Preferences for code completion are configured by selecting Tools|Editor Options to open the Editor Options dialog box. Select the Code Insight page.

Code templates

Code templates include commonly used programming statements (such as if, while, and for statements) that you can insert into your code. The templates defined for your installation and some provided by default are listed. Select a code template to be entered in your code file in one of these ways:

- Use the scroll bar as necessary then double-click on the template to insert in your code.
- Type the name of the template until the characters entered refer to the entry in the list you want to include. Press Enter.
- Use the scroll bar as necessary then double-click on the template to insert in your code.

Code templates are defined using the Environment Options dialog box. Select Tools|Environment Options and select the Code Insight page.

Type Library (Tools|Environment Options)

See also Environment options

Use the Type Library page of the Environment Options dialog box to select options for the Type Library editor. The new settings take effect when you click OK.

SafeCall function mapping

These options determine which functions are declared as **safecall** when declarations specified in Object Pascal are converted into IDL in the generated type library. Safecall functions automatically implement COM conventions for errors and exception handling, converting HRESULT error codes into exceptions. If you are entering function declarations in IDL (see Editor Language), you must explicitly specify the calling convention as safecall or stdcall.

Check box	Description
All v-table interfaces	Use SafeCall for all interfaces.
Only dual interfaces	Use SafeCall only for dual interfaces.
Do not map	Do not use the SafeCall calling convention.
Display updates before refreshing	This option displays the Apply Updates dialog box which provides a chance to veto proposed changes to the sources when you try to refresh, save, or register the type library. If not checked, the Type Library editor automatically updates the sources of the associated object when you make changes in the editor.

Editor Language

You can select the language to use in the Type Library editor for entering the interface details. You can choose either Pascal (Object Pascal language) or IDL (Microsoft Interface Definition Language). You will probably use Pascal for CORBA interfaces because the CORBA IDL differs slightly from Microsoft IDL.

Apply Updates dialog box

The Apply Updates dialog box is displayed if you check the Display updates before refreshing option on the Type Library page of Tools|Environment Options and you try to modify a type library. This dialog box is displayed allowing you to recheck proposed changes to the sources when you try to refresh, save, or register the type library.

Select Updates

In this list box, you see a list of changes, in order, that will be made to your project. You can check or uncheck the box next to each change to include or exclude the changes to that file. If you uncheck a change on which later changes depend (for example the creation of a file to which later changes add code), the later changes are automatically unchecked.

Details

This list box displays all the changes that will be added to implement the currently selected change. When you click OK, the changes in this edit window, including any modifications you make within the dialog, are added for every update checked in the Select Updates list.

If an update consists of new code that is added to a file, the Details box shows a single edit control that displays the new code. If the update modifies existing code, the Details page shows two text windows: the first is the new code that reflects the modifications, and the second shows the original code that has been changed.

Don’t show this dialog again

This checkbox indicates whether you want this dialog box to be displayed each time you modify a type library and attempt to refresh, save, or register the type library.

Check this box to implement changes without checking with you. (Checking this box unchecks the Display updates before refreshing option on the Type Library page of Tools|Environment Options.)

Explorer (Tools|Environment Options)

See also Environment options

The Code Explorer contains a tree diagram that shows all the types, classes, properties, methods, global variables, and global routines defined in the unit that is currently displayed in the Code editor. The Project Browser displays classes, units, and identifiers associated with your project.

Use the Explorer page of the Environment Options dialog box to select options for the Code Explorer and the Project Browser. The new settings take effect when you click OK.

Explorer options

These options determine how the Code Explorer is displayed.

Check box	Effect
Automatically show Explorer	Code Explorer appears docked onto the Code editor. When unchecked, use View|Code Explorer to display.
Highlight incomplete class items	Incomplete properties and methods appear in bold in the Explorer.
Show declaration syntax	By default, only the names of code elements are displayed in the Code Explorer. Check this to show the syntax and type of methods or properties.

Explorer sorting

These options determine how elements will be sorted in the Code Explorer.

Radio button	Effect
Alphabetical	All source elements are listed alphabetically in the Code Explorer.
Source	Source elements are listed in the order in which they are declared in the source file.

Class Completion Option

This option determines how class completion works (Shift+Ctrl+C).

Check box	Effect
Finish incomplete properties	If you write a property declaration, completes the remainder of the declaration for reading and writing that property. If unchecked, class completion applies only to methods.

Initial browser view

Radio button	Effect
Classes	Displays the browser with the classes information on top.
Units	Displays the browser with the units information on top.
Globals	Displays the browser with the globals information on top.

Browser scope

Check box	Effect
Project symbols only	The browser displays symbols from units in the current project only.
All symbols (VCL included)	The browser displays symbols from all units used (directly or indirectly) by the current project.

Explorer categories

These options let you control how source elements are categorized in the Code Explorer or Project Browser. If a category is checked, elements of that type are grouped under a single node in the tree diagram. If a category is unchecked, each element in that category is displayed independently on the diagram's trunk. The Virtuals, Statistics, Inherited, and Introduced categories are for the Project Browser

only.

The folders in bold take precedence when a conflict exists and an element can appear in two folders. For example, a private field would be listed in the private folder if both Private and Fields were checked.

If a folder is checked, the glyph to the left of the checkbox shows whether the folder is expanded. Click there to expand or close a folder. The change goes into effect when you click OK.

Delphi Direct (Tools|Environment Options)

Environment options

Delphi Direct provides access (in your default browser) to the latest Delphi news posted online. Use the Delphi Direct page of the Environment Options dialog box to control how often Delphi Direct picks up new information from borland.com.

Option	Description
Polling interval	Specifies how often to to get new information from borland.com.
Last poll	Indicates the last time information was updated from borland.com.
Automatically show Delphi Direct on refresh	Displays Delphi Direct automatically upon getting new information from borland.com.

Translation Tools (Tools|Environment Options)

See also Environment options

Use the Translation Tools page of the Environment Options dialog box to configure the Integrated Translation Environment (ITE).

Repository

Sets the location of the Translation Repository, a database for translations that can be shared by different projects.

Filename	Enter the full name and directory path of the .RPS file where the Translation Repository is stored.

Resource DLL wizard

Options for the Resource DLL wizard, which generates resource DLLs for localized versions of a project.

Automatic repository query	Automatically populate resource DLLs with translations for any strings that have matches in the Repository.
Automatically compile projects	Compile projects, without asking first, whenever required by ITE tools (for example, when running the resource DLL wizard).
Show Translation Manager after RDW	Automatically open the Translation Manager after running the Resource DLL wizard.

Multiple find action

Determines how the Repository responds when it finds more than one translation for the same source string.

Skip	Don't retrieve anything if the Repository contains more than one match.
Use first	Retrieve the first match.
Display selection	Offer the user a choice. (Works only from the Repository pop-up menu in the Translation Manager. For Auto Translate, this option is equivalent to **Skip**.)

Grid fonts

Select the fonts that are available for different languages in the ITE.

Debugger Options **(Tools|Debugger Options)**

Choose Tools|Debugger Options to display the Debugger Options dialog box containing several tabbed pages of settings:

- General
- Event Log
- Language Exceptions
- OS Exceptions
- Distributed Debugging
- Integrated debugging check box: Check to set the Integrated Debugger to active.

General (Tools|Debugger Options)

Use the General page of the Debugger Options dialog box to set general debugger options primarily for the user interface. Check the options you want to use.

Option	Description
Map TD32 keystrokes on run	Allows you to use the keystrokes from TD32 in the IDE. It will automatically turn on **Mark buffers read-only on run**.
Mark buffers read-only on run	Marks all editor files, including project and workgroup files, read-only when the program is run. When this option is selected, it will not change the attributes of the files after the program terminates. If the file was not marked read-only before running the program, Delphi will change the attributes of the file back to their original configuration after the program terminates.
Inspectors stay on top	Keeps all debugger inspector windows visible when they are not active.
Allow function calls in new watches	Allows function calls in new watches. By default, this option is not set.
Rearrange editor local menu on run	Moves the Debugger area of the Code editor context menu to the top when you run a program from the IDE to more easily access the Debugger commands. Display the Code editor context menu by right-clicking anywhere in the Code editor window.
Debug spawned processes	Automatically debugs processes which are spawned by the process you are debugging. If not checked, spawned processes are run but are not under control of the debugger.

Paths

Type	Description
Debug Symbols Search Path	Specifies the path to your debug symbols including any TDS, RSM, and DCP files. These files are normally stored with your PKG, EXE, or DLL file.
Debug DCU Path	To use this option, you must also set **Use Debug DCUs** on the Project|Options Compiler page. When that option is set and a path is given, the debugger looks for DCUs in this path before looking in the unit search path.

Distributed Debugging (Tools|Debugger Options)

See also

Use the Distributed Debugging page of the Debugger Options dialog box to set debugger options for remote debugging (available only when using the Enterprise edition).

Option	Description
Enable COM cross-process support	Cross-process stepping option that lets you step into remote COM processes while debugging. Also, adds COM events to the event log. This option is off by default.
Enable CORBA cross-process support	Cross-process stepping option that lets you step into remote CORBA processes while debugging. When checked, the controls in the ORB events list box are enabled and you can set options for each ORB event that the debugger sends notifications for. For each event, you can define any number of action sets. This option is off by default.
ORB Events list box	Set options for different ORB events that the debugger knows about. The Enable CORBA cross-process support option must be checked to access ORB events. Click the event you want to set options for and add/remove actions using the Actions list box.
Actions list box	Lists relevant actions for the selected ORB event. Add or remove actions for selected ORB events.
Integrated debugging check box	Check to set the Integrated Debugger to active. This option is on by default.

ORB events

See also

When the **Enable CORBA cross-process support option** is checked on the Tools|Debugger Options| Distributed Debugging page, the controls in the ORB events list box are enabled. You can set options for each ORB event that the debugger sends notifications for. All of these events are automatically logged in the event log if the CORBA option is checked.

For each event, you can define any number of action sets. An action set consists of one or two actions and one or more optional conditions.

Actions

The possible actions are

Action	Description
Log Event	Adds the event to the event log. On by default for all ORB events.
Break on Event	Stops the process when the event occurs.

ORB Events and Conditions

You can set specified conditions for the ORB events listed below in Conditions for each ORB event. The possible conditions depend on which event is selected.

For example, on the client side, you could set interface, object, and operation conditions for SendRequest. You could specify that the interface=Account, the object=Jack B. Quick (the instance of Account or a person's account), and the operation =balance. If you check Break on Event when setting these conditions, processing stops when a request is sent from a client to a server to get Jack B. Quick's account balance).

To set actions and conditions for an ORB event:

1. Check the Enable COM cross-process support option on the Distributed Debugging page of the Debugger Options dialog box.
2. Select the event for which you want to set conditions.
3. Click Add.

   The ORB event dialog box with appropriate conditions is displayed. (Some events have no conditions, others have one, two, or three.) The name of the ORB event is in the header.
4. Specify one or two actions (Log Event or Break on Event) you want to perform when the event occurs.
5. Optional: Specify under what conditions you want the specified action to take place for that ORB event. Type the name of a valid interface, object, and/or operation (depending on which fields are available in the dialog box).
6. Click OK.
7. Continue adding as many actions and conditions as you want for each ORB event (repeat steps 3-6) until you are tracing and breaking on the events you want.

The current actions for that event are then listed in the Actions list box.

Conditions for each ORB event

Following are the ORB events that the debugger tracks and the conditions that you can set for each:

ORB event	Conditions
Bind	Interface, Object
BindExceptionOccurred	Interface, Object
Bind Failed	Interface, Object
Bind Succeeded	Interface, Object

ClientExceptionOccurred	Interface, Object
ClientPrepareRequest	Interface, Object, Operation
ClientReceiveReply	Interface, Object
ClientReceiveReplyFailed	Interface, Object
ClientSendRequest	Interface, Object, Operation
ClientSendRequestFailed	Interface, Object, Operation
ClientSendRequestSucceeded	Interface, Object, Operation
Rebind	Interface, Object
RebindFailed	Interface, Object
RebindSucceeded	Interface, Object
ServerExceptionOccurred	Operation
ServerLocate	None
ServerLocateFailed	None
ServerLocateForwarded	None
ServerLocateSucceeded	None
ServerPrepareReply	Interface, Object, Operation
ServerReceiveRequest	Interface, Object, Operation
ServerRequestCompleted	Interface, Object, Operation
ServerSendReply	Interface, Object, Operation
ServerSendReplyFailed	Interface, Object, Operation
ServerShutdown	None

For more information on the specific ORB events, look up “interceptors” in the VisiBroker documentation.

ORB event dialog box

See also

This dialog box is displayed when setting actions or conditions for an ORB event. The name of the ORB event for which you are setting options appears in the header of the dialog box.

You can set the following actions for each ORB event.

Action	Description
Log Event	Adds the event to the event log. On by default for all ORB events.
Break on Event	Stops the process when the event occurs.

In addition, you can set none, some, or all of the following conditions for each ORB event. If conditions are set, you'll only break or log the specific event when one or more conditions trigger the action to occur. See Conditions for each ORB event for which conditions apply to each event or just look at the dialog box to see which conditions are available.

Condition	Description
Interface	Name of the CORBA interface that will trigger the trace or break
Object	The specific instance of the CORBA interface that will trigger the trace or break
Operation	The method that will trigger the trace or break

Event Log **(Tools|Debugger Options)**

See also

Use the Event Log page of the Debugger Options dialog box to set event log options. The event log shows process control messages, breakpoint messages, OutputDebugStrings messages, and window messages. By right-clicking, you can bring up the context menu to clear the event log, save the event log to a text file, add a comment to the event log and set options for the event log. By setting options, you can control how many messages to display and how many events to show.

General

Option	Effect
Clear log on run	Causes the event log to be purged at the start of each debug session. If this option is checked while debugging multiple processes, the event log view is cleared when the very first process is started. However, any process started while at least one process is already being debugged will not cause the event log view to be cleared.
Unlimited Length	Removes the limit on the length of the event log. When this option is unchecked, set the maximum length of the event log in the **Length** field.
Length	Displays the maximum length of the event log. If the **Unlimited Length** check box is checked, this option is inactive. For multiple process debugging, length is the total for the event log, not for a process.
Display process info with event	When checked, shows the process name and process ID for the process that generated each event.

Messages

Messages	Effect
Breakpoint messages	Enabling writes a message to the event log each time a breakpoint or First-chance exception is encountered. The message includes the current EIP address of the program being debugged in addition to information about the breakpoint (pass count, condition, source file name, and line number) or exception.
Process messages	Enabling writes a message to the event log each time a process loads or terminates, whenever a module is loaded or unloaded by the process.
Output messages	Enabling writes a message to the event log each time your program or one of its modules call OutputDebugString.
Window messages	Enabling writes a message to the event log for each window message that is sent or posted to one of your application's windows. The log entry will have details about the message, including the message's name and any relevant data encoded in its parameters. Messages are not immediately written to the log if your process is running and not stopped in the debugger. As soon as you pause the process in the debugger (by encountering a breakpoint or using Run| Pause) the messages will be written to the event log.

Integrated debugging check box Check to set the Integrated Debugger to active.

The default settings for these options is:

- Length (100)
- Unlimited Length (ON)
- Clear Log on Run (ON)
- Breakpoint Messages (ON)
- Process Messages (ON)
- Output Messages (ON)

- Window Messages (OFF)
- Integrated debugging (ON)

COM options

When the Enable COM cross-process support option on the Distributed Debugging page of the Debugger Options dialog box is checked, COM events are added to the event log. There are three types of COM events: ClientStart, ServerStart, and ClientEnd. Each event shows the GUID, the method number, and the HResult of the COM RPC.

CORBA options

When the Enable CORBA cross-process support option on the Distributed Debugging page of the Debugger Options dialog box is checked, you can step into remote CORBA processes while debugging. When checked, the controls in the ORB events list box are enabled and you can set options for each ORB event that the debugger sends notifications for. For each event, you can define any number of action sets. Events for which you set options are added to the event log.

Language Exceptions (Tools|Debugger Options)

Use the Language Exceptions page of the Debugger Options dialog box to configure how the debugger handles language exceptions when they are raised by the program you are debugging.

Exception Types to Ignore	Lists types of exceptions you want the debugger to ignore (checked) or not (unchecked) while debugging. The debugger does not halt execution of your program if the exception thrown is listed and checked. It will not halt execution if the exception thrown is derived from any exception listed in the list box and checked. You can add and remove additional types of exceptions to the list box using the Add and Remove buttons. For example, if you add EMathError to the list and check it, and your program raises an EMathError exception, the debugger will not stop your program at that point. Additionally, if your program raises an EOverflow exception, the debugger will not stop because EOverflow is derived from EMathError.
Stop on Delphi Exceptions	Check the "Stop On Delphi Exceptions" checkbox if you want the debugger to halt execution of your program when your program raises a Delphi exception. By default, this checkbox is checked. If checked, you can tell the debugger to ignore specific exception types by using the "Exception Types to Ignore" listbox. The default for this setting is ON.
Add button	Click the Add button to bring up the Add Exception dialog.
Remove button	Click to remove a selected item from the list. Select the item you want to remove and click Remove.
Integrated debugging	Check to set the Integrated Debugger to active. This option is on by default.

Exception Types to Ignore

The following default exception types are listed in the Exception Types to Ignore list box on the Language Exceptions page of the Debugger Options dialog box.and cannot be removed from the list:

Exception type	Default	Maps to
VCL Eabort Exception	Ignored	EAbort
Visibroker Internal Exceptions	Ignored	IODictionary<IOUniqueId,dpIOHandler*> ::OBJECT_NOT_EXIST
CORBA System Exceptions	Ignored	CORBA_SystemException and CORBA_SystemException*
CORBA User Exceptions	Not ignored	CORBA_UserException and CORBA_UserException *

You can add more exceptions to the list box by clicking Add and typing the name of the exception. Added exceptions will include a check box that lets you check the items you want to ignore and uncheck items you want the debugger to stop on.

OS Exceptions **(Tools|Debugger Options)**

The scroll box on the top lists exceptions, and in the fields on the bottom you specify how the exception will be handled. To change the options for handling exceptions, highlight the exception you want to change and adjust the **Handled By** and **On Resume** options.

- **Handled By** specifies whether the exception will be handled by the Debugger or by your program. If you have added exception handling to your project, select User Program.
- **On Resume** specifies whether Delphi will continue to handle the exception, or whether the project will run unhandled.
- **Add button** bringsup the Add Exception Range dialog box. This allows you to add user-defined exceptions to be handled by the debugger.
- **Remove button** removes a selected item from the list. Select the item you want to remove and click Remove. This allows you to remove a user-defined exception from the list. Currently, you can only remove exceptions that you have added. This button will be gray anytime a default exception is selected. You can only remove user-defined exceptions.
- **Integrated debugging check box** sets the Integrated Debugger to active. This option is on by default.

Add Exception Range dialog **(Tools| Debugger Options|OS Exceptions)**

Choose Add from the Tools| Debugger Options| OS Exceptions tab Add button to bring up the Add Exception range dialog to specify a range of exceptions on which you want to break.

If you give a low and high value, Delphi stops on any OS exception with a value in the specified range. To stop on a single value, specify the same value for the low and high range. Specify the lower and upper range of the exception in the **Range Low** and **Range High** fields.

Note: To determine the numeric value associated with each OS exception, see the Exceptions list on the OS Exceptions tab of the Debugger Options dialog box.

Add Language Exception dialog **(Tools| Debugger Options|Language Exceptions)**

Choose Add from the Tools| Debugger Options| Language Exceptions tab Add button to add types to the list by entering the type name in the edit window of the Exception Type drop-down listbox.

Tools|Repository

See also

Choose Tools|Repository to display the Object Repository dialog box. 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

Pages included on the Object Repository correspond to the User Defined pages in the New Items dialog box displayed by File|New. The objects listed are available from the New Items dialog box.

Object Repository dialog box

See also

Choose Tools|Repository to display the Object Repository dialog box.

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 default options listed below become available.

- New form
- Main form
- New project

You 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.

Object Repository Options	Description
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. The pages in the Object Repository correspond to the user defined pages in the New Items dialog box. Select File|New to display the New Items dialog box.
Objects	The Objects list box displays the items on the currently selected page of the Object Repository.
Add Page button	To add a new blank page, click the Add Page button. The Add Page dialog box appears. Type the name of the page you want to add and click OK.
Delete Page button	To remove an empty page from the Object Repository, in the Pages list box, select the name of the page you want to delete and click the Delete Page button
Rename Page button	To rename an item in the Object Repository, in the Pages list box, select the name of the page you want to rename and click the Rename Page button. The Rename Page dialog box appears. Type the name of the page you want to rename and click OK. The renamed page appears in the Pages list box.
Edit Object	To edit the properties of items in the Object Repository, from the Objects list box, select the item you want to edit and click the Edit Object button. The Edit Object Info dialog box appears. Edit the information as desired and click OK.
Delete Object	Use the Delete Object button to remove the selected object from the Object Repository page.
Up/Down arrows	To change the position of the selected page, click the up arrow or the down arrow. You can also move pages by using a drag-and-drop operation.

There are three ways to add an object to a page:

- Right-click in a form, select Add to repository.
- Select Project|Add to repository.
- Drag an object listed in the Object column to a page listed in the Page column of the Object Repository.

Add Page dialog box

Use the Add Page dialog box to add a page to the Object Repository. You access the Add Page dialog box from the Object Repository dialog box.

Page name

Type the name of the new page into the Page name text box.

Rename Page dialog box

Use the Rename Page dialog box to rename a page in the Object Repository. You access the Rename Page dialog box from the Object Repository 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.

Options	Description
Title	Displays the title of the selected item.
Description	Displays the description of the selected item.
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.

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 context menu.

   The item description appears in the Description column.

Tools|Translation Repository

See also

Choose Tools|Translation Repository to display the Translation Repository.

Tools|Configure Tools

Choose Tools|Configure Tools to display the Tools Options dialog box.

Use the Tools Options dialog box to add, delete, or edit programs on the Tools menu.

Tools Options dialog box

Choose Tools|Configure 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. When two or more programs you have added to the Tools menu have conflicting shortcuts, a red star appears to the left of the program's entry in the list on the left.
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	Click 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.

Close

Click Close to return to the IDE.

To add a program to the tools menu:

1. Choose Add.

   The Tool Properties dialog box appears.
2. Specify a title for the program. The title you specify will be listed on the Tools menu.
3. Specify the program file or choose Browse to select it from a list.
4. Specify the working directory for the program, if necessary.
5. 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.
6. Choose OK.

   The Tool Properties dialog box closes. The new program is on the Tools list in the Tool Options dialog box.
7. Choose Close.

   The Tool Options dialog box closes. The new program is on the Tools menu.

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. The Tool Properties dialog box appears with information for the selected program.

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 Tools Options dialog box.

Tool Properties	Description
Title	Enter a name for the program you are adding. This name will appear on the Tools menu. To add an accelerator to the menu command, precede that letter with an ampersand (&). If you specify a duplicate accelerator, The Tool Options dialog box displays a red asterisk (*) next to the program names.
Program	Enter the location of the program you are adding. Include the full path to the program. To search your drives and directories to locate the path and file name for the program, click the Browse button
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. Type the parameters or use the Macros button to supply startup parameters. You can specify multiple parameters and macros.
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. Select a macro and click Insert to add the macro to the Program, Working dir, or Parameters text box above.
Browse	Click Browse to select the program name for the Program edit box. When you click Browse, the Select Transfer Item dialog box opens.

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.

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|Optional tools on the tools menu

The bottom of the Tools menu is customizable. You can remove the tools listed there or add other installed or custom tools you want to access while using Delphi.

Choose Tools|Configure Tools to display the Tools Options dialog box where you can add, delete, or edit programs on the Tools menu.

The following tools are displayed by default:

Database Desktop	Displays the database desktop where you can create, view, sort, modify, and query tables in Paradox, dBASE, and SQL formats.
Package Collection Editor	Create and edit package collections. Package collections are a convenient way to bundle packages and associated files for distribution to other developers.
Image Editor	Create and edit resource files, icons, bitmaps, and cursor files for use in applications.
TeamSource	Starts a workflow management tool that helps application development teams manage their daily tasks in a shared development environment. **Note:** *TeamSource is a separate product and requires a separate installation, and is not available in all versions of Delphi.*

Any other tools you have added to the Tools menu will also appear.

Tools|Database Desktop

Choose Tools|Database Desktop to display the Database Desktop. Database Desktop is a database tool where you can create or restructure database tables, or browse and edit their data. You can work with tables in Paradox, dBASE, and SQL formats. For details on using Database Desktop, click on dbddesk.hlp in the Borland\Database Desktop directory. A complete Database Deskop Help system is displayed.

Tools|Image Editor

Choose this command to invoke the Image Editor. The Image editor is a program that lets you create and edit images to use in your application.

View menu

Use the View menu commands to display or hide different elements of the Delphi environment and open windows that belong to the integrated debugger.

Project Manager	Displays the Project Manager
Translation Manager	Displays the Translation Manager
Object Inspector	Displays the Object Inspector
To-Do List	Lets you view the To-Do list associated with the current project.
Alignment Palette	Displays the Alignment Palette
Browser	Displays the Project Browser
Explorer	Displays the Code Explorer
Component List	Displays the Components dialog box
Window List	Displays a list of open windows
Debug Windows	Displays the Debugger submenu
Desktops	Lets you display, save, or delete different desktop views.
Toggle Form/Unit	Toggles between a form and its unit window
Units	Displays the View Unit dialog box
Forms	Displays the View Form dialog box
Type Library	Displays the Type Library editor window
New Edit Window	Opens a new Code editor
Toolbars	Hides or shows the toolbars or Component palette

View|Project Manager

See also

Choose View|Project Manager to display the Project Manager. If the Project Manager is already open, it becomes the active window.

Use the Project Manager window to view a project group, projects in a project group, and to navigate among a project's files. You can use the Project Manager to add projects to a project group or delete projects, or to activate a project if your project group consists of more than one project. You can also use the Project Manager to add, delete, save, or copy a file to the current project. The Project Manager lists all the units and associated forms in projects within the current project group.

You can position the Project Manager anywhere on your desktop. You can also dock it with other windows such as the Code editor or other tool windows.

View|Translation Manager

See also

Choose View|Translation Manager to display the Translation Manager. If it is already open, it becomes the active window.

View|Object Inspector

See also

If you have closed the Object Inspector, choose this command to reopen it. You can also choose View|Object Inspector to toggle between the Object Inspector and the last active form or Code editor file.

Use the Object Inspector to edit property values and event-handler links.

View|To-Do List

See also

Choose View|To-Do List to display the To-Do List for the current project. The to-do list displays tasks that need to be done to complete the current project.

Items for the entire project are listed. Items in the project whose source code is not open in the Code editor are shown in gray.

You can sort the items alphabetically, by status, or priority by clicking on the appropriate column.

To-do lists

See also

A to-do list records items that need to be completed for a project. You can add project-wide items to the to-do list by adding them directly to the to-do list, or you can embed specific items directly in the source code.

You can right-click in the to-do list to display the To-Do List context menu where you can edit, add, or delete to-do list items.

You can perform the following tasks:

- Adding items to a to-do list
- Adding to-do list items in the source code
- Editing to-do lists

After you create a to-do list, you can display it when the project is open.

To display a to-do list:

Choose View|To-Do List.

The following to-do items are shown in the to-do list:

- Items from the to-do file (called *project*.todo) for the current (active) project
- Items in source units that are part of the current (active) project
- Items in source units that are open in the editor

You can right-click and choose Filter to limit items that are displayed.

To-do list format

The to-do list has the following columns:

Column	Description	
Action Item	Includes three pieces of information:	
	Check box	Specifies whether or not the item has been completed (indicated by a box with or without a checkmark). A check means it has been done. Done items are shown as crossed out. If Show Completed Items is unchecked, completed items will not appear in the list.
	Kind	Indicates where the to-do list item originated Items are either entered in the project's to-do list (you see a window icon) or they are entered in the source code (you see a unit icon). This information lets you know where you can edit the item (see Editing to-do lists). If the unit icon for an item is grayed out, that source file is not part of the current project.
	Action Item	Lists the task to be done. If the item's text is grayed out, the item comes from a source file that is part of the current project but is not open in the editor. Double-click the item to open its source in the editor.
Priority	Specifies the importance of the item using a decimal number from 1 (the highest) to 5 (the lowest). The top of the column shows a boxed exclamation point. Specifying a priority of 0 assigns no priority to the item.	
Module	Names the module that the item concerns. This is automatically filled in when you add to-do list items in the source code.	
Owner	Says who's responsible for completing the task. Owner names can be any length and contain any characters except hyphen (-) or colon (:).	
Category	Indicates a type of task (for example, UI or error handling). Category names can be any length and contain any characters except hyphen (-) or colon (:).	

You can sort items by clicking on the column heading, for example, to sort action items alphabetically or by priority. Or you can use the Sort on the right-click menu.

You can also be selective about what items are visible in the to-do list. You can right-click and choose Filter to select items by owner, category, or item type. You can also right-click and choose Show Completed Items to display or hide items that are done.

Adding items to a to-do list

See also

To add items to the to-do list:

- Right-click on the to-do list and choose Add.

The Add To-Do List Item dialog box is displayed. You can also specify the priority, owner, and category of the item. This is the best way to add global items that concern the whole project.

When you add global items to the to-do list as described in this topic, a *project*.todo file is created and is stored with the project file.

You can also add to-do list items into the source code.

Adding to-do list items in the source code

See also

To embed to-do list items in the source code:

You can add specific items directly within the source code in two ways:

- Right-click in the Code editor and choose Add To-Do Item. Type the item in the Add To-Do List Item dialog box.
- Type the item in the source code using the to-do item syntax.

To-Do Item syntax

Use the following syntax for to-do list items in your source code:

```
{TODO|DONE [n] [-o<owner>] [-c<category>] : <to-do item text>}
```

The word TODO is changed to DONE to mark an item as completed or checked.

Where:

`n`	is a priority that can be set to a number from 1 (highest) to 5 (lowest). Setting n to 0 means assign no priority. It is optional but must be specified right after the TODO or DONE keyword.
`TODO`	is a keyword that indicates a to-do list item. When the item is completed, changes to DONE. Case is not important.
`DONE`	indicates a completed to-do item. Replaces the word TODO when you check an item in the to-do list. Case is not important.
`-o owner`	is the name of the person or group responsible for the item. It may contain spaces and is optional.
`-c category`	is the type of item, such as UI task. It may contain spaces and is optional.

Note The to-do item text many not contain any character (or characters) that terminate a comment.

To-do list items you enter in the source code are added to the list as you type them. The to-do item text is added to the list. The status, priority, and owner are added if you specified them in the code.

The order in which you specify owner and category is not important but the status (if included) must go first. The name of the module containing the embeded item is automatically added to the to-do list.

For example,

```
{TODO 2 -oNell: Implement stubbed out methods}
```

Creates a priority 2 to-do list item for which Nell is responsible and which says “Implement stubbed out methods”.

```
{Todo 1 -oSarah Alexander -cUI changes: Tell documentation about all changes
  }
```

Creates a priority 1 to-do list item for which Sarah Alexander is responsible and which says “Tell documentation about all changes”.

When you put a check mark in the to-do list for the above item, the syntax in the source code changes to the following:

```
{DONE 1 -oSarah Alexander -cUI changes: Tell documentation about all changes
  }
```

Editing to-do lists

See also

To edit items in a to-do list:

You edit to-do list items by selecting the item, right-clicking and choosing Edit. The Edit To-Do List Item dialog box is displayed where you can change the to-do list item, its priority, owner, or category, then click OK.

If an item's text is grayed out in the to-do list, it comes from a source file in the project that is not currently open. It can't be edited or deleted until it is open in the editor. Double-click the item to open the source file containing the item in the editor.

Click Done in the Edit To-Do Item dialog box to mark at item as completed (or click the checkbox within the to-do list).

You can also edit to-do list items that have been added in the source code directly within the source code itself. The syntax for these items is described in Adding to-do list items in the source code. The name of the module containing the to-do list item is listed in the to-do list.

View|Alignment Palette

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

Use the alignment palette to align components to the form, or with each other.

The alignment palette has Tool Help for each button.

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.
	Horizontally aligns three or more selected components so that the middle components are equidistantly spaced between the outer components.
	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.
	Vertically aligns three or more selected components so that the middle components are equally spaced between the outer components.
	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 acts, click and hold on the button. The icon on the button changes to show you how it will align the selected components. To apply the button's alignment to a selection, release the button. To prevent alignment after you click and hold the button, drag the mouse off the palette before releasing the mouse button.

Note: You can also use the Alignment dialog box to align components.

View|Browser

See also

Choose View|Browser to open the Project Browser.

On the Explorer page of the Environment Options dialog, you can set the scope of the project browser so that you view symbols from your project only or from all units used by your project. You can also set the initial browser view to the tab you want to see most often (classes, units, or globals).

The Explorer categories on the Explorer page of the Tools|Environment options let you control how source elements are grouped in the project browser. If a category is checked, elements of that type are grouped in a folder of that name. If a category is unchecked, elements of that type are shown outside of a folder.

View|Code Explorer

See also

By default, the Code Explorer is docked to the left of the Code editor. If this window is closed, you can reopen it by choosing View|Code Explorer.

The Code Explorer makes it easy to navigate through your unit files and automates the creation of classes.

- To close the Code Explorer, undock it and click the upper right corner.
- To reopen the Code Explorer, choose View|Code Explorer.

The Code Explorer contains a tree diagram that shows all the types, classes, properties, methods, global variables, and global routines defined in your unit. It also shows the other units listed in the **uses** clause. You can expand or collapse the nodes on the tree. Whichever unit file is open in the Code editor is also open in the Code Explorer.

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 mouse or keyboard.

Options	Description
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 list box	Select the component you want to add. Components are listed in alphabetical order, and their button representation is on the left.
Search edit box	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. To select Add To Form from the keyboard, press *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 by using the keyboard, 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.

View|Toggle Form/Unit

See also

Choose View|Toggle Form/Unit switch between the current form or its unit file.

Alternative ways to perform this command are:

- Choose the Toggle Form/Unit button from the toolbar.

View|Units

See also

Choose View|Units to display the View Unit dialog box.

View Unit dialog box

Use this dialog box to view 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 opens it.

Alternative ways to perform this command:

- Choose the View Units button on the toolbar.

View|Forms

See also

Choose View|Forms to display the View Form dialog box.

View Form dialog box

Use this dialog box to view 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:

- Choose the Select Form From List button from the toolbar.

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 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|Debug Windows

Use the View|Debug Windows menu commands to open windows that belong to the integrated debugger. These windows show the process and/or thread ID of the process or thread being viewed.

Breakpoints	Displays the Breakpoints List dialog box
Call Stack	Displays the Call Stack dialog box
Watches	Displays the Watch List dialog box
Local Variables	Shows the current function’s local variables while in debug mode
Threads	Displays the threads status box
Modules	Displays the Modules window
Event Log	Displays the event log window
CPU	Displays the CPU window
FPU	Displays the FPU window

View|Debug Windows|Breakpoints

Choose View|Debug Windows|Breakpoints to open the Breakpoint List window

The Breakpoint List window lists all currently set breakpoints.

Each breakpoint listing shows the following:

- For Source breakpoints, this is the file in which the breakpoint is set. For Data breakpoints set on a specific variable, this shows the name of the variable. For Address breakpoints, this is the address at which the breakpoint is set, unless the address can be correlated to a source line in which case this is the source filename. In that case, it shows the file in which the breakpoint is set.
- For Source breakpoints, this is the line number of the breakpoint. For Data breakpoints, this is the length of the breakpoint. For Address breakpoints, this is empty unless the address can be correlated to a source line. If so, it shows the source line number and a hex offset of the address from the beginning of the line.
- Any condition associated with the breakpoint
- Any pass count associated with the breakpoint

View|Debug Windows|Call Stack

Choose View|Debug Windows|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|Debug Windows|Watches

Choose View|Debug Windows|Watches to open the Watch List.

The Watch List 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|Debug Windows|Local Variables

See also

Choose View|Debug Windows|Local Variables to show the current function's local variables while in debug mode.

This command is always available, but the view will be empty unless the debugger is paused. 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.

Local Variables context menu

Inspect	Displays information about the currently selected variable in the Inspector Window.
Stay On Top	Keeps the Local Variables window visible, even when it does not have focus.
Dockable	Allows the Local Variables window to be docked to other windows in the IDE.

View|Debug Windows|Threads

Choose View|Debug Windows|Threads to view the Threads status box which displays the status of all the threads currently executing in the application being debugged. Multiple process debugging is also supported via the Threads view.

View|Debug Windows|Modules

See also

Choose View|Debug Windows|Modules to open the Modules window to view a list of modules that are loaded into memory when the current project is run. A module can be an executable file, DLL, or package that the current project needs loaded into memory during runtime. The Modules window shows each module's name, its runtime image base address, and a path indicating the location from which the module is loaded.

Normally, you would open this window after you have compiled a project and are debugging it. It is also helpful when optimizing to improve load time by specifying preferred image base offsets for each required module.

Note: The runtime image base address is the memory offset, in hexadecimal, where the module actually loads, as distinct from the preferred image base address you may have specified in the Project Options window.

The Modules window has three parts:

- Module pane (upper left)
- Source pane (lower left)
- Entry point pane (right)

The Module pane lists each module name and the address at which it is loaded. If the module has debug information, the Source pane shows a tree view of the source files that contain code that was used to build the module. If the module has debug information, the Entry point pane shows a list of all global symbols. If the module does not have debug information, the Entry point pane lists the function entry points into the module.

Multiple process debugging

For multiple-process debugging each process and its associated modules are shown. The current process is denoted by a green arrow glyph.

View|Debug Windows|Event Log

Choose View|Event to display the event log. The event log shows process control messages, breakpoint messages, and window messages. Right-click on the event log to display the context menu to clear the event log, save the event log to a text file, add a comment to the event log, and set options for the event log.

By setting options, you can control how many messages to display and what kind of events to show.

View|Debug Windows|CPU

Choose View|Debug Windows|CPU to display the CPU window for debugging a specific low-level aspect of an application such as a contents of the program stack, registers or CPU flags, memory dumps, or assembly instructions disassembled from the application's machine code.

View|Debug Windows|FPU

Choose View|Debug Windows|FPU to display the FPU window. You use the FPU window to view the contents of the FPU component of the CPU. You can display either floating-point information or MMX information.

The FPU window displays values and status for each register in the FPU as well as the FPU status, control, and tag words. The flags encoded in the control and status word are displayed in separate panes. You can also view the address, opcode, and operand that corresponds to the last FPU instruction executed.

View|New Edit Window

Choose View|New Edit Window to open a new Code editor window 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|Toolbars

See also

Choose View|Toolbars to show or hide the following in the IDE:

Toolbar	Icons on toolbar by default
Standard	New, Open, Save, Save All, Open Project, Add File to Project, Remove File from Project
View	View Unit, View Form, Toggle Form/Unit, New Form
Debug	Run, Pause, Trace Into, Step Over
Custom	Any commands you add. Contains Help Contents, by default; see View|Toolbars|Customize.
Component palette	Tabbed pages of commonly used objects
Desktops	Displays the Desktops toolbar which includes a pick list of the available desktop layouts and icons to let you Save Current Desktop and Set Debug Desktop.

Check the items you want to display and uncheck those you want to hide. The toolbars provide icons as shortcuts for actions you can perform. The Component palette has several tabs each of which displays icons representing components you can use to design your application.

You can also customize all of the toolbars adding or removing items from the toolbars according to your needs.

View|Toolbars|Customize (Customize dialog box)

See also

Choose View|Toolbars|Customize to change the toolbar configuration. The Customize dialog box is displayed.

The pages of the Customize dialog box are

- Toolbars
- Command
- Options

These pages let you customize which toolbars are displayed, what commands are on the toolbars, and how tooltips are displayed.

When any of the pages of the Customize dialog box is displayed, you can delete or rearrange any of the buttons currently on the toolbar. However, none of the buttons on the toolbar is active.

Toolbars **(Customize dialog box)**

See also

The Toolbars page of the Customize dialog box lets you choose which toolbars to display. It includes a Reset button that you can use to return any toolbar to its default (factory) configuration.

Check all the toolbars that you want to display.

Reset Select one or more of the toolbars to reset (multiselect using Ctrl or Shift). Then choose Reset to reset the toolbars to the default configuration (deleting any added buttons and adding any deleted buttons). The Reset button is only active when one or more toolbars is selected.

When the Toolbars page of the Customize dialog box is displayed, you can delete or rearrange any of the buttons currently on the toolbar. However, none of the buttons on the toolbar is active.

Commands (Customize dialog box)

See also

The Commands page of the Customize dialog box lets you add or delete buttons on the toolbar. It has two list boxes:

Categories	Select a menu whose commands you want to add as buttons to the toolbar. The commands associated with each category are shown in the Commands list box.
Command	Drag and drop a command from this list box onto the toolbar. The Commands list box displays all the commands available for the category selected in the Categories list box. The icon to the left of the menu command shows the button that will appear on the toolbar.

Choose Reset on the Toolbars page of the Customize dialog box to reset any toolbar to its default configuration (deleting any added buttons and adding any deleted buttons).

When the Commands page of the Customize dialog box is displayed, you can delete or rearrange any of the buttons currently on the toolbar. However, none of the buttons on the toolbar is active.

Options **(Customize dialog box)**

The Options page of the Customize dialog box lets you choose whether or not to display tooltips for toolbar buttons. You can include shortcut keys in the tooltip text or not. Choose the options you prefer.

When the Options page of the Customize dialog box is displayed, you can delete or rearrange any of the buttons currently on the toolbar. However, none of the buttons on the toolbar is active.

View|Desktops

See also

The View|Desktops command lets you switch to the various desktop layouts you have saved. The top part of the submenu that displays when you select View|Desktops lists the available layouts.

View|Desktops lets you access the following commands:

- Save Desktop
- Delete
- Set Debug Desktop

You can also use the Desktops toolbar in the IDE to select the desktop layout you want. It includes a pick list of the available desktop layouts and icons to let you save the current desktop or make the current desktop the debugging desktop.

View|Desktops|Save Desktop

See also

The View|Desktops|Save Desktop command lets you customize and save the current layout of the Delphi desktop.

To save the current desktop:

1. Arrange the desktop as you want it including displaying, sizing, and docking particular windows, and placing them where you want on the display.
2. Select View|Desktops|Save Desktop (or click the Save current desktop icon on the Desktops toolbar).
3. Type a name for this particular desktop layout and click OK.

This layout will remain in effect for all projects and will be used when you next start Delphi.

You can also save a desktop by arranging it as you want it, typing the name directly into the combo box in the Desktops toolbar and press Enter. The current desktop layout is saved under the new name.

You can create as many layouts as you like. The names are added to the View|Desktops submenu and to the pick list on the Desktops toolbar. To change to one of the saved desktop layouts, select another choice in either location.

Save Desktop dialog box

See also

You use this dialog box to type a name for this particular desktop layout and click OK. You can select the saved customized desktop setting from the Desktops toolbar or by selecting View|Desktops and the name of the desktop you want to use.

This dialog box is displayed when you select View|Desktops|Save Desktop (or click the Save current desktop icon on the Desktops toolbar).

View|Desktops|Delete

See also

The View|Desktops|Delete command displays a list box where you can delete any of the desktop layouts you have saved.

To delete a customized desktop setting:

Select the desktop you want to delete and click OK.

Delete Desktop dialog box

See also

You use this dialog box to select one or more desktop settings to delete and click Delete.

This dialog box is displayed when you select View|Desktops|Delete Desktop.

View|Desktops|Set Debug Desktop

See also

The View|Desktops|Set Debug Desktop command lets you select one of the desktop layouts you have saved as the one to use during runtime and debugging. A dialog box listing the layouts you can choose from is displayed.

To set the debug desktop:

Select the desktop you want to use for debugging and click OK. The debug desktop is automatically displayed during all debug sessions.

Note When the debug session ends, the current desktop reverts to the last desktop you were using before the debug session began.

Set Debug Desktop dialog box

See also

You use this dialog box to select one of your desktop settings to use during runtime and debugging. Select the desktop you want to use for debugging and click OK.

This dialog box is displayed when you select View|Desktops|Set Debug Desktop.

View|Type Library

See also

The Type Library editor lets you examine and create type information for ActiveX controls, Automation servers, MTS objects, and other COM objects. You can provide type information with an object either stand-alone in a type library (.TLB) file or you can integrate it into the EXE or ActiveX library as a resource.

By including the type library with an application or ActiveX library, you are making information contained in the library, such as its object interfaces, properties, methods, and events, available to other to other applications and programming tools.

When you use the wizards to create an ActiveX control or Automation object, a type library is automatically created for you. You can then use the Type Library editor to examine or modify the type information created by the wizard. Use the Type Library editor to add additional functionality, such as new properties, methods, or events, to your type library.

Type Library editor

See also

Use the Type Library editor to make changes to your type library. The Type Library editor generates the required IDL syntax automatically. Any changes you make in the editor are reflected in the corresponding control.

To open the Type Library editor,

- Choose View|Type Library.

The Type Library option is available only for projects that contain a type library. An ActiveX control and Automation object will contain a type library if you create it using a wizard.

The main elements of the Type Library editor are:

- Toolbar

These buttons add instances of new information types to the current type library. When you click a button, its icon is added to the object list pane.

- Status Bar pane

Syntax and translation errors and warnings appear here when you edit, save, or load a type library.

- Object list pane

Each instance of an information type in the current type library appears in the object list, represented by a unique icon. Select an icon to see its data pages displayed in the information pane at the right.

- Pages of type information. For each object selected in the object list pane, the Type Library editor displays tabbed pages of the object's attributes, flags, if any, and text. Some objects have additional pages.

You can view and edit the following Type Library information:

- Type library information
- Interface pages
- Dispatch type information
- CoClass pages
- Enumeration type information
- Alias type information
- Record type information
- Union type information
- Module type information

Type Library attributes

Edit the attributes in the Type Library to change information about the type library itself.

To edit the type library attributes,

- In the Type Library Editor object list pane, select the type library, which has the icon: 

The attributes page displays type information about the currently selected type library.

Type Info attributes

Type info is a general term for the types of information available in a type library such as interfaces, dispinterfaces, coclasses, enumerations, and so on. You can change information about each type by modifying its attributes page.

To edit type info attributes,

- In the Type Library Editor object list pane, select the type you want to edit and its attributes page appears.

See the following topics for details on modifying the attributes of each type info:


Interface attributes


Dispinterface attributes


CoClass attributes


Enumeration attributes


Alias attributes


Record attributes


Union attributes


Module attributes

Member attributes

You can add or remove members to each type info.

To modify type member attributes,

1. In the Type Library Editor object list pane, select the type you want create or modify.
   The toolbar displays the members available for this type to the left of the vertical bar.
2. Choose the member you want to modify and its attributes page appears.

For interfaces and dispinterfaces, you can add methods or properties.

For enumerations, you can add constants.

Records, are comprised of fields.

A union defines a C-style union.

A module can be comprised of methods and constants.

Text

All type library elements have a text page that displays the IDL syntax for the element. Any changes you make in other pages of the element are reflected here. If you add IDL code directly in the text page, changes are reflected in the other pages of the Type Library editor.

The Type Library editor generates syntax errors if you add IDL identifiers that are currently not supported by the editor; the editor currently supports on those IDL identifiers that relate to type library support (not RPC support).

CoClass implements

You specify the globally unique identifier (GUID) and supported interfaces on the CoClass implements page.

For each interface, specify the following information.

Type library uses

The uses page lists any other type libraries that this type library references.

Parameters

The parameters page allows you to specify the parameters and return values for the functions contained in your type library.

You can specify the following information.

Type library flags

The type library flags specify how other applications must use the server associated with this type library. You can set the following flags:

Flag	Meaning
Restricted	Prevents the library from being used by a macro programmer.
Control	Indicates that the library represents a control.
Hidden	Indicates that the library exists but should not be displayed in a user-oriented browser.

Type info flags

You can set the following flags for type info. The same flags menu appears for all type info (interfaces, CoClasses, enumerations and so on). Unavailable flags are dimmed.

For details, see:

Interface and dispinterface flags

CoClass flags

Type member flags

The following menu of flags appears when you have a type member (properties, methods, constants, and fields). Unavailable flags are dimmed.

For details, see

Method flags

Property flags

Parameter flags

The parameters flags dialog box allows you to specify parameters and return values for your functions.

To get to this dialog box,

1. In the Type Library Editor, select a method or property.
2. In the Parameters page, click in the flags field.
   An ellipsis appears.
3. Click on the ellipsis to display this Parameter Flags dialog box.

For details on the flags, see Property and method parameters page

Remote Data Module wizard

See also

Use the Remote Data Module wizard to create a data module that can be accessed remotely as a dual-interface Automation server. A remote data module resides in the application server between a client and server in a multi-tiered database environment.

To bring up the Remote Data Module wizard:

1. Choose File|New to open the New Items dialog box.
2. Choose the tab labeled Multitier.
3. Select the Remote Data Module item in the list view.

Remote Data Module wizard options

CoClass Name

Enter the base name for the Automation interface of your remote data module. The class name for your remote data module (a descendant of TRemoteDataModule) will be this name with a T prepended. It will implement an interface named using this base name with an I prepended. To enable a client application to access this interface, set the ServerName property of the client application's connection component to the base name you specify here.

Instancing

Use the instancing combo box to indicate how your remote data module application is launched. The following table lists the possible values:

Value	Meaning
Internal	The remote data module is created in an in-process server. Choose this option when creating a remote data module as part of an active Library (DLL).
Single Instance	Only a single instance of the remote data module is created for each executable. Each client connection launches its own instance of the executable. The remote data module instance is therefor dedicated to a single client.
Multiple Instance	A single instance of the application (process) instantiates all remote data modules created for clients. Each remote data module is dedicated to a single client connection, but they all share the same process space.

Threading Model

If you are creating the remote data module in an active library (DLL), use the threading combo box to indicate how client calls are passed to your remote data module's interface. The following table lists the possible values:

Value	Meaning
Single	The library only receives one client request at a time. Because all client requests are serialized by COM, you don't need to deal with threading issues.
Apartment	Each instance of your remote data module services one request at a time. However, the DLL may handle multiple requests on separate threads if it creates multiple COM objects. Instance data is safe, but you must guard against thread conflicts on global memory. This is the recommended model when using BDE-enabled datasets. (Note that when using BDE-enabled datasets you must add a session component with AutoSessionName set to True.)
Free	Your remote data module instances can receive simultaneous client requests on several threads. You must protect instance data as well as global memory against thread conflicts. This is the recommended model when using ADO datasets.
Both	The same as Free except that all callbacks to client interfaces are serialized.

MTS Data Module wizard

See also

Use the MTS Data Module wizard to create the server in a multi-tiered database application that uses MTS. MTS data modules must exist within an Active Library (DLL), so that they can be instantiated as needed by the MTS proxy.

To bring up the Remote Data Module wizard

1. Choose File|New to open the New Items dialog box.
2. Choose the tab labeled Multitier.
3. Select the MTS Data Module item in the list view.

The MTS Data Module wizard Options

CoClass Name edit control

Enter the base name for the Automation interface of your remote data module. The class name for your remote data module (a descendant of TMTSDataModule) will be this name with a T prepended. It will implement an interface named using this base name with an I prepended. To enable a client application to access this interface, set the ServerName property of the client application's connection component to the base name you specify here.

Threading model

Use the threading combo box to indicate how client calls are passed to your remote data module's interface. The following table lists the possible values:

Value	Meaning
Single	The data module only receives one client request at a time. Because all client requests are serialized by MTS, you don't need to deal with threading issues.
Apartment	Any data module instance receives one request at a time, but not always on the same thread. Instance data is thread-safe, but global memory must be explicitly protected against thread conflicts.
Free	For MTS data modules, Free is the same as Apartment.
Both	The same as Apartment except that all callbacks to client interfaces are serialized.

Transaction Model

Use the Transaction Model radio buttons to indicate the MTS transaction attributes of your application server interface. The following table lists the possible values:

Value	Meaning
Requires a transaction	Every time a client calls the remote data module's interface, the call is enlisted in an MTS transaction. If the caller supplies a transaction context, a new transaction need not be created.
Requires a new transaction	Every time a client calls the remote data module's interface, a new transaction context is automatically created for that call.
Supports transactions	The data module can be enlisted in an MTS transaction, but the client must supply the transaction context.
Does not support transactions	The data module interface can't participate in MTS transactions.

CORBA Data Module wizard

See also

Use the CORBA Data Module wizard to create a data module that can be accessed remotely by CORBA clients. A CORBA data module resides in the application server between a client and server in a multi-tiered database environment.

To bring up the CORBA Data Module wizard:

1. Choose File|New to open the New Items dialog box.
2. Choose the tab labeled Multitier.
3. Select the CORBA Data Module item in the list view.

CORBA Data Module wizard options

Class Name

Enter the base name of the object that implements the CORBA interface for your remote data module. The class name for your remote data module (a descendant of TCorbaDataModule) will be this name with a T prepended. It implements an interface named using this name with an I prepended. To enable a client application to access this remote data module, set the RepositoryID property of the client application's CORBA connection component to the base name you specify here.

Instancing

Use the instancing combo box to indicate how your CORBA server application creates instances of the CORBA data module. The following table lists the possible values:

Value	Meaning
Instance-per-client	A new CORBA data module instance is created for each client connection. The instance persists until a timeout period elapses with no requests from the client. This allows the server to free instances when they are no longer used by clients, but runs the risk that a CORBA data module may be freed prematurely if the client does not use the data module's interface often enough.
Shared Instance	A single instance of the CORBA data module handles all client requests. Because all clients share the single instance, it must be stateless.

Threading

Use the threading combo box to indicate how client calls invoke your remote data module's interface. The following table lists the possible values:

Value	Meaning
Single-threaded	Each data module instance is guaranteed to receive only one client request at a time. Instance data is safe from thread conflicts, but global memory must be explicitly protected.
Multithreaded	Each client connection has its own dedicated thread. However, the data module may receive multiple client calls simultaneously, each on a separate thread. Both global memory and instance data must be explicitly protected against thread conflicts.

CORBA Object wizard

See also

Use the CORBA Object wizard to create a server that can be accessed remotely by CORBA clients.

To bring up the CORBA Object wizard:

1. Choose File|New to open the New Items dialog box.
1. Choose the tab labeled Multitier.
1. Select the CORBA Object item in the list view.

The CORBA Object wizard Options

Class Name

Enter the base name of the object that implements the CORBA interface for your remote data module. The class name for your CORBA object (a descendant of TCorbaImplementation) will be this name with a T prepended. It implements an interface named using this name with an I prepended. To enable a client application to access this remote data module, use the CreateInstance method of the stub factory class that is automatically generated in the _TLB unit.

Instancing

Use the instancing combo box to indicate how your CORBA server application creates instances of the CORBA object. The following table lists the possible values:

Value	Meaning
Instance-per-client	A new CORBA object instance is created for each client connection. The instance persists until a timeout period elapses with no requests from the client. This allows the server to free instances when they are no longer used by clients, but runs the risk that the CORBA object may be freed prematurely if the client does not use its interface often enough.
Shared Instance	A single instance of the CORBA object handles all client requests.

Threading

Use the threading combo box to indicate how client calls invoke your CORBA object's interface. The following table lists the possible values:

Value	Meaning
Single-threaded	Each CORBA object instance is guaranteed to receive only one client request at a time. Instance data is safe from thread conflicts, but global memory must be explicitly protected.
Multithreaded	Each client connection has its own dedicated thread. However, the CORBA object may receive multiple client calls simultaneously, each on a separate thread. Both global memory and instance data must be explicitly protected against thread conflicts.

Link not found

The topic you requested is either not available or not linked to this Help system. This can occur if you launched this Help file from a system on which Delphi has not yet been installed, or if the subject matter you are requesting is not available in your edition of Delphi.

Borland INPRISE

The topic you requested is now loading. If it does not appear within a few seconds, the topic is either not available or not linked to this Help system. This can occur if you launched this Help file from a system on which Delphi has not yet been installed, or if the subject matter you are requesting is not available in your edition of Delphi.

About the Object Inspector

See also

The Object Inspector is the connection 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.
- Filter visible properties and events.

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 the object type of the selected component. This lets you quickly display properties and events for the 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

Object Inspector tabs provide a means to switch between the Property page and the Events page of the Object Inspector. To change pages, click a tab.

You can display and filter properties and events by category. By filtering the properties, you can reduce the number of properties visible in the Object Inspector and focus on those which are primarily of interest at the time. You can also more easily locate related properties by viewing them by category. For example, when localizing your application for other countries, you can display only properties that need to be localized by unchecking all categories except Localizable. See Property and event categories in 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. By setting properties at design time you are defining the initial state of a component. You can set runtime properties by writing source code within event handlers.

The Properties page displays the properties of the component that is selected on the form.

If the Properties are arranged by name, the first column on the Property page lists the names of the selected component's published properties:

If a plus sign (**+**) appears beside a property name, this can be clicked to display the sub-properties of that property. These can be a list of possible values when the property represents a set of flags (the value column lists the set enclosed in square brackets []), or subproperties if the property represents an object (the value column gives the name of the object, enclosed in parentheses). Similarly, if a minus sign (**-**) appears, this can be clicked to collapse the subproperties. When a property has focus, you can also use the keyboard + and – keys to expand or collapse properties.

The second column on the Property page displays the property values:

When the property is selected, the value changes to an edit control where you can type a new value.

If the value can be set using a dialog, an ellipses button (...) appears when the property is selected. Click this button to display a dialog where you can set the property. You can also display the dialog by double-clicking.

If the value is an enumerated type, a drop-down button appears when the property is selected. Click this button to display a drop-down list that you can use to set the property. You can see images in the drop-down lists for properties that include images such as cursors, brush types, colors, and image lists. To view images referenced by the ImageIndex property, you need to set the property that holds the image list to the image list containing the images.

If the value is another component, you can shift the Object Inspector's focus to that component by holding down the Ctrl key while double-clicking. For example, if the DataSet property of a data source is set to Table1, Ctrl-double-clicking on Table1 in the value column displays Table1's properties in the Object Inspector.

If you arrange the properties by category (right-click Arrange|by Category), the categories are listed alphabetically. You can view properties associated with each category by clicking the **+** sign. For more information, see Property and event categories in the Object Inspector.

The Properties page displays the published properties of the selected component. For more information, see How the Object Inspector displays properties.

Events page

See also

The Events page of the Object Inspector enables you to connect forms and components to program events. To generate a default event handler for an event, double-click the right column. Delphi creates the event handler and switches focus to the Code editor. In the Code editor, you write the event handlers that specify how a component or form responds to a particular event.

When you select an event, the value column can display a drop-down list of existing event handlers that can respond to the event. Choose one of these existing event handlers if you write one event handler to respond to multiple events.

The Events page displays only the published events of the component that is selected in the form.

How the Object Inspector displays properties

See also

The Object Inspector dynamically changes the set of properties it displays, based on the component selected. Only the shared properties are displayed. For example, if you select a Label and a GroupBox, you'll see the property Color along with other properties. If you select a Label and then a Button, the choice for Color goes away because Color is not a property for buttons. 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 values displayed are 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 those values in all the selected components.

  There is one exception to this: when you select multiple components in a form, the 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 name to more than one component in a form.

See Property and event categories in the Object Inspector for how to filter properties and arrange them by category.

Tabbing to specific properties

See also

You can jump directly to a property in the Object Inspector by pressing the Tab key followed by any alphabetic character. The cursor jumps to the Property column of the first property beginning with the typed letter. (The Object Inspector lists property names alphabetically.)

To tab to a specific property (in this case, *Width*),

1. Select the form.
2. In the Object Inspector, select the form's AutoScroll property.
3. Press Tab, W to jump directly to the Width property.
4. Press Tab again to place the cursor in the Value column, where you can begin entering your edits.

Pressing Tab acts as a toggle between the Value column and the Property column. Whenever you are in the Property column, pressing an alphabetic character jumps you to the first property starting with that character.

Changing component properties

See also

You can change component properties at design time or dynamically when the application runs. You can also view a form as a text file and make changes that will be reflected in the Object Inspector.

To change a component property at design time,

1. Select the component in the form or with the Object selector.
2. Select the property that you want to change by selecting it from the Properties page.
3. Type a new value for that property.

To change a component property at runtime,

1. Select the component in your source code using the Code editor. (For example, *Form1*)
2. Select the property that you want to change (*Color*) and type a new value (*clAqua*).

   See the following example:

```
Form1->Color = clAqua;
```

See Property and event categories in the Object Inspector for how to filter properties and arrange them by category.

Displaying and setting shared properties

See also

You can set shared properties to the same value without having to individually set them for each component.

To display and edit shared properties,

1. In the form, Shift+click to select the components whose shared property you want to set.

   The Properties page of the Object Inspector displays only those properties that the selected components have in common. (Notice, however, that the Name property is no longer visible because each component must have a unique name.)
2. With the components still selected, use the Object Inspector to set the property.

See Property and event categories in the Object Inspector for how to filter properties and arrange them by category.

Property and event categories in the Object Inspector

You can display and filter properties and events by category in the Object Inspector.

Filtering properties or events

To change the filter, right-click, choose View, and check or uncheck the categories that are listed on the menu. Properties associated with checked categories are visible in the Object Inspector.

Note: Legacy properties (such as Ctl3D and OldCreateOrder) are not visible, by default.

Displaying properties or events by category

To display properties by category, right-click and choose Arrange|by Category. The categories are listed alphabetically. You can collapse or expand the categories by clicking the **+** or **–** collapse icon and the state is persistent until you change it.

Note: Some properties occur in multiple categories. If you change the value under one category, the value changes consistently in all places.

Displaying properties alphabetically

To redisplay properties alphabetically, right-click and choose Arrange|by Name. The categories are no longer visible in the Object Inspector. Properties that are visible are listed alphabetically.

Component writers can create categories and assign properties to categories using the RegisterPropertyInCategory procedure. See Property categories: functions and classes for details.

Viewing nested properties

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.

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 in the Value column for any properties that has been declared as TStrings.
- Double-click the word (TStrings) in the Value column for any property that has been declared as TStrings.

The String List context menu contains the following commands:

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.

Load String List Dialog Box

See also

Use the Load String List dialog box to select a text file to load into a property of type TStrings.

To open this dialog box,

1. Bring up the String List editor.
2. Right-click and choose 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 this dialog box,

1. Bring up the String List editor.
2. Right-click and choose 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) 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 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

See also

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.

The Picture editor provides the following commands:

Load

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

Display the Save Picture As dialog box, where you can specify a directory and file name in which to store the image.

Clear

Remove the association between the current image and the selected component.

Loading an image at design time

See Also

Use the 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 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 a 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.)

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.

Save Picture As dialog box

See also

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

See also

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.

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.

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.

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.

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 runtime.

Edit Mask label

Displays the mask definition for the current 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 (in points). This combo box displays only those font sizes that are available for the selected font.

Effects

Check these options to make the text strike-through 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.

Script

Lists the available language scripts for the selected font.

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.

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.

Select whether or not you want to create a new file using the associated OLE server or use an existing file. If you use an existing file, it must be associated with an application that can act as an OLE server.

Create New

Choose Create New to specify that you want to launch a server application to create a new OLE object. After choosing Create New, the ObjectType list box is displayed.

Create From File

Choose Create From File to specify that the OLE object has already been saved as a file. After choosing Create From File, the File, Browse and Link controls are displayed.

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.

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.

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 spreadsheet.

When the server is a VCL-based 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 Filter 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 (;).

Open dialog box

See also

Use the Open dialog box at design time to load a multimedia file into the MediaPlayer component.

To open the Open dialog box,

1. Place 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. By default, 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.

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 display the ListView Items editor,

- Select the TListView object and 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 display the TreeView Items editor:

- Select the TTreeView object and 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).

Collection Editor dialog box

See also

The Collection Editor dialog box is used to edit the items maintained by a collection object. A collection object is a descendant of TCollection. The Collection Editor displays information about the items in the collection, and allows you to add, remove, or rearrange the individual items. For some types of collection, additional buttons are provided to allow other manipulations of the list.

The items displayed in the list window of the Collection Editor can be selected using the mouse. Once an item is selected, its properties and events can be set using the object inspector. The following table indicates what properties of the items are displayed in the list window for each type of collection item, and how the collection is used.

Collection	Item type	Properties displayed	Use
TCheckConstraints	TCheckConstraint	ImportedConstraint, or, if no ImportedConstraint is blank, CustomConstraint	Represents record-level constraints for the data in a BDE dataset. (Constraints property)
TCoolBands	TCoolBand	Text	Represents a set of bands in a CoolBar component. (Bands property)
TDBGridColumns	TColumn	FieldName	Represents the field binding and display properties of a column in a data-aware grid. (Columns property)
THeaderSections	THeaderSection	Text	Represents the display properties of the sections in a HeaderControl object. (Sections property)
TListColumns	TListColumn	Caption	Represents the columns of a report-style List View component. (Columns property)
TStatusPanels	TStatusPanel	Text	Represents the individual panels of a StatusBar component. (Panels property)
TWebActionItems	TWebActionItem	Name, PathInfo, Enabled, and Default	Represents the action items that create the responses to HTTP request messages for a Web dispatcher or Web module. (Actions property)

To display the Collection Editor dialog box, first place the component that uses the collection on a form. Select the property that is implemented using the collection (listed in parentheses in the preceding table), and click on the ellipsis. For some components, the Collection Editor may also be displayed by right-clicking the component, and selecting the appropriate editor from the context menu.

Dialog box options

Item list
The Item list displays the properties listed in the third column of the preceding table for each item in the collection. The properties for a selected item are displayed in the Object Inspector and are edited there.

Add button
Adds a new item to the collection. You can select the item and edit its parameters in the Object Inspector.

Delete button
Removes the selected item from the collection.

Move Up/Down buttons
Change the order of the items. For most collections, the order determines the order in which items are displayed or used by the object that maintains the collection.

Add All Fields button (TDBGridColumns only)
Add a column for every field in the dataset to which the data-aware grid is bound. This button is only enabled if the data-aware grid is bound to an active dataset.

Restore Defaults button (not for all collections)
Restore the default properties (obtained from the field component) of the currently selected column. This button is enabled if the currently selected column is bound to a field (the FieldName property is set).

Read From Dictionary button (TCheckConstraints only)
Add a CheckConstraint object for every record-level constraint in the data dictionary. Each CheckConstraint object will have its ImportedConstraint property set to the constraint from the dictionary.

ImageList Editor

Use the ImageList Editor at design time to add bitmaps and icons to a TImageList component.

While working in the image list editor, you can click Apply to save your current work without exiting the editor, or click OK to save your changes and exit the dialog. Using the Apply button is especially useful because once you exit the dialog, you can't make any more changes to the existing images.

To display the ImageList editor:

- Select the TImageListobject and double-click the component or right-click and select ImageList Editor.

Selected Image

The selected image control displays the currently selected image. This image can be changed by clicking on another image in the Images list view below. When an image is selected, you can delete it from the list of images. If the image was not added to the image list before the current invocation of the editor, you can use the other controls to alter its properties. However, once the image list editor is closed, these properties are fixed and the selected image controls are grayed if the ImageList Editor is again displayed and that image is selected.

Transparent color

Use the Transparent color drop-down to specify which color is used to create a mask for drawing the image transparently. The default transparent color is the color of the bitmap's left-most pixel in the bottom line. You can also change the transparent color by clicking directly on a pixel in the selected image.

When an image has a transparent color, any pixels in the image of that color are not rendered in that color, but instead appear transparent, allowing whatever is behind the image to show through.

If the image is an icon, Transparent color appears grayed and the transparent color is set to clNone. This is because icons are already masked.

Fill color

Use the Fill color drop-down to specify a color that is added around the edges of the selected image when it is smaller than the dimensions indicated by the Height and Width properties of the image list control.

This control is grayed if the selected image completely fills the dimensions specified by the image list (that is, if it is at least as big as the Height and Width properties). This control is also grayed for icon images, because icons act like masks with any outer boundaries transparent.

Options

Use the Options radio buttons to indicate how the image list should render the selected image if it does not fit exactly in the dimensions specified by the image list's Height and Width properties. (These buttons are disabled for icons)

Setting	Description
Crop	Displays the section of the image beginning at the top-left, extending the image list width and height towards the bottom-right.
Stretch	Causes the entire image to stretch so that it fits the image list width and height
Center	Centers the image within the image list width and height. If the image width or height is larger than the imagelist width or height, the image may be clipped.

Images

Contains a preview list view of all the images in the image list, and controls for adding or deleting images from the list. Each image is displayed within a 24x24 area for easier viewing of multiple images. Beneath each image is a caption that indicates the zero-based position of the image within the image list. You can edit the caption to change an image's position in the list or drag the image to its new

position.

Add
Displays the Add Images dialog box, which lets you select one or more bitmaps or icons to add to the image list. The images then appear highlighted in the preview list view and their captions are assigned sequential values in the image list.

If a bitmap is larger than the image list width or height by even increments, a prompt appears asking whether the ImageList editor should divide the bitmap into several images. This is useful for toolbar bitmaps, which are usually composed of several small images in a sequence and stored as one larger bitmap.

Delete
Removes the selected images from the image list. All images left after clicking Delete are repositioned so they are a contiguous zero-based list.

Clear
Removes all images from the imagelist.

Export
Allows you to save the selected image to a file. This file contains the bitmap in it’s currently altered state, including any cropping or stretching.

Action List editor

Use the Action List editor at design time to add actions to a TActionList component.

To display the Action List editor:

- Select the TActionListobject and double-click the component or right-click and select Action List editor.

Tool bar

At the top of the Action List editor is a tool bar containing 4 buttons. These are as follows

Button	When clicked
New Action	Inserts a new action into the list. By clicking the drop-down arrow next to the button, you can choose whether to add a new action that you define, or a standard (predefined) action. If you choose Standard Action, you will be presented with a dialog where you can choose the predefined action.
Delete	Deletes the action currently selected in the list boxes.
Arrow buttons	Moves the currently selected action up or down to change its position in the list.

Right click the Tool bar to display the ActionList tool bar context menu. This contains one item:

Command	When clicked
Text labels	Displays or hides the labels on the buttons in the toolbar.

List boxes

The lower portion of the Action List editor contains two list boxes that represent the current list of actions. The first list indicates the value of the Category property of the action. You can change this value by selecting the action and changing the value of the Category property in the Object Inspector.

The second list indicates the name of the action. You can change this value by selecting the action and changing the value of the Name property in the Object Inspector.

Right click in the lower portion of the Action List editor to display the ActionList context menu. This contains the following items:

Command	When clicked
New Action	Adds a new (not predefined) action to the Action List editor. You can then use the object Inspector to edit its properties.
New Standard Action	Displays the Standard Actions dialog box, where you can select a predefined action.
Move Up	Moves the currently selected action toward the beginning of the list.
Move Down	Moves the currently selected action toward the end of the list.
Cut	Cuts the currently selected action to the clipboard, removing it from the list.
Copy	Copies the currently selected action to the clipboard without removing it from the list.
Paste	Pastes an action from the clipboard above the currently selected action.
Delete	Deletes the currently selected action.
Select All	Selects all actions in the list.
Panel Descriptions	Displays or hides labels over the listbox indicating their purpose.
Toolbar	Displays or hides the toolbar.

New Standard Action dialog box

Use the New Standard Action dialog box to add a predefined action to your action list. Standard actions perform common tasks such as navigating datasets, managing the windows in an MDI application, or working with the Windows clipboard. Each standard action performs a specific function when invoked, and enables or disables any linked controls as appropritate.

Choose the action you want to add from the list and click OK. For a description of each predefined action class, see Predefined Action Classes.

Remote debugging

The debugger supports remote debugging of EXEs, DLLs, and packages that are built with debug symbols. The remote debug server is not supported on Windows 95 or Windows 98 as a service. The main components of remote debugging are:

- The "client" IDE, which provides the UI for the debugging session (delphi32.exe).
- The "debug server" on the remote machine (bordbg50.exe). The server's function is to control the debuggee and interact, via a network connection, with the IDE. The debug server must have access to the debug kernel dll (bordbk50.dll) and an evaluator dll (dcc50.dll). To install you will need to run setup.exe located in the RDEBUG directory.

Starting the debug server

To start the server, you need either administration rights or debugging rights on the remote machine. The client IDE will not be able to connect to the remote debug server unless the latter is running.

There are two ways to start the debug server, manually, using the BORDBG50.EXE or via an NT service: On Windows 95 or 98, NT services are not available and the debug server must be started manually.

- To start the server manually, run `BORDBG50.EXE -listen` from the command line (the only option for Windows95/98).
- Installing the debug server as an NT service should be done via the remote debug setup program. You can use the Services applet from the Control Panel to check whether the "Borland Remote Debugging Service" is installed and running. (NT only).

Setting the client IDE

1. Enable “Include remote debug symbols” on the “EXE and DLL options” pane under Project|Options|Linker.
2. Under Remote tab of the Run|Parameters menu, set the Remote path to directory and EXE name as the server will see it.
3. Set remote host to the server machine name or IP address.
4. Click Debug project in the IDE.

Note: The server needs to be able to see the EXE and the symbols. If you run the server as a service, the server will not have access to network shared drives. In this case, you have to either copy the symbols and EXE to the server machine, or set the output directory to be on the server. IDE needs to find the source. For example:

- Set the Remote Path on the Remote tab of the Run|Parameters menu relative to the remote system, that is, as the remote system sees it.
- Set the Output path (on the Directories/Conditionals page of the Project Options dialog) relative to the local machine (IDE). Use the shared drive+path to identify this output directory.

Connecting to the remote machine

Before starting remote debugging the IDE needs to connect to the remote machine. To do this, it needs to specify a machine name and, optionally, a port number and password.

The local and remote machines must be linked by TCP/IP. Communication for remote debugging uses a TCP socket and relies on standard Internet name resolution to establish this connection (DNS). This means the local machine must be able to obtain an IP address for the remote machine. The command "nslookup name" utility can be used to confirm the IP address bound to a particular name. Note that the DNS and Microsoft networking names for a machine can be different. Both names can be obtained from the networking applet in the control panel.

The client IDE and debug server currently use port 8000 as a connection point.

BORDBG50.EXE has three command-line options:

1. -listen (have the server wait for a connection; non-service mode)
2. -install (installs the service (as an NT service only) does not start service)
1. -remove (removes the service, stopping it if necessary)

Using the server under Windows 95 or Windows 98

On Windows 95 or Windows 98, the Remote Debugging service can only be run as a program. To run as a program, run `Bordbg50.exe -listen` from the command line or a shortcut.

Multiple process debugging

The integrated debugger supports multiple process debugging on NT. You can select and debug a process in one of several ways:

Project Manager	Add the projects you want to debug to the Project Manager. You cannot compile while debugging multiple processes, so choose Project|Build All before you begin debugging.
Thread Status box	Choose View|Debug Windows|Threads to use the Thread Status box to set and change the current process. The green arrow indicates the current process. The blue arrow indicates non-current processes.
Run|Parameters dialog	Use the Run|Parameters to start a new process debugging.
Run|Attach to Process	Use the Run|Attach to Process to attach to an already running process.
Debug toolbar Run button	Use the Debug toolbar drop down Run button to select a process from the drop down list. Selecting the process will make it active.

Multiple process debugging includes remote debugging. For example, the IDE can be used to debug three processes on machine "A", two on machine "B", and one local process. For more information, see Remote debugging.

Because processes may share common files, you should always do a Project|Build All projects before starting a multi-process debugging session

Inspectors are associated with the thread that was active when they were created. When a thread terminates, only the inspectors that were created while the process was active are destroyed.

Multi-process debugger views

Most of the debugger views, from the View|Debug Windows menu, are multi-process aware and display the current process EXE name and thread ID in the caption of the windows. No additional information is show for single process, single-threaded processes. The thread ID is shown for single process, multi-threaded processes. The process name and ID is included for multiple processes; multi-threaded processes also include the thread ID.

While debugging multiple processes, you can temporarily set debugger options for specific processes from the Thread Status box. See Setting debugging options for specific processes.

Distributed debugging

See also

The integrated debugger supports distributed debugging on NT for both:

- COM Cross-Process Support
- CORBA Cross-Process Support

Cross-process debugging is not supported on Windows 95/98 because those operating systems do not have multiple-process debugging support.

Stack support

When both the source and target of a cross-process remote procedure call (RPC) are under debugger control (for both the COM and CORBA case), it is assumed that the stack of the source only includes a line pointing to the stack of the target. If the target is not under debugger control, the source thread's stack is augmented with any additional information that can be obtained from the event trace.

COM cross-process support

This support is provided to help developers debug processes that exchange COM cross-process remote procedure calls (RPCs). When you enable "COM Cross-Process Support" in the IDE, three features are activated: remote process attach, call tracing and cross-process stepping.

Cross-process attach

The debugger attempts to gain control of the target of a cross-process remote procedure call (RPC) when the call begins. If this attempt succeeds, the list of debugger processes under your control will increase by one. If the target process is on another machine, the attempt will succeed only if remote debugging has been enabled on that machine. The event log and the process/Thread Status box can be inspected to see whether the list of debuggee processes has grown. The "target" of a cross-process RPC is the server if the process originally being debugged was a client or the client if the process originally debugged was the server.

Call tracing

A trace of all cross-process RPC calls is added to the event log. Up to three entries are created for each call:

- a client-side entry indicating the client is calling the server
- a server-side entry indicating the server is starting to work on the call
- a client-side call indicating completion of the call

Each entry includes the IID (interface identifier) and the zero-based index of the method being called.

Cross-process stepping

When "Cross-Process Support" has been enabled, step-into operations will follow the "distributed" thread of control, rather than the actual thread, and thus may terminate in a process and thread other than the one where they originated. This behavior can be by-passed by issuing a "step-to-next-source" command (SHIFT-F7) rather than a simple Step-Into command (F7). The following conditions must occur for the step-into operation to complete in another thread:

- The debugger must be successfully attached to the target process
- The method being invoked must include debug information for the call-site being returned to

The stack view indicates that the source thread is blocked until completion of code executing in the target thread.

CORBA cross-process support

This support is intended to help debug applications that use the VisiBroker ORB. Enabling "CORBA Cross-Process Support" provides the same functionality described for COM Cross-Process Support and extends this support with two additional features:

- Trace-Only Processes
- ORB-Event Breakpoints

When setting a breakpoint on an event while debugging a CORBA application, it is useful to keep the Call Stack and Event Log windows open. They can provide useful information, such as indicating what initiated the event break.

ORB-event breakpoints

The event trace generated from CORBA remote procedure calls (RPCs) is more detailed than that available for COM RPCs. For a list of the ORB events that may be generated, see ORB Events.

Relevant information is included with each of these events; for example, the operation name and transaction id for receive/reply events, the interface and host name for bind events.

In addition to event tracing, you can stop on each occurrence of these events. The decision to stop can be augmented with additional conditions based on the information available for each event. For example, stop if the operation name is "X" or the host name is "Y".

Trace-only processes

The debugger may not always be able to obtain control of the process that is the target of a cross-process remote procedure call (RPC). This can happen if the target is executing on a non-Win32 platform on a host where remote debugging is not enabled, or because of access restrictions on the target process. In the case of a COM RPC, no information about the target will be available from the debugger. However, in the case of a CORBA RPC, an event stream is generated from the target even when that target cannot be brought under debugger control. Processes which cannot be debugged but which generate an event trace are called "Trace-Only Processes".

Information about trace-only processes is limited to their event log trace. In particular, no information about threads or libraries is available.

Operations on trace-only processes are as follows. If the process is stopped because of an event breakpoint, it may be continued or its current thread may be stepped. Stepping the current thread of a trace-only process is synonymous with running until a particular event occurs:

- If a StepInto command is issued and the current event is ClientPrepareRequest or ClientSendRequest the debugger will stop the process that services that request when it generates a ServerReceiveRequest event.
- If a StepOver command is issued and the current event is ClientPrepareRequest or ClientSendRequest the debugger will stop the current process when it generates a ClientSendRequestFailed, ClientSendRequestSucceeded, ClientReceiveReply, or ClientReceiveReplyFailed for the same request.

Because no stack is available for the current thread of a trace-only process, the stack of the thread that initiated the RPC activity will show additional entries that describe the status of the RPC call.

Exceptions

You can set the IDE under Tools|Debugger|Options, Language Exceptions to either handle (stop execution of the process) or ignore the following exceptions:

- VisiBroker exceptions
- Internal exceptions
- CORBA system exceptions
- CORBA user exceptions

Setting debugging options for specific processes

When debugging multiple processes, you can set process-specific options.

To set a particular process's debugging options:

1. Run the application containing the process you want to debug.
2. Choose View|Debug Windows|Threads to display the Thread Status box.
3. Select the process for which you want to set local debugging options.
4. Right-click and choose Process Properties.

   The Temporary Process Options dialog box is displayed where you can set debugger options similar to setting them globally using Tools|Debugger Options. Those options that are relevant to debugging specific processes are included in the Temporary Process Options dialog box.

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
- 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 runtime. 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

Manipulating strings in a list

See also

Quite often, 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

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.

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.

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.

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.Items.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.

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.

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.

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,

- Use the Assign method to 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.

Examples

The following example copies the items from a combo box into a memo:

```
Memo1.Lines.Assign(ComboBox1.Items);
```

The following example adds all the items from the combo box to the end of the memo:

```
Memo1.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).

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 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.

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 runtime, 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

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.

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 runtime,

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 runtime.
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.

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.

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].

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;
```

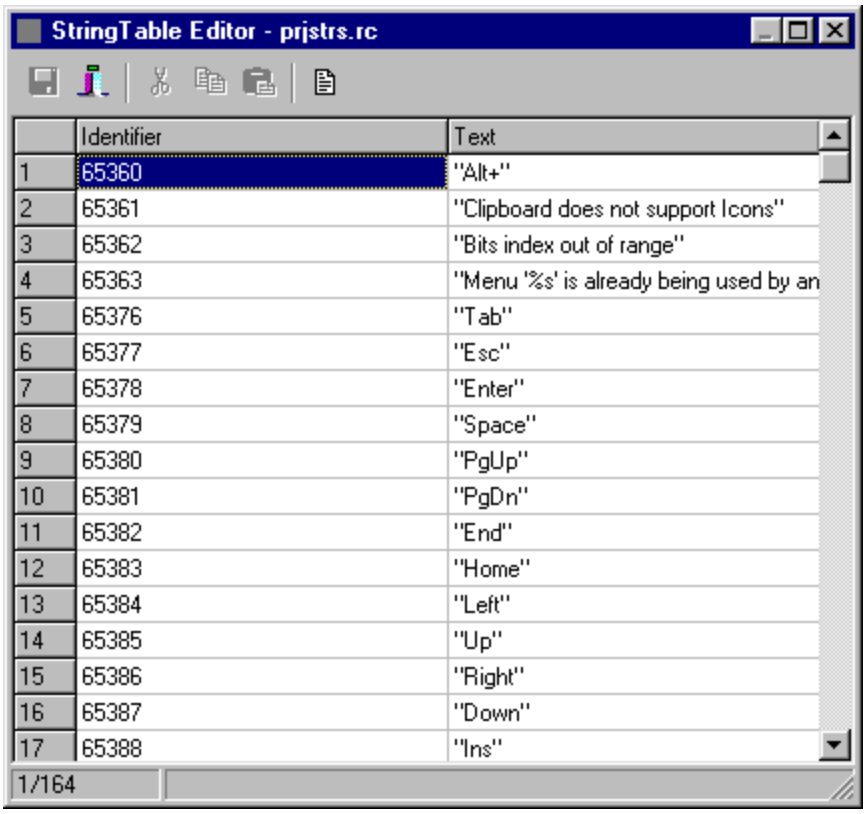
You can also add objects at the same time you add the strings, by calling the string list's AddObject method instead of Add, which just adds a string. AddObject takes two parameters, the string and the object. For example,

```
Fruits.AddObject('apple', AppleBitmap);
```

StringTable editor

See also

There are two columns in the StringTable editor, the Identifier column and the Text column. The Identifier column is read-only and lists the unique identifier of the string listed in the Text column. To edit a string in the Text column, just place the cursor in the cell you want to change and edit. The Text column is always in edit mode when you enter it.



Strings can contain escape sequences, but the StringTable editor will not accept invalid escape sequences. If the escape sequence is invalid, the StringTable editor will automatically insert the backslash character.

Note:

The strings in the Text column must be within quotation marks in order for you to edit the strings. The StringTable editor will not allow you to proceed until the string is formatted correctly with double quotes.

To edit longer strings, select the string and open the Multi-line editor. There are four ways of opening the Multi-line editor:

- Click the Multi-line editor button on the toolbar.
- Double-click the string's Identifier code.
- Select the string's Identifier code and press *Enter*.
- Press *Ctrl+E*.

To enter a carriage return in the Multi-line editor, press *Ctrl+Enter*. The code (/012) for a carriage return will be entered at the cursor position.

Saving .RC files in the StringTable editor

When you press the Save button on the toolbar, your data is saved and compiled. The editor calls Brcc32.exe to compile the .RC file. When Brcc32.exe is called to compile your .RC file, it uses your project's search path (Brcc32.exe equivalency: include path), conditional defines, and unit output directory (Brcc32.exe equivalency: output file name).

Note:

If you specify a unit output directory, the StringTable editor appends the name of the .RC file to the end of the unit output directory to complete the output file name.

Warning:

The StringTable editor always uses the .RC file name to generate the .RES file. So, if the .RC file names is File1.rc, the generated filename will be File1.res. If the names of the .RC and .RES files do not match in the $R directive, Delphi will use the incorrect .RES file. This is done to keep the compiled files in sync with the source files.

Watch List

To display the Watch List, choose View|Debug Windows|Watches. The Watch List displays the current value of the watch expression based on the scope of the execution point.

The Watch List shows the process and/or thread ID of the process or thread being viewed. The process ID is only shown if more than one process is loaded in the debugger. The thread ID is only shown if the process whose state you are examining contains more than one thread.

The left side of the Watch List shows the expressions entered as watches. Corresponding data types and values appear on the right. Values of compound data objects (such as arrays and structures) appear between braces { }.

- The Watch List will be blank if you have not added any watches.

If the execution point moves to a location where any of the variables in an expression is undefined (out of scope), the entire watch expression becomes undefined. If the execution point returns to a location where the watch expression can be evaluated (that is, if the execution point re-enters the scope of the expression), the Watch List again displays the current value of the expression.

Note For a watch to work on an element of a variant array, the watch property "Allow Function Calls" must be enabled. For example, say a program has a variant containing an array called A and you want to put a watch on A[0]. If "Allow Function Calls" is not set in the Watch Properties dialog, the value for A[0] is shown as "Inaccessible value."

Watch List commands

Right-click the Watch List to access the following commands that enable you manipulate watch points:

Edit Watch	Opens the Watch Properties dialog box that lets you modify the properties of a watch
Add Watch	Opens the Watch Properties dialog box that lets you create a watch
Enable Watch	Enables a disabled watch expression
Disable Watch	Disables an enabled watch expression
Delete Watch	Removes a watch expression
Enable All Watches	Enables all disabled watch expressions
Disable All Watches	Disables all enabled watch expressions
Delete All Watches	Removes all watch expressions
Stay On Top	Keeps the window visible when out of focus
Inspect	Displays information about the currently selected expression.
Break When Changed	Add a new Data Watch breakpoint
Dockable	Toggles the window for docking

Edit Watch **(Watch List context menu)**

Choose Edit Watch from the Watch List context menu to open the Watch Properties dialog box, where you can create and modify watches. After you create a watch, use the Watch List 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 context menu.
- Right-click an existing watch in the Watch List and choose Edit Watch from the Watch List context menu.

Add Watch **(Watch List context menu)**

Choose Add Watch from the Watch List context menu to open the Watch Properties dialog box, where you can create and modify watches. After you create a watch, use the Watch List 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 context menu.

Enable Watch **(Watch List context menu)**

Choose Enable Watch from the Watch List context menu 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 the IDE 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.

Disable Watch **(Watch List context menu)**

Choose Disable Watch from the Watch List context menu 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 the IDE 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.

Delete Watch **(Watch List context menu)**

Choose Delete Watch from the Watch List context menu 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.

Enable All Watches **(Watch List context menu)**

Choose Enable All Watches from the Watch List context menu 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 the IDE 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.

Disable All Watches **(Watch List context menu)**

Choose Disable All Watches from the Watch List context menu 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 the IDE 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.

Delete All Watches **(Watch List context menu)**

Choose Delete All Watches from the Watch List context menu 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.

Stay On Top

When this option is checked, the Watch List stays visible when not in focus.

Inspect

Select Inspect to display information about the currently selected item from the watch list in the Inspector Window. Inspect is only available at runtime, when the expression has a value.

Break When Changed

Select Break When Changed to add a new Data breakpoint on the selected watch. A data breakpoint is only valid for the current debug session. After the process is terminated, the breakpoint is disabled. At the start of the next debug session you can re-enable the breakpoint by selecting Break When Changed again. You can also use the Enable command in the breakpoint view.

Thread Status box

Choose View|Debug Windows|Threads to view the Thread Status box.

Use this status box to view the status of all processes and threads of execution that are executing in each application being debugged.

Thread status box

Thread ID	Displays the OS assigned thread ID and process name.
State	The thread state is Runnable, Stopped, Blocked, or None; for processes, the state indicates how the process was created: Spawned, Attached, or Cross-process Attach.
Location	Displays the source position. Displays the address if there is no source location available. If the process is remote, the name of the remote machine is shown.

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 exception.
Unknown	The thread is not the current thread so its status is unknown.
Stepped	The last step command was successfully completed.

Threads and multiple process debugging

The threads shown in the thread status box can be running in the same process or in different processes. The first process loaded appears at the top of the list and additional processes get added to the bottom of the list. As a process terminates, it is removed from the list. Each process can have one or more threads which it owns. These threads are shown in the list directly under their owning process. The main thread is directly under its owning process and new threads which the process creates get added to the bottom of that process' thread list. As threads terminate, they are removed from the list

There is the concept of current process and current thread. The current process and current thread become the context for the next user action (run, pause, reset, etc.) Also, most debugger views show information pertinent to the current process and current thread. The current process is denoted using a green arrow glyph. Non-current processes are denoted using a light blue arrow glyph.

You can change the current process by selecting a non-current process or thread and choosing 'Make current' from the popup menu. When invoked on a process, that process and its current thread become current. When invoked on a thread which is not owned by the current process, the thread's owning process become current.

When a debug event occurs (breakpoint, exception, paused), the thread status view indicates the status of each thread as it executes. Using the context menu, you can make a different thread or process current. When a thread is marked as current, the next step or run operation is relative to that thread.

For more information, see Multiple Process Debugging, Setting debugging options for specific processes, and Run Until Return.

Thread Status box commands

Right-click the Thread Status box to access the following commands:

View Source	Displays the Code editor at the corresponding source location of the selected thread ID, but does not make the Code editor the active window.
Go to Source	Displays the Code editor at the corresponding source location of the selected thread ID and makes the Code editor the active window.
Make Current	Makes the selected thread the active thread process if it is not so already. Iif the thread is not already part of the active process, its process also becomes the active process.

Terminate	Terminates the process, if a process is selected, or the process that the thread is part of, if a thread is selected.
Process Properties	Lets you set debugger options temporarily for a particular process during the debugging session.
Dockable	Lets you dock the threads status box to other windows in the IDE.

Modules window

See also

Use the context menus (right-click) in the Modules window to add modules to the Modules window, halt program execution when a module is loaded, or navigate to entry points or display a module's source file in the Code editor.

In the Module pane (upper left)

- Choose **Break On Load** to halt the execution of the application when it loads the selected module into memory.
- If the selected module is already loaded, choose **Add Module** to add a new module to the Modules window. Add Module displays the Add Module dialog box.
- If the selected module is not yet loaded, choose **Edit Module** to replace it with a different module. Edit Module displays the Edit Module dialog box.

In the Source pane (lower left)

- Choose **Edit Source** to display the source code for the module in the code editor. Edit Module transfers focus to the code editor so that you can edit the source.
- Choose **View Source** to display the source code for the module in the code editor without changing focus away from the Modules window.

If a file cannot be found, add the path to the file to the Debugger Source path on the Project|Options| Directories/Conditionals option page.

In the Entry point pane (right)

- Choose **Go to Entry Point** (Enter is the shortcut key) to display the module's entry point and address in the CPU window. The entry point is only shown if the source for it can found.

You can sort the modules listed by entry point name or by address by clicking either column header.

Note: The runtime image base address is the memory offset, in hexadecimal, where the module actually loads, as distinct from the preferred image base address you may have specified in the Project Options window.

Modules and multiple process debugging

The Modules window shows a list of all processes under control of the debugger as well as a list of the modules currently loaded by each process. The first process loaded appears at the top of the list and additional processes get added to the bottom of the list. As a process terminates, it is removed from the list. Each process can have one or more modules which it loads. These modules are shown in the list directly under their owning process. The first module loaded by a process appears directly under its owning process and additional modules loaded by a process get added to the bottom of that process' module list. As modules get unloaded, they are removed from the list.

The current process is indicated by a green arrow glyph in the gutter next to it. Noncurrent processes have no glyphs next to them.

Module load breakpoints are not process specific and will get encountered by any process which loads a module which has a breakpoint set on it.

Add or Edit Module dialog box

See also

Choosing Run|Add Breakpoint|Module Load Breakpoint displays the Add Module dialog box.

Use the Add or Edit Module dialog box to add a module to the Modules window. Modules are automatically added to the Modules window when they are loaded into memory, but if you want to halt execution for debugging when the module first loads into memory, you must add it to the modules window first and then choose BreakOnLoad from its context menu.

Type the module name (usually a .DLL or .BPL) into the edit box, or click the browse button to locate the module with an explorer dialog.

About the toolbars

See also

The toolbars in the IDE provide shortcuts for menu commands. Commands are organized into several toolbars, which can be independently repositioned or pulled into floating tool windows by dragging with the mouse.

You can display or remove toolbars from the display using View|Toolbars or right-clicking on any of the toolbars and checking or unchecking the names of the toolbars.

The toolbars that can appear in the IDE are

- Standard
- View
- Debug
- Custom
- ComponentPalette
- Desktops

The toolbars have Help Hints. To enable Help Hints, select Show Hints from the Options page of the Customize Toolbar dialog. When Help Hints are enabled, you can point to any of the tools on the toolbar and pause to see what the tool is used for.

You can also save and select customized desktop settings from the Desktops toolbar.

Standard toolbar

See also

The Standard toolbar contains the following buttons by default:

Button	Meaning
New	Opens the New Items dialog box.
Open	Displays the File Open dialog box. The Open button has a drop-down button that allows you to select from a list of the most recently opened files.
Save	Lets you store changes to all files included in the open project using the current name for each file. This is the same as selecting File|Save from the menu.
Save All	Lets you save all open files, including the current project and modules. This is the same as selecting File|Save All from the menu.
Open Project	Lets you open an existing project. This is the same as selecting File| OpenProject from the menu.
Add file to project	Opens the Add to Project dialog box.
Remove file from project	Opens the Remove from Project dialog box.

View toolbar

See also

The View toolbar contains the following buttons by default:

Button	Meaning
View Unit	Opens the View Unit dialog box.
View Form	Displays the View Form dialog box.
Toggle Form/Unit	Toggles between a form and its unit window.
New Form	Creates and adds a blank form to the current project. This is the same as choosing File|New Form from the menu.

Custom toolbar

See also

The Custom toolbar contains a single button (help contents) which brings up the contents and index for accessing on-line help.

You can add your own command buttons to this tool bar using the toolbar Customize dialog.

Debug toolbar

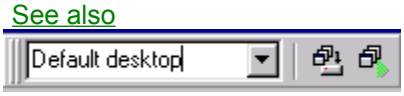
See also

The Debug toolbar contains the following buttons by default:

Button	Meaning
Run	Compiles and executes your application. This is the same as choosing Run|Run from the menu. This button also has a drop down list that lets you change the active project or process. If you are debugging more than one project and you want to switch to a process that is not currently active, click on the little down arrow in the Run Button and select the exe you want to make active. This will not run the exe, just activate it. If the exe is not currently stopped in the debugger, selecting it makes its project the active project in the project manager.
Pause	Temporarily pauses the execution of a running program. This is the same as the Run|Program Pause menu command.
Trace Into	Executes a single program line, tracing into any procedures or functions. This is the same as the Run|Trace Into menu command.
Step Over	Executes a single program line, without tracing into any procedures or functions. This is the same as the Run|Step Over menu command.

Desktops toolbar

See also



The Desktops toolbar contains the following items by default:

Item	Description
Pick list	Lets you switch to the various Desktop layouts you have saved.
Save current desktop	Displays the Save Desktop dialog box where you specify a name under which to save the current desktop settings.
Set debug desktop	Sets the current desktop as the debug desktop, which is automatically displayed during runtime.

You can also use menu items to save the current desktop settings (View|Desktops|Save Desktop), delete desktop layouts you no longer need (View|Desktops|Delete), and specify a particular desktop layout to use during runtime (View|Desktops|Set Debug Desktop).

Toolbar context menu

The toolbar context menu contains commands that enable you to display or hide the toolbars. Check the toolbars you want to display and uncheck the ones you want to hide. It also includes a Customize command that you can use to customize the toolbars. See Configuring the toolbars.

Configuring the toolbars

See also

The toolbars in the IDE are configurable. That is, you can do any of the following:

- Add buttons to a toolbar
- Remove buttons from a toolbar
- Rearrange the buttons on a toolbar

Before you can configure the toolbar, you must display the Customize dialog box by choosing View|Toolbars|Customize or choosing Customize from the toolbar context menu.

To add buttons to the toolbar:

1. Display the Commands page of the Customize dialog box.
2. Enlarge the toolbar area by dragging the grabber on the toolbar to the right.
2. Select a menu from the Categories list box. The commands associated with the selected category 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 any toolbar in an open space.

To remove a button from the toolbar:

Display any page of the Customize dialog box. Drag the button off the toolbar.

To rearrange the buttons on the toolbar:

Display any page of the Customize dialog box. Drag and drop the button to a new position.

Customizable desktop settings

See also

You can customize and save your desktop settings. A Desktops toolbar in the IDE includes a pick list of the available desktop layouts and two icons to easily customize the desktop.

To customize a desktop layout and save all desktop settings:

1. Arrange the desktop as you want it including displaying, sizing, and docking particular windows, and placing them where you want on the display.
2. Click the Save current desktop icon on the Desktops toolbar. (You can also select View|Desktops|Save Desktop.)
3. Type a name for this particular desktop layout and click OK.

You can also type the name of a new or existing desktop layout in the combo box in the Desktops toolbar and press Enter. If you type a new name, the current desktop layout is saved under the new name. If you type an existing desktop, that desktop is then displayed.

Any selected layout will remain in effect for all projects and is used when you next start Delphi.

You can create as many layouts as you like. The names are added to the pick list on the Desktops toolbar. To change desktop layouts, select another choice in the pick list in the Desktops toolbar or select View|Desktops and choose the desktop from the dialog box.

Note: For convenience, you may want to set up a particular layout to use while debugging.

Setting the debug desktop

You can select one of the desktop layouts that you have added as the layout which is enabled every time you run an application. This is useful when debugging an application. You can create a layout including useful debug windows (such as the Watch list and Modules window) positioning them and docking them as you like for the runtime environment.

Note When the debug session ends, the current desktop reverts to the last desktop you were using before the debug session began.

To set the debug desktop:

1. Customize a desktop layout that you want to use for runtime.
2. Click the Set debug desktop icon on the Desktops toolbar. (You can also select View|Desktops|Set Debug Desktop.)
3. Select the desktop layout that you want to be used when you run applications.

The debug desktop is enabled when you run an application. The desktop layout selected in the Desktops toolbar pick list is used at all other times.

To delete a desktop layout:

1. Choose View|Desktops|Delete.
2. Select the desktop layout that you no longer need and click Delete.

Delphi Productivity Tools

Delphi includes a number of specialty applications designed to help you work more efficiently. The following links provide easy access to the Help systems for these tools.

Note: The Delphi 5 Enterprise edition includes all of the tools described below. Delphi 5 Professional does *not* offer SQL Builder, SQL Monitor, and TeamSource. Delphi 5 Standard includes *only* the Image Editor.

- The **Image Editor** lets you create, open, and save icons, cursors, and bitmaps for use in your applications.
- **WinSight** provides debugging information about window classes, windows, and messages. You can use this tool to examine how any application creates classes and windows, and monitor how windows send and receive messages. *(Available in Delphi 5 Enterprise and Professional editions only.)*
- **Borland Database Engine (BDE)** is the 32-bit Windows-based core database engine and connectivity software behind Borland products, as well as Paradox® for Windows and Visual dBASE® for Windows. This Help file offers a reference to the BDE's features and language elements. *(Available in Delphi 5 Enterprise and Professional editions only.)*
- The **BDE Administrator** lets you configure the Borland Database Engine (BDE), configure numerous database drivers, create and delete OBDC drivers, and create and maintain database aliases. *(Available in Delphi 5 Enterprise and Professional editions only.)*
- **Borland SQL Links for Windows (32-bit version)** is a set of BDE-hosted driver connections to database servers. By creating queries, SQL Links emulates full navigation capabilities, enabling users to access and manipulate data in SQL databases by using convenient features in Borland applications. *(Available in Delphi 5 Enterprise and Professional editions only.)*
- **Local SQL (online reference)**. Local SQL is the subset of the SQL-92 specification used to access dBASE, Paradox, and FoxPro tables. On receving local SQL statements from front-end applications, the Borland Database Engine (BDE) translates the statements into BDE API functions. *(Available in Delphi 5 Enterprise and Professional editions only.)*
- **Data Pump** lets you move data (both database schema and content) between databases. *(Available in Delphi 5 Enterprise and Professional editions only.)*
- **Database Explorer** is a hierarchical database browser with editing capabilities, letting you browse and edit database server-specific schema objects, including tables, fields, stored procedure definitions, triggers, and indexes. *(Available in Delphi 5 Enterprise and Professional editions only.)*
- **SQL Builder** lets you visually and interactively create and execute SQL queries, and serves as a tool for SQL. *(Available in Delphi 5 Enterprise edition only.)*
- **SQL Monitor** lets you view statement calls made through SQL Links to a remote server or through the ODBC socket to an ODBC data source. *(Available in Delphi 5 Enterprise edition only.)*
- The **TeamSource** workflow management tool uses a parallel model of source control to help with the management and coordination of work in a shared development environment. **Note:** *The TeamSource tool, available only in the Delphi 5 Enterprise edition, is a separate product and requires a separate installation.*

About the integrated debugger

See Also

No matter how careful you are when writing 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. The IDE 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

Types of errors

There are three basic types of program errors:

- Compile-time
- Logical errors
- Runtime errors

The integrated debugger can help you track down both runtime errors and logic errors. By running to specific program locations and viewing the state of your program at those places, you can monitor how your program behaves and find the areas where it is not behaving as you intended.

Compile-time errors

Errors that violate a rule of language syntax. You cannot compile your program unless it contains valid statements.

The most common causes of compile-time (syntax) errors are

- Typographical mistakes
- Missing semicolons
- References to undeclared variables
- Wrong number or type of arguments passed to a function
- Wrong type of values assigned to a variable

Runtime errors

Runtime 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 runtime 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.

Logic errors

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.

Fixing syntax errors

If your code has compile-time (syntax) errors and you try to compile it, the Message View of the Code editor opens and displays the errors and warnings generated.

To correct syntax errors,

1. In the Message View, double-click the error or warning that you want to fix. (If the Message View is not open, right-click the Code editor and choose Message View.)

   The IDE positions your cursor on the line in your source code that caused the problem.
2. Make your correction.
3. If your code has more than one problem, double-click another error or warning in the Message window.
4. Choose Project|Build All or Project|Make to recompile your program.
5. Choose Run|Run to verify that your program is operating correctly.

Planning a debugging strategy

After program design, program development consists of a continuous cycle of coding and debugging. Only after you thoroughly test your program should you distribute it to your end users. To ensure that you test all aspects of your program, it is best to have a thorough plan for your debugging cycles.

One good debugging method involves dividing your program into different sections that you can debug systematically. By closely monitoring the statements in each section, you can verify that each area is performing as designed. If you do find a programming error, you can correct the problem in your source code, recompile the program, and resume testing.

Using the integrated debugger

Although there are many ways to debug code, you will typically use one or more of the following steps:

1. Preparing your project for debugging by compiling and linking your program with debug information.
2. Control Program Execution by running to a program location you would like to examine.
3. Examine the state of the program data values and view the program output.
4. Modify program data values to test bug fixes.
5. Reset or pause the debugging session.
6. Fix the error.

Preparing your project for debugging

If you find a runtime or logic error in your program, you can begin a debugging session by running your program under the control of the debugger:

1. Compile and link your program with debug information.
2. Run your program from the IDE.

Generating debug information for your project

The IDE automatically generates debug information. To manually choose to turn on debug information for your project,

1. Choose Project|Options.
2. Click the Compiler tab.
3. From the Debugging pane click Debug information to include symbolic debug information. To view variables local to procedures and functions click Local Symbols.
4. If you are using remote debugging, check “Include remote debug symbols” on the EXE and DLL options pane of the Linker tab. The “Debug project on remote machine” checkbox on the Run| Parameters Remote tab is automatically linked to the “Include remote debug symbols” checkbox. These options are only linked in one direction, meaning that changing the “Debug project on remote machine” checkbox has no effect on the “Include remote debug symbols” checkbox.

Note: When you check “Include remote debug symbols”, Delphi generates a .RSM file containing the remote symbols. This file should stay with the .EXE on the remote machine.

Enabling the debugger

The debugger is enabled automatically. To manually choose to enable the debugger,

1. Choose Tools|Debugger Options.
2. Check Integrated Debugging. This option is on by default.
3. Choose Tools|Environment Options and check Minimize On Run from the Preferences tab if you want to minimize the IDE when you run your program.
4. From the Preferences tab click Hide Designers On Run to close the Object Inspector and Form Designer when you run your program.

Note: If you are using TD32 or have a console mode application, you should choose the appropriate checkboxes on the EXE and DLL options pane of the Linker tab.

Turning debugging information off

Adding debug information increases the file size of your program. When you have fully debugged your program, be sure to build the final executable files with debugging information turned off to reduce the final size of your program files.

To turn off debugging information

1. Choose Project|Options
2. Click the Compiler tab and from the Debugging pane uncheck Debug information and Local symbols.

Running your program in the IDE

After you compile your program with debug information, you can begin a debugging session by running your program from the IDE. Doing so lets you control when your program runs and when it pauses. Whenever your program is paused in the IDE, the debugger takes control.

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, arrange your screen so you can see both the Code editor and your application window as you debug.

Debugging with program arguments

To pass runtime arguments to the program you want to debug,

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 and click OK. If you are using remote debugging, add the argument to the control on the remote tab.

Controlling program execution

The most important aspect of a debugger is that it lets you control the execution of your program. You can 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 treats 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 treats a single statement that spans several lines of text as a single line of code.

The debugger lets you control program execution in the following ways:

- Running to the cursor
- Stepping through code
- Running to a breakpoint location
- Pausing your program

Execution point

The execution point indicates the next line of source code or machine instruction in your program that will be executed when you run your program through the integrated debugger. Whenever you pause program execution, the debugger highlights a line of source code or machine instruction, marking the location of the execution point.

Running to the cursor

See also

When beginning a debugging session, you often run your program to a spot just before the suspected location of the problem. At that point, use the debugger to ensure that all data values are as they should be. If everything appears to be correct, you can run your program to another location, and again check to ensure things are functioning correctly.

You can tell the debugger you want to execute your program normally (not step-by-step) until a certain spot in your code is reached. In the Code editor or CPU window, position the cursor on the line where you want to begin (or resume) debugging. Then either:

1. Right-click the Code editor and choose Debug|Run to current.
2. Right-click in the Disassembly pane of the CPU window and choose Run To Current.
3. Choose Run|Run to Current from the main menu.
4. Use F4, under the default keymapping.

Stepping overview

Stepping is the simplest way to move through your code one statement or machine instruction at a time. Stepping lets you run your program one line (or instruction) at a time – the next line of code (or instruction) will not execute until you tell the debugger to continue. After each step, you can examine the state of the program, view the program output, and modify program data values. Then, when you are ready, you can continue executing the next program statement.

You can step through code in two basic ways:

Trace Into — The Trace Into command causes the debugger to walk through your code one statement or instruction at a time. If the execution point is located on a function call, the debugger moves to the first line of code or instruction that defines that function. From here, you can execute that function, one statement or instruction at a time. When you step past the return of the function, the debugger resumes stepping from the point where the function was called. (Stepping through your program one statement at a time is known as single stepping.)

Step Over — The Step Over command is the same as Trace Into, except that when the execution point is on a function call, the debugger executes the function at full speed and then pauses on the line of code or instruction following the function call.

You can also use Run|Run Until Return to run the loaded program until execution returns from the current function. The process stops on the instruction immediately following the instruction that called the current function.

Statement stepping and instruction stepping

The debugger lets you step through either

- statements in your source code viewed in the Code editor.
- machine instructions viewed in the CPU window.

The debugger automatically steps through your code at the instruction level and displays the CPU window in the following situations:

- If the CPU window has focus when you choose the Trace Into or Step Over command.
- If you pause the program in a spot where there is no debug information available.
- When an exception is raised at a point where there is no debug information, and the user checks the view CPU checkbox from the exception dialog box that appears.
- If the debugger stops at an address or data breakpoint.

Statement Stepping granularity

The debugger steps over single lines of lines of code based on the following rules:

- If you string several statements together on one line, you cannot debug those statements individually; the debugger treats all statements as a single line of code.
- If you spread a single statement over multiple lines in your source file, the debugger executes all the lines as a single statement.

Stepping over code

See also Overview of Stepping

To Step Over, choose the Run|Step Over or press *F8* (default key mapping).

When you choose the Step Over command, the debugger executes the code highlighted by the execution point. If the execution point is highlighting a function call, the debugger executes that function at full speed, including any function calls within the function highlighted by the execution point. The execution point then moves to the next complete line of code or executable instruction.

As you debug, you can choose to Trace Into some functions and Step Over others. Step Over is good to use when you have fully tested a function, and you do not need to single step through its code.

Tracing into code

See also Overview of stepping

To Trace Into code, choose either of the following commands:

- Run|Trace Into or press *F7* (default key mapping)
- Run|Trace To Next Source Line or press *Shift+F7* (default key mapping).

When you choose Run|Trace Into, the debugger executes the code highlighted by the execution point. If the execution point is highlighting a function call, the debugger moves the execution point to the first line of code or instruction that defines the function being called. If the executing statement calls a function that does not contain debug information, the debugger runs the function at full speed (as if you had chosen the Step over command).

When you choose Run|Trace To Next Source Line, the debugger moves to 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.

When you step past a function return statement (in this case, the end statement), the debugger positions the execution point on the line following the original function call.

As you debug, you can choose to Trace Into some functions and Step Over others. Use Trace Into when you need to fully test the function highlighted by the execution point.

You can also use Run|Run Until Return to run the loaded program until execution returns from the current function. The process stops on the instruction immediately following the instruction that called the current function.

Running to a breakpoint

See also

You set breakpoints on lines of source code or address locations (machine instructions) where you want program execution to pause during a run. Using a breakpoint is similar to using the Run to Cursor command in that the program runs at full speed until it reaches a certain point. Unlike Run to Cursor, however, you can have multiple breakpoints and you can choose to stop at a breakpoint only under certain conditions. Once your program's execution is paused, you can use the debugger to examine the state of your program.

Interrupting program execution

See also

Sometimes while debugging, you will find it best to stop program execution or to start the debugging session from the beginning of the program.

Choose…	To…
Run|Program Pause	temporarily pause the execution of a running program.
Run|Program Reset	terminate the current debugging session, and start with a fresh slate.

Pausing your program

See also

Instead of stepping through code, you can use a simpler technique to pause your program:

Choose Run|Program Pause and your program will stop executing.

You can then examine the value of variables and inspect data at this state of the program. When you are done, choose Run|Run to continue the execution of your program.

- In most cases, the CPU window will display when you pause your program, such as when the current instruction does not have corresponding source code.

Restarting a program

See also

Sometimes while debugging, you might need to start over from the beginning of your program. For example, it might be best to restart the debugging session if you have executed past the point where you believe there is a bug, or if variables or data structures become corrupted with unwanted values.

To restart your program, choose Run|Program Reset.

When you terminate the process, the IDE

- resets the integrated debugger so that running or stepping, begins at the start of the program.
- does not change the location of the source code displayed in the Code editor so that you can easily position the cursor to run your program to the line you were on when you reset it.
- disables any Data Breakpoints that are set. You must re-enable them when you start your next debug session.

Fixing program errors

See also

Once you have found the location of the error in your program, you can type the correction directly into the Code editor and the change takes effect immediately. Once you change a line of code in the Code editor, however, the IDE prompts you to rebuild your program before you resume program execution and continue debugging.

Instead of fixing an error while debugging, you might want to test your fix by modifying data values using the debugger. This way, you do not have to recompile your program to see if your fix works.

Using breakpoints

See also

Breakpoints pause program execution during a debugging session at source code or address locations that you specify. You can set breakpoints before potential problem areas, then run your program at full speed. Your program pauses when it encounters a breakpoint, and the Code editor or CPU view Disassembly pane displays the line or address location 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 or machine instruction at a time.

The IDE keeps track of all your breakpoints during a debugging session and associates them with your current project. You can maintain all your breakpoints from a single Breakpoints List window and not have to search through your source code files to look for them.

Debugging with breakpoints

When you run your program from the IDE, it will stop whenever the debugger reaches the location in your program where the breakpoint is set, but before it executes the line or machine instruction.

- If you set a breakpoint on a line in your source code, the line that contains the breakpoint appears in the Code editor highlighted by the execution point.
- If you set a breakpoint on an address location, the instruction that contains the breakpoint appears in the CPU window Disassembly pane (or in the Code editor on the line that most closely corresponds to the address location) highlighted by the execution point.

At this point, you can perform any other debugging actions.

Setting breakpoints after program execution begins

While your program is running, you can switch to the debugger (the IDE), just as you would switch to any Windows application, and set a breakpoint. When you return to your application, the new breakpoint is set, and your application will pause or perform a specified action when it reaches the breakpoint.

- You must set a breakpoint on an executable line of code or machine instruction. For example, 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 are disabled when you run your program.

Setting breakpoints

See also

You can set breakpoints before you begin debugging or while your program is running using the Code editor or the CPU window Disassembly pane. Your application halts when it reaches a breakpoint.

- For a breakpoint to be valid, it must be set on an executable line of code. Breakpoints set on comment lines, blank lines, declarations, or other non-executable lines of code are invalid and become disabled when you run your program.

Source breakpoints

To set a breakpoint on a line of source code, select the line in the Code editor where you want to set the breakpoint, then use **one** of the following methods:

- Click the left margin of the line.
- Right-click anywhere on the line and choose Debug|Toggle Breakpoint.
- Place the insertion point anywhere in the line and press *F5* (default key mapping).
- Right-click the Breakpoint List window and choose Add|Source Breakpoint.

Breakpoints are shown in color with a filled circle in the left gutter of the Code editor (red by default). When you point to the circle in the gutter, a tooltip displays showing the breakpoint's pass count and condition.

If you know the line of code where you want to set a breakpoint,

1. Choose Run|Add Breakpoint|Source Breakpoint and type the source-code line number in the Line Number box.
2. Complete the settings in the Add Source Breakpoint dialog box to create the breakpoint.

When you set a breakpoint, the line on which the breakpoint is set becomes highlighted, and a stop-sign appears in the left margin of the breakpoint line.

Invalid breakpoints

If a breakpoint is not placed on an executable line of code, the debugger considers it invalid. For example, a breakpoint set on a comment, a blank line, or declaration is invalid. If you set an invalid breakpoint, the debugger marks the breakpoint invalid and runs. To correct this situation, delete the invalid breakpoint from the Breakpoint List window. You can then set the breakpoint in the intended location. You can, however, also ignore invalid breakpoints; the IDE disables any invalid breakpoints when you run your program.

- During the linking phase of compilation, lines of code that do not get called in your program are marked as dead code by the linker. In turn, the integrated debugger marks any breakpoints set on dead code as invalid.

Address breakpoints

The debugger supports address breakpoints. When set, the debugger breaks if the instruction at the specified address gets executed.

You can set an address breakpoint in the following ways:

- When in the Breakpoint List window, choose Add|Address breakpoint. On the dialog, enter an address.
- From the Run menu choose Add Breakpoint|Address Breakpoint. On the dialog, enter an address.
- From the gutter of the CPU window click the mouse.
- From the Disassembly pane of the CPU window right-click and choose Toggle breakpoint.
- Press *F5*.

Address breakpoints are only available when the process is paused in the debugger.

Data breakpoints

The debugger supports data breakpoints. When set, the debugger breaks if the memory at the specified address is written to. You can set a data watch breakpoint three ways:

- When in the breakpoint view, choose Add|Data breakpoint. On the dialog, enter an address and specify a length. You can also enter symbol names such as variable names.
- When in the watch view, select an item, right-click, and choose Break When Changed. Selecting this menu item sets a Data Watch breakpoint.
- From the Run menu choose Add Breakpoint|Data Breakpoint. On the dialog, enter an address and specify a length.

When your current debug session ends, Data Breakpoints are marked disabled. On the start of your next debug session, you need to re-enabled them from either the Breakpoint view (Breakpoint list window) or the Watch view (Watch List). Data breakpoints are only available when the process is paused in the debugger.

Modifying breakpoint properties

See also

You can specify breakpoint properties when you create a breakpoint, or you can edit the properties after creation. Use the Breakpoint Properties dialog boxes to modify breakpoint properties.

Adding breakpoints

Use the Add Breakpoint dialog boxes to add a breakpoint. You can open these Breakpoint dialog boxes in the following ways:

- Choose Run|Add Breakpoint and select Source Breakpoint, Address Breakpoint, Data Breakpoint or Module Load Breakpoint.
- Choose View|Debug Windows|Breakpoints, then right-click the Breakpoint List window. Choose Add, and then choose Source Breakpoint, Address Breakpoint, or Data Breakpoint.

Editing breakpoints

Use the Breakpoint Properties dialog boxes to modify an existing breakpoint. You can open these Breakpoint Properties dialog boxes in the following ways:

- Right-click an existing source, address, or data breakpoint in the Breakpoint List window and choose Properties. Do **not** check Keep existing breakpoint if you want to modify a breakpoint.
- Right-click in the gutter on an existing source breakpoint in the Code editor and choose Properties.
- Right-click in the gutter on an existing address or source breakpoint in the CPU window and choose Properties.

Use the following options to specify where and when you want a breakpoint to pause your program. These options are available depending upon the type of breakpoint set and the point at which you decide to modify it:

Filename

Sets or changes the program file for the breakpoint. Enter the name of the program file for the breakpoint. (This option appears only for a breakpoint set on a line of source code in the Code editor.)

Line Number

Sets or changes the line number for the breakpoint. Enter or change the line number for the breakpoint. (This option appears only for a breakpoint set in the Code editor on a line of source code.)

Address

Sets a breakpoint on a machine instruction. Enter a specific starting address or any symbol, such as a variable or a class data member or method, that evaluates to an address. (This setting appears only for a breakpoint set on a machine instruction in the Disassembly pane in the CPU window.)

Condition

Specifies a conditional expression that is evaluated each time the breakpoint is encountered. Program execution stops when the expression evaluates to true. You can enter any valid language expression. All symbols in the expression, however, must be accessible (within scope) from the breakpoint's location.

- For more information, see Creating Boolean expressions.

Pass Count

Stops program execution at a certain line number or machine instruction after a specified number of passes. The integrated debugger decrements the pass count number each time the line containing the breakpoint is encountered. When the pass count equals 1, program execution pauses.

When you use pass counts with conditions, program execution pauses the *n*th time that the conditional expression is true. The debugger decrements the pass count only when the conditional expression is true.

- For more information, see Using Pass Counts.

Creating conditional breakpoints

See also

When a breakpoint is first set, by default, program execution pauses each time the breakpoint is encountered. The Add Breakpoint dialog boxes lets you customize your breakpoints so that your program pauses only when a specified set of conditions is met.

To create a conditional breakpoint:

1. Choose Run|Add Breakpoint and select Source Breakpoint, Address Breakpoint, or Data Breakpoint OR right-click the Breakpoint List window, choose Add, and then choose Source Breakpoint, Address Breakpoint, or Data Breakpoint.
2. Enter the required information on the Condition line of the dialog box.

The integrated debugger provides two types of breakpoint conditions:

- Boolean expressions
- Pass counts

To modify conditions, use the Breakpoint Properties dialog boxes. For details see Modifying Breakpoint Properties.

Creating Boolean expressions

See also

The Condition edit box in the Breakpoint dialog box lets you enter an expression that is evaluated each time the breakpoint is encountered during the program execution. If the expression evaluates to true (or not zero), the breakpoint pauses the program run. If the condition evaluates to false (or zero), the debugger does not stop at the breakpoint location.

Conditional breakpoints are useful when you want to see how your program behaves when a variable falls into a certain range or what happens when a particular flag is set.

For example, suppose you want a breakpoint to pause on a line of code only when the variable *mediumCount* is greater than 10. To do so,

1. Place the insertion point on the line of code you want in the Code editor and press *F5* to set the breakpoint.
2. Choose View|Debug Windows|Breakpoints to open the Breakpoint List window.
3. In the Breakpoint List window, highlight the breakpoint you just created, then right-click and choose Properties.
4. On the Breakpoint Properties dialog box, enter the following expression into the Condition edit box:

```
mediumCount > 10
```

5. To modify a breakpoint, do NOT check Keep existing breakpoint.

- You can input any valid language expression into the Condition edit box, but all symbols in the expression must be accessible (within scope) from the breakpoint's location.

Using pass counts

See also

The Pass Count edit box enables you to specify a particular number of times that a breakpoint must be passed for the breakpoint to be activated. A pass count tells the debugger to pause program execution the *n*th time that the breakpoint is encountered during the program run (you supply the number *n* which is set to 1 by default).

The current pass count number decrements each time the line containing the breakpoint is encountered during the program execution. If the current pass count equals the specified pass count number when the breakpoint line is encountered, program execution pauses on that line of code. For example, if you enter a pass count of 2, your program stops the second time the debugger reaches the line where the breakpoint is set.

When you use a pass count in conjunction with a Boolean condition, the breakpoint pauses program execution the *n*th time that the condition is true; the condition must be true for the pass count to decrement. For example, if you enter the expression x>3 in Conditions and the number 2 in Pass Count, your program stops the second time the debugger reaches the breakpoint when the value of *x* is greater than 3.

Locating breakpoints

See also

If a breakpoint is not visible in the Code editor or in the CPU view, you can use the Breakpoint List window to quickly locate the breakpoint.

To scroll the Code editor to the location of a breakpoint in your source code,

- Right-click on a source breakpoint in the Breakpoint List window and choose View Source.

To scroll the Code editor to the location of a breakpoint in your source code and make the Code editor active,

- Right-click on a source breakpoint in the Breakpoint List window and choose Edit Source.

To scroll the CPU window to the location of an address breakpoint and make the CPU window active,

- Right-click on an address breakpoint in the Breakpoint List window and choose either Edit Source or View 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 so you can modify the source code at that location.

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 Enable and toggle it OFF (so that it no longer has a check mark next to it).

To disable all breakpoints,

- Right-click the Breakpoint List window, but not on a breakpoint, and choose Disable All.

To enable a single breakpoint,

- Right-click the breakpoint in the Breakpoint List window and choose Enabled and toggle it OFF (so that it no longer has a check mark next to it).

To enable all breakpoints,

- Right-click the Breakpoint List window, but not on a breakpoint, and choose Enable All.

See Breakpoint List context menu for a complete list of the context menus available from the Breakpoint List window.

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, the CPU window, or the Breakpoints window:

To delete a single breakpoint,

- Right-click the breakpoint in the Breakpoint List window and choose Delete.
- Right-click the breakpoint in the Code editor and choose Debug|Toggle breakpoint.
- Right-click the breakpoint in the CPU window and choose Toggle breakpoint.
- Place the insertion point anywhere in the line in the Code editor containing the breakpoint or highlight the breakpoint in the CPU window and press *F5*. (using the default keymapping).
- Click the stop-sign glyph in the left gutter of the line containing the breakpoint in the Code editor or CPU window.
- Use the Delete key or Ctrl+D in the Breakpoint list window to delete the selected breakpoint.

To delete all breakpoints,

- Right-click the Breakpoint List window and choose Delete all.

Examining program data values

See also

After you have paused your application using the integrated debugger, you can examine the different symbols and data structures with regard to the location of the current execution point. 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.

Data evaluation operates at the level of expressions. An expression consists of constants, variables, and values contained in data structures, combined with language operators.

- Almost anything you can use as the right side of an assignment operator can be used as a debugging expression, except for variables not accessible from the current execution point.

You can view the state of your program by

- Watching program values
- Evaluating and modifying expressions
- Inspecting data elements
- Viewing the low-level state of your program
- Viewing functions in the Call Stack window
- Viewing Local Variables from the View|Debug Windows menu.

Modifying program data values

See also

Sometimes you will find that a programming error is caused by an incorrect data value. Using the integrated debugger, you can test a "fix" by modifying the data value while your program is running. You can modify program data in the following ways:

- Modifying variables
- Changing the value of inspector items
- Using the CPU window's Memory Dump pane

Watch expressions

See also

If you want to monitor the value of a variable or expression while you debug your code, add a watch to the Watch List. The Watch List window displays the current value of the watch expression based on the scope of the execution point.

Each time your program's execution pauses, the debugger evaluates all the items listed in the Watch List and updates their displayed values.

You can set a watch expression in the following ways:

- The easiest way to set a watch is to place the insertion point on a term in the Code editor, then right-click and choose Debug|Add Watch at Cursor.
- You can also set a watch and specify its properties on the Watch Properties dialog box (from the Run|Add Watch menu). For more information, see Setting watch properties.

Setting watch properties

See also

Use the Watch properties dialog box to set the properties of a new watch expression or to change the properties of an existing one.

You can open the Watch Properties dialog box in the following ways:

- Choose Run|Add Watch from the main menu.
- Right-click the Watch List and choose Add Watch.
- Select a watch in the Watch List, then right-click and choose Edit Watch.

Formatting watch expressions

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, by default, integer values are displayed in decimal form. If you select Hexadecimal in the Watch Properties dialog box for an integer type expression, the debugger changes the display format from decimal to hexadecimal.

If you are setting up a watch on an element in a data structure (such as an array), you can display the values of consecutive data elements. For example, suppose you have an array of five integers named *xarray*. Type the number `5` in Repeat Count on the Watch Properties dialog box to see all five values of the array. To use a repeat count, however, the watch expression must represent a single data element.

To format a floating-point expression, select Decimal at Display format and enter a number for Digits on the Watch Properties dialog to indicate the number of significant digits you want displayed in the Watch List.

The following table describes the watch expression format options and their effects.

Option	Types affected	Description
Hexadecimal	integers/characters	Shows integer values in hexadecimal with the 0x prefix, including those in data structures.
Character	characters/strings	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 forth).
Decimal	integers	Shows integer values in decimal form, including those in data structures.
Floating point	floating point	Shows the significant digits specified; from 2-18. The default is 7.
Memory dump	all	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. Use the Repeat Count setting to specify the number of bytes you want to display.
Pointer	pointers	Shows the address of the pointer.
Structure/Union	structures /unions	Shows field names and unions as well as values such as X:1;Y:10;Z:5.
String	char, strings	Shows ASCII 0 to 31 as C escape sequences. Use this option only to modify memory dumps.
Default	all	Shows the result in the display format that matches the data type of the expression.

Enabling and disabling watches

See also

Evaluating many watch expressions can slow down the process of debugging. Disable a watch expression when you prefer not to view it in the Watch List window, but want to save it for later use.

When you set a watch, it is enabled by default. Disabling a watch hides the watch from the current program run. When you disable a watch, its settings remain defined, but the debugger does not evaluate it.

To enable or disable a watch,

1. Choose View|Debug Windows|Watches to open the Watch List.
2. Select a watch, then right-click and choose Enable or Disable watch.

The flag <disabled> appears next to a watch that is disabled.

To disable or enable all watches,

Right-click the Watch List and choose Enable All Watches or Disable All Watches.

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,

1. Choose View|Debug Windows|Watches to open the Watch List.

1. Select a watch, then right-click and choose Delete Watch.

OR

Use the del key or the Ctrl+D key combination to delete the selected Watch.

To delete all watches in a source code file,

- Right-click the Watch List and choose Delete All Watches.

Evaluating and modifying expressions

See also

Use the Evaluate/Modify dialog box to evaluate or change the value of an existing expression or property. The Evaluate/Modify dialog box has the advantage over watches in that it enables you to change the values of variables and items in data structures during the course of your 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. This technique can be useful if you think you have found the solution to a bug, and you want to try the correction without having to exit the debugger, changing the source code, and recompiling the program.

To evaluate an expression or property

1. Open the Evaluate/Modify dialog box one of the following ways.
   - Choose Run|Evaluate/Modify.
   - Right-click the Code editor and choose Debug|Evaluate/Modify.
2. Type an expression in the Expression box.

   By default, the word at the cursor position in the current Code editor is placed in the Expression input box. You can accept this expression, enter another one, or choose an expression from the history list of expressions you have previously evaluated. If you want to evaluate a function call, enter the function name, parentheses, and arguments just as you would type it into your program, but leave out the statement-ending semicolon (;).
3. Choose Evaluate. The value of the item appears in the Result edit box.

Evaluating expressions

See also

You can evaluate any valid language expression, except those that contain variables that are not accessible from the current execution point.

Formatting values

To format the result that displays, add a comma and one or more format specifiers to the end of the expression entered in the Expression box. For example:

- To display a result in hexadecimal, type `,H` after the expression.
- To see a floating point number to 3 decimal places, type `,F3` after the expression.

For a complete list of format options, see Evaluate/modify format specifiers.

Evaluate/Modify dialog box

The Evaluate/Modify dialog box provides the following options:

Expression

Lets you specify the variable, array, or object to evaluate or modify.

Result

Displays the value of the item specified in the Expression text box after you choose Evaluate or Modify.

New value

Lets you assign a new value to the item specified in the Expression edit box.

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.

Modifying variables

After you have evaluated a variable or data structure item, you can modify its value. When you modify a value 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.

To change the value of an expression

1. Open the Evaluate/Modify dialog box one of the following ways.
   - Choose Run|Evaluate/Modify
   - Right-click the Code editor and choose Debug|Evaluate/Modify.
2. Specify the expression in the Expression edit box. To modify a component property, explicitly specify the property name. For example, enter: `Form1.Button1.Height`
3. Enter a value in the New Value edit box.
4. Choose Modify. The new value is displayed in the Result box.

- You cannot undo a change to a variable after you choose Modify. To restore a value, however, you can enter the previous value in the Expression box and modify the expression again.

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 with a single expression.
- 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 runtime error, it is not a legal modification value.
- Use caution when you modify variables or when evaluating functions while debugging an application – any side effects that occur will modify the data values of the program you are debugging. For example, if you evaluate a function that increments a variable, the new value of that variable will be reflected when you continue to step through your application. Modifying values (especially pointer values and array indexes) can have undesirable effects because you might overwrite other variables and data structures. Because these errors might not be immediately apparent, use caution whenever you modify program values from the debugger.

Inspecting data elements

See also

Inspector windows are only available when the process is stopped in the debugger.

Inspector windows are the best way to view data items because the debugger automatically formats Inspector windows according to the type of data it is displaying. Inspector windows are especially useful when you want to examine compound data objects, such as arrays and linked lists. Because you can inspect individual items displayed in an Inspector window, you can "walk" through compound data objects by opening an Inspector window on a component of the compound object.

To display an Inspector window directly from the Code editor,

1. Place the insertion point in the Code editor on the data element you want to inspect.
2. Right-click and choose Debug|Inspect.

To inspect a data element from the menu bar,

1. Choose Run|Inspect from the menu bar to display the Inspect dialog box.
2. Type the expression you want to inspect, then choose OK.

Scope

Unlike watch expressions, the scope of a data element in an Inspector window is fixed at the time you evaluate it:

- If you use the Inspect command from the Code editor, the debugger uses the location of the insertion point to determine the scope of the expression you are inspecting. This makes it possible to inspect data elements that are not within the current scope of the execution point..
- If you use the Run|Inspect command from the menu bar, the data element is evaluated within the scope of the execution point.

  If the execution point is in the scope of the expression you are inspecting, the value appears in the Inspector window. If the execution point is outside the scope of the expression, the value is undefined and the Inspector window becomes blank.

Inspecting local variables

While in debug mode, you can show the current function's local variables. To do so, choose View|Debug Windows|Local variables.

Data types

The number of panes and the appearance of the data in the Inspector window depends on which of the following types of data you inspect:

- scalar variables
- functions
- constants
- arrays
- pointers
- classes
- objects
- records and interfaces

For example, if you inspect an array, you will see a line for each member of the array with the array index of the member. The value of the member follows in its display format, followed by the value in hexadecimal.

Inspecting scalar variables

See also

When you inspect a scalar variable, such as simple data items including **Integer**, **Real**, and so on, the top of the Inspector window shows the name, type, and address of the variable. The middle pane shows the name of the scalar on the left and its current value on the right. Integer values are displayed first in decimal, followed by the hexadecimal value enclosed in parentheses.

If the variable inspected is of type **Char**, the equivalent character appears to the left of the numeric values. If the present value does not have a printable character equivalent, the debugger displays a backslash (\) followed by the Object Pascal hexadecimal value that represents the character value.

Inspecting pointers and arrays

See also

When you inspect a pointer or an array, Inspector windows show the values of variables that point to other data items. The top of the Inspector window shows the name, type, count, address (or register if applicable), and pointer location of the variable. The middle pane shows the current values of the data pointed to. The bottom of the Inspector window shows the data type to which the pointer points.

If the value pointed to is a compound data object (such as a structure or record, or an array), the values are enclosed in braces ({}) and the Inspector window displays as much of the data as possible.

If the pointer appears to be pointing to a null-terminated character string, the debugger displays the value of each item in the character array. The left of each line displays the array index ([0], [1], [2], and so on), and the values appear on the right. When you inspect character strings, the entire string appears at the top of Inspector window, along with the address of the pointer variable and the address of the string that it points to.

Inspecting records and interfaces

See also

When you inspect a structure or record, the Inspector window shows the values of members contained in compound data objects.

The top of the window shows the name of the object. The middle pane lists the names and values of the data members of contained in the object, and contains as many lines as needed to show the entire data object.

The bottom of the window shows the data type of the member currently selected.

Inspecting functions

See also

When you inspect a function, the top of the Inspector window shows the function or procedure name, prototype, and its address in memory. The middle pane shows the function's arguments. To inspect a function, enter the function’s name without parentheses or arguments.

If the function is currently on the call stack, its parameters appear at the bottom of the inspector window.

Isolating the view in an Inspector window

See also

You can more closely inspect certain elements (such as classes, records, and arrays) in the Inspector window to isolate the view to the member level:

1. Select an item in the Inspector window.
2. Right-click and choose Inspect to open a new Inspector window, or choose Descend to update the display of the current Inspector window.

The scope of the data element remains the same as it was when you opened it on the Inspector window. If you select a data member that is a pointer to a class, the Inspector window displays the class pointed to.

Changing the value of Inspector items

See also

An ellipsis (…) appears next to a data element that can modified.

To change the value of a inspected element,

1. Select an item in the Inspector window.
2. Click the ellipsis (…), or right-click the element and choose Change.
3. Type a new value, then choose OK.

Using Module view in a debugging session

See also

Use the module view to see different modules, such as .EXEs and DLLs, within a single debug session. Module view is a three-paned view that shows information about the different modules loaded by the process you are debugging.

To display module view,

1. Start a debug session.
2. Choose View|Debug Windows|Modules to display the Module View.
3. Select a module from the list in the upper-left pane.

   In the lower-left pane, a tree-view list of source files appears.
4. Expand source files to show the files included in the source file.
5. Right-click on any source file or entry point to go to the code editor. For more information on the context menu, see Module Window context menu.

If a file cannot be found, add the path to the file to the Debugger Source path on the Project|Options| Directories/Conditionals option page.

The upper-left pane shows each module name and the address at which it is loaded. The lower left pane shows a tree view display of source files used to build the module. The right pane shows a list of entry points into the module.

Locating function calls

See also

While debugging, it is useful to know the order of function calls that brought you to your current location. The Call Stack window lets you view the current sequence of function calls. It also shows the values of the arguments passed to each function call (the arguments with which the call was made).

To open the Call Stack window,

- Choose View|Debug Windows|Call Stack from the menu bar.

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.

You can also use Run|Run Until Return to step out of the top-most function in the stack.

Customizing the colors of the execution point and breakpoints

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 Tools|Environment Options.
2. On the Environment Options dialog box, select the Colors tab.
3. From the Element list, select the following options that you want to change:
   Execution point
   Enabled Break
   Disabled Break
   Invalid Break
4. Select the background (BG) and foreground (FG) colors you want.

Handling exceptions in the Debugger

See also

You can control the way exceptions are handled while you debug your program. In addition, most hardware exceptions are treated as language exceptions. The IDE traps the hardware exceptions generated by your application, and you can gracefully recover rather than having your program execution end with a system crash.

If a hardware or language exception occurs while you are debugging a an application, your program halts and the Exception dialog box displays. If you choose OK, you can continue to run your program if your program handles the exception.

To pause the program run when an exception occurs,

1 Choose Tools|Debugger Options.

2 Choose the Language Exceptions or the OS Exceptions tab.

For OS Exceptions:

1 Find the type of exception in the Exceptions scroll box.

2 In the Handled By box click Debugger.

3 In the On Resume box, specify whether you want the IDE to continue to handle the exception when the program resumes.

To add a new type of exception not listed in the Exceptions scroll box, click Add. Specify the low and high range for the new exception and click OK.

The IDE displays the Exception dialog box when an exception is generated. When you choose OK to close the dialog box, the IDE opens the Code editor with the execution point positioned on the location of the exception (if no corresponding source is available, a checkbox “View CPU” appears on a dialog. If you want the CPU view opened, check the box and click OK.).

Debugging multi-threaded applications

See also

The integrated debugger supports debugging multi-thread programs in both Windows NT and Windows 95. Only a single thread, however, can be “active” at a given time. The active thread is the one that responds to debugger commands such as stepping and expression evaluation.

The Call Stack, the CPU, the Watch, and the Local Variables windows are “thread aware,” meaning that they display information based on a particular thread.

You can specify the active thread in the following ways:

- Choose View|Debug Windows|Threads from the menu bar, then select a Thread ID listed in the Thread Status window, right-click and select Make Current.
- Right-click the CPU window and choose Change Thread, then select a Thread ID listed in the Select a Thread dialog box

Debugging class member functions

See also

If you use classes in your programs, you can still use the integrated debugger to step through the member functions in your code. The debugger handles member functions the same way it would step through functions in a program that is not object-oriented.

Debugging dynamic link libraries

See also

The following topics cover issues when debugging DLLs.

Specifying the host EXE

When debugging a DLL, you don't need to add the host .EXE to a project to debug it. You can specify a pathname to the .EXE by selecting Run|Parameters and entering the path to the .EXE in the Host application edit box. Press the Load button to load the .EXE in the debugger.

Using Module load breakpoints when debugging .DLLs

Use Module load breakpoints to halt an application when it loads a specified .DLL. To set a Module load breakpoint either:

- Select either Run|Add Breakpoint|Module Load Breakpoint
- Choose View|Debug Windows|Modules to display the Modules window and right-click anywhere in the upper-left pane and select Add Module

Then in the Add Module dialog box, enter the module name of the .DLL or click Browse to find the .DLL. Click OK. When the application loads the specified .DLL, the application will halt.

Setting a debug source path

The debug source path is specified under Project|Options|Directories\Conditionals. Debug source paths for modules in the current project, or project group, are automatically set. If you are debugging modules (EXEs, DLLs) in different projects or projects groups, you need to add the debug source path for each module that is not part of the current project group.

Locating TDS files

TDS files must be in the same directory as the corresponding DLL or EXE.

Alignment palette context menu

The Alignment palette context menu contains the following commands:

Stay On Top

Show Hints

Hide

Help

Stay On Top **(Alignment palette context menu)**

Choose Stay On Top from the Alignment palette context menu to keep the Alignment palette in front of all other windows and dialog boxes.

Component palette context menu

The component palette context menu enables you to edit or rearrange the components on the component palette. You can also use the component palette context menu to hide the component palette.

Properties

Show Hints

Hide

Help

To display the component palette context menu,

- Right-click anywhere on the component palette.

Properties (Component palette context menu)

Choose Properties from the component palette context menu to open the Palette page of the Tools| Environment Options dialog box.

Use this dialog box to rearrange the components on the component palette.

Web page editor

The Web page editor lets you add Web items to a MIDAS page producer and view the resulting HTML document. The Web items generate the HTML that translates the <#FORMS> tag in the MIDAS page producer's default template.

To display the Web page Editor, double click on a TMidasPageProducer component or click the ellipsis button next to its WebPageItems property.

Note: You must have Internet Explorer 4 or better installed to use the Web page editor.

Parent Components

The upper left pane in the Web page editor displays the hierarchy of Web items that produce HTML for the MIDAS page producer. The subitems of each Web item produce HTML that the Web item uses as part of its own generated HTML. Each type of Web item can only contain certain classes of subitems. For example, the MIDAS page producer itself can only contain components that generate an HTML form.

Web items that do not use subitems are not displayed in the Parent Components pane.

When you select a Web item in the hierarchy, you can

- Change its properties using the Object Inspector.
- Add subitems by clicking the New Item button on the toolbar to display the Add Web Component dialog box. You can also display this dialog by pressing the Insert key or by right-clicking and choosing New Component from the context menu.
- Delete the item and all its subitems by clicking the Delete button on the toolbar. You can also delete items by pressing the Del key or by right-clicking and choosing Delete from the context menu.
- Cut or Copy the item to the clipboard by right-clicking and choosing the appropriate menu item.
- Paste an appropriate Web item from the clipboard to appear as a subitem of the selected item.
- If the Web item is a FieldGroup or QueryFieldGroup, add the field or parameter values it represents by right-clicking and choosing the appropriate menu item.
- View the subitems of the selected item in the Child Components pane.

Child Components

The upper right pane in the Web page editor displays the subitems of the currently selected item in the Web item hierarchy. Note that this is the only place you can see Web items that do not have subitems of their own.

When you select a Web item in the Child Components pane, you can

- Change its properties using the Object Inspector.
- Change its position in its parent's list of Web items. You can change its position either by clicking the up and down buttons on the toolbar or by right-clicking and choosing Move Up or Move Down.
- Cut or Copy the item to the clipboard by right-clicking and choosing the appropriate menu item.

Browser pane

The Browser pane appears in the lower portion of the dialog on the Browser tab page. This pane shows you how the generated HTML document looks in Internet Explorer. At the top of the display you can see warnings that describe any problems detected when generating the HTML document.

HTML pane

The HTML pane appears in the lower portion of the dialog on the HTML tab page. This pane shows you the generated HTML document. At the top of the display you can see warnings that describe any problems detected when generating the HTML document, except that instead of XML data packets you see a <#DATAPACKET> tag.

You can copy any or all of the HTML in this pane by right-clicking and choosing Copy when the desired HTML is selected. This is useful if you want to use the generated HTML as an HTML template (replacing the HTMLDoc property).

Add Web Component dialog box

The Add Web Component dialog box lets you add a Web item as the subitem of another component that generates HTML in a MIDAS Web application.

To display the Add Web Component dialog box, click the New Item button on the toolbar of the Web page editor.

The Add Web Component dialog box lists all the types of Web item that can be used by the currently selected Web item in the Web page editor.

To select a single item: click with your left mouse button or navigate using the arrow keys.

To select a contiguous set of items: click on the first and list items while pressing the shift key or use the arrow keys while pressing the shift key.

To select multiple items that are not next to each other: click on the items while pressing the control key.

When you press OK, the selected items are added to the currently selected item in the Parent Components pane of the Web page editor.

Add Field Controls dialog box

The Add Field Controls dialog box lets you specify the fields or parameters displayed in the HTML generated by a TFieldGroup or TQueryFieldGroup component. When you add fields or parameters to the TFieldGroup or TQueryFieldGroup component, it automatically replaces its Web items to represent each field or parameter in a labeled single-line edit control.

To display the Add Field Controls dialog box, right click a FieldGroup or QueryFieldGroup in the Web page editor and choose Add Fields or Add Parameters. (Only QueryFieldGroup components have an Add Parameters command).

The Add Field Controls dialog box lists all the fields or parameters maintained by the XML broker and dataset associated with the selected FieldGroup or QueryFieldGroup. Select the fields or parameters you want to display in the HTML form.

To select a single item: click with your left mouse button or navigate using the arrow keys.

To select a contiguous set of items: click on the first and list items while pressing the shift key or use the arrow keys while pressing the shift key.

To select multiple items that are not next to each other: click on the items while pressing the control key.

When you press OK, FieldGroup or QueryFieldGroup replaces its Web items with components that represent each field or parameter in a labeled single-line edit control.

Web page editor context menu

The contents of the Web page editor context menu vary, depending on what is selected in the Web page editor. The following table lists the menu commands and indicates what they do.

Command	What it does
New Component	Displays the Add Web Component dialog box, where you can add subitems to the currently selected Web item in the Parent Components pane.
Restore Defaults	Restores the default property settings to the currently selected items in the Child Components pane. This command only applies to components that display a field value. It restores the Caption property to the DisplayName of the associated field component, and (if the HTML control is a radio group, text edit control, or text area control) the DisplayWidth property to the field's DisplayWidth property.
Add All Params	Adds components to display every parameter of the associated XML broker as a subitem to a selected QueryFieldGroup component. The QueryFieldGroup must have an associated XML broker.
Add Params	Displays the Add Field Controls dialog box, where you can specify which parameters are displayed by a QueryFieldGroup component. The QueryFieldGroup must have an associated XML broker.
Add All Fields	Adds components to display every field of the associated dataset as a subitem to a selected FieldGroup or QueryFieldGroup component. The FieldGroup or QueryFieldGroup must have an associated XML broker.
Add Fields	Displays the Add Field Controls dialog box, where you can specify which fields are displayed by a FieldGroup or QueryFieldGroup component. The FieldGroup or QueryFieldGroup must have an associated XML broker.
Move Up	Moves the Web item that is selected in the Child Components pane up one position. The order of subitems determines the order in which the HTML they generate is arranged.
Move Down	Moves the Web item that is selected in the Child Components pane down one position. The order of subitems determines the order in which the HTML they generate is arranged.
Cut	Cuts the currently selected Web item and all its descendants to the clipboard. This can apply to a Web item in the Parent Components pane or an item in the Child Components pane.
Copy	Copies the current selection to the clipboard. This can apply to a Web item in the Parent Components pane or an item in the Child Components pane.
Paste	Pastes a Web item on the clipboard so that it becomes a subitem of the currently selected item in the Parent Components pane. If the clipboard does not contain a Web item, or if the Web item in the clipboard can't act as a child of the selected item, Paste generates an error message.
Delete	Deletes the currently selected Web item and all of its descendants. This can apply to a Web item in the Parent Components pane or an item in the Child Components pane.
Select All	When focus is in the Parent Components pane, the root node is selected. When focus is in the Child Components pane, all Web items in the list are selected.
Panel Descriptions	Shows or hides the labels for the Parent Components pane and the Child Components pane.
Tool bar	Shows or hides the tool bar at the top of the Web page editor.

HTML pane context menu

The following table lists the commands in the context menu of the HTML pane of the Web page editor.

Command	What it does
Copy	Copies the currently selected HTML to the clipboard.
Select All	Selects all of HTML in the window, so that it can be copied to the clipboard.
Panel Descriptions	Shows or hides the labels for the Parent Components pane and the Child Components pane.
Tool bar	Shows or hides the tool bar at the top of the Web page editor.

Web page editor toolbar menu

This menu appears when you right click on the toolbar of the Web page editor. It contains a single item

Text Labels	Displays or hides the text labels on tool buttons.

Open Styles file dialog

To open this dialog, click the Ellipsis button on a TMidasPageProducer's StylesFile property in the Object Inspector.

Use the Open Styles file dialog box to add a style sheet definition to the HTML document created by a MIDAS page producer. Select a file where each line defines a style by giving a style selector followed by a set of attributes in curly braces. These definitions can define styles for standard HTML elements such as

```
H2 B  {color: red}
```

or they can define styles that you name, such as

```
.MyStyle  {font-family: arial; font-weight: bold; font-size: 18px }
```

Note that user-defined style names must begin with a dot.

Open Styles file dialog box

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 type wildcards to use as filters in the Files list box.
Files of Type	Choose the type of file you want to open; the default file type is Text file (*.txt). All files in the current directory of the selected type appear in the Files list box.
Up One Level	Click this button to move up one directory level from the current directory.
Create New Folder	Click this button to create a new subdirectory in the current directory.
List	Click this button to view a list of files and directories in the current directory.
Details	Click this button to view a list of files and directories along with time stamp, size, and attribute information.

New Web Server Application dialog box

See also

Use the New Web Server Application dialog box to specify the type of server your Web server application will work with. After choosing the type of Web server application, click OK to create a new project configured to use Internet components and containing an empty Web module.

To bring up the New Web Server Application dialog box:

1. Choose File|New to open the New Items dialog box.
2. Choose the tab labeled New.
3. Select the Web Server Application item in the list view.

The New Web Server Application Options

ISAPI/NSAPI Dynamic Link Library

ISAPI and NSAPI Web server applications are DLLs that are loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by TISAPIApplication. Each request message is handled in a separate execution thread.

Selecting this type of application adds the library header of the project files and required entries to the uses list and exports clause of the project file.

CGI standalone executable

A CGI standalone Web server application is a console application that receives client request information on standard input and passes the results back to the server on standard output. This data is evaluated by TCGIApplication. Each request message is handled by a separate instance of the application.

Selecting this type of application adds the required entries to the uses clause of the project file and adds the appropriate $APPTYPE directive to the source.

Win-CGI standalone executable

A Win-CGI standalone Web server application is a Windows application that receives client request information from a configuration settings (INI) file written by the server and writes the results to a file that the server passes back to the client. The INI file is evaluated by TCGIApplication. Each request message is handled by a separate instance of the application.

Selecting this type of application adds the required entries to the uses clause of the project file and adds the appropriate $APPTYPE directive to the source.

Response Editor dialog box

See also

This dialog box lets you to define the contents and format of an HTTP response message for the TQueryTableProducer or TDataSetTableProducer component.

To display the Response Editor dialog box, place a query table producer or dataset table producer on a form. Right-click the component and choose Response Editor from the menu.

Dialog box options

Field list

The field list in the upper right of this dialog box shows the Field Name and Field Type for each field in the response table. You can use the Object Inspector to change properties for the highlighted field.

Table Properties

These are the properties of the overall response table.

Align
THTMLTableAttributes.Align, the horizontal alignment of the table within the HTML document.

Border
THTMLTableAttributes.Border, the width of the table border; -1 indicates no border will be drawn.

BgColor
THTMLTableAttributes.BgColor, the background color of the HTML table.

Cellpadding
THTMLTableAttributes.CellPadding, the amount of space to leave around the contents of each cell in the HTML table. A value of -1 indicates that the web browser should decide how to pad the cells of the HTML table.

CellSpacing
THTMLTableAttributes.CellSpacing, the amount of space to leave between cells in the HTML table. A value of -1 indicates that the web browser should decide how to separate the cells of the HTML table.

Width
THTMLTableAttributes.Width, the width of the entire HTML table as a percentage of the width of the web browser window. The default, 100, means the table will span the entire browser window.

HTML listing

At the bottom of the dialog, you can see the generated HTML that reflects your current settings.

Add button

Adds a column to the response table. Use the Object Inspector to enter the title and choose a field.

Delete button

Removes the selected column from the table.

Move Up/Down buttons

Change the order of columns in the table by moving the selected column up (toward the left edge of the table) and down (toward the right).

Add All Fields button

Creates a column for every field in the dataset (query) bound to the query table producer.

Restore Defaults

Restores default property settings for the selected column.

Open HTML file dialog

See also

To open this dialog, click the ellipsis button on a page producer's HTMLFile property in the Object Inspector.

Use the Open HTML file dialog box to assign an HTML template to the page producer. The HTML template contains a combination of HTML and special HTML-transparent tags. An HTML-transparent tag has the form

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

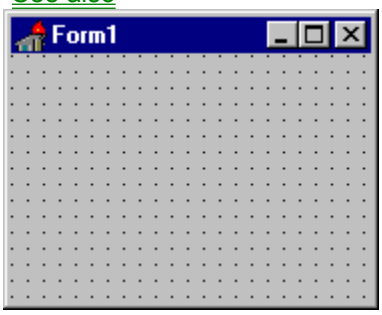
The Content method of the page producer translates the HTML-transparent tags into HTML.

Open HTML file dialog box

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 type wildcards to use as filters in the Files list box.
Files of Type	Choose the type of file you want to open; the default file type is HTML file (*.htm, *.html). All files in the current directory of the selected type appear in the Files list box.
Up One Level	Click this button to move up one directory level from the current directory.
Create New Folder	Click this button to create a new subdirectory in the current directory.
List	Click this button to view a list of files and directories in the current directory.
Details	Click this button to view a list of files and directories along with time stamp, size, and attribute information.

About forms

See also



Forms are the foundation of all Delphi applications. The form is a component. You place other components onto the form's client area 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.

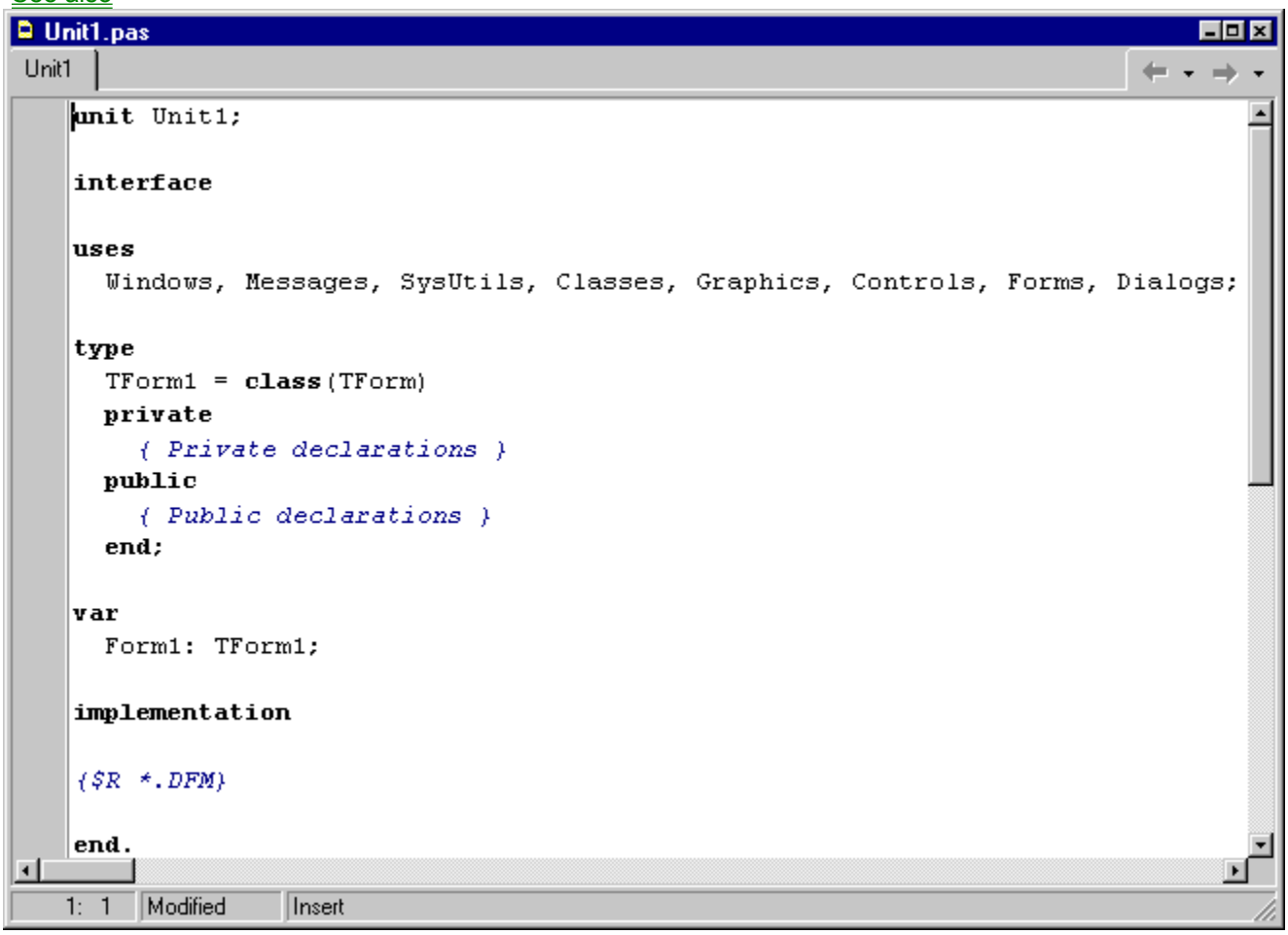
Client area
Enables you to view or modify any part of a form in the active window. Use the grid of dots to align objects on the form.

Opening a context menu

- 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
- Code Insight tools
- Code browser
- Module navigation
- Code Explorer
- Class completion

Many commands are available on the Code editor context menu.

To customize the Code editor, use the Tools|Environment Options dialog box.

To get Help on a token in the Code editor, place your cursor on that token and press 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

To view a different file in the Code editor

- Click its associated tab.

To create and use a keyboard macro

- Press Ctrl+Shift+R to begin recording a macro. Enter keystrokes, then press Ctrl+Shift+R to finish and save the macro.
- To play back the macro, press Ctrl+Shift+P.

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

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 runtime.
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, see the following topics:

Opening the Menu Designer

Editing Menu Items Without Opening the Menu Designer

Menu Designer context menu

Using Menu Templates

Importing Menus from Resource Files

Accessing and Editing Menus at Runtime

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

Designing menus

See also

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 runtime.

Your code can also change menus at runtime, to provide more information or options to the user. For information, see Accessing and Editing Menus at Runtime..

About the Project Manager

See also

The Project Manager allows you to combine projects that work together into a single project group. Project groups allow you to organize and work on interdependent projects such as separate tiers in a multi-tiered application or DLLs and executables that work together.

With the Project Manager, you can easily visualize how all your project files are related. Also, you can select any file displayed, right-click, and perform any number of project management tasks, such as opening, adding or removing files, and compiling your projects. With project groups, you can add and remove projects from the project group and compile all projects within the group at one time.

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.

When you start Delphi, you can open the Project Manager by choosing View|Project Manager. If you save your desktop settings, you can open the Project Manager window 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.

Closing the Project Manager

- Click on the Close menu box (the X on the far right).
- Choose Close from the Control menu (right-click in the window title bar).

See also

Project Manager context menus
Compiling, building, and running projects
Setting project options
Working with projects
Creating a backup of an entire project

Project Browser

See also

The Project Browser lists the units, classes, types, properties, methods, variables, and routines declared or used in a project. With the Project Browser, you can

- view class hierarchies in a tree diagram.
- list the units that a project contains or uses.
- see the identifiers declared or used in a project.
- find declarations and references in source code.

To open the Project Browser, choose View|Browser.

The information displayed in the Project Browser is dependent on several Browser options and compiler settings. Before using the Browser, save and compile your project.

The Project Browser has two resizable panes: the Inspector pane (on the left) and the Details pane (on the right). The Inspector pane has three tabs:

Classes shows classes in a hierarchical diagram.

Units lists units, identifiers declared in each unit, and the other units that use and are used by each unit.

Globals lists classes, types, properties, methods, variables, and routines.

The Details pane (which is equivalent to the Symbol Explorer) provides more information about the item selected in the Inspector pane. It displays a different set of tabs depending on the kind of item selected. Detail-pane tabs include Scope, Inheritance, and References:

Scope lists identifiers declared in the class or unit selected in the Inspector pane.

Inheritance displays a local hierarchy tree for the class selected in the Inspector pane.

References lists file names and line numbers where the item selected in the Inspector pane appears in the current project's source code. You can double-click any reference to jump to that line in the Code editor. You must enable the correct compiler options for the References page to work.

To show or hide the Details pane, right-click in the Project Browser and choose Details. To make the Project Browser a dockable window, right-click and select Dockable.

The Project Browser supports incremental searching. To search for an item in a tree diagram, just type its name.

The Project Browser uses the same icons as the Code Explorer to identify the items in its tree diagrams.

For more information about how the Project Browser parses source code, see How the Project Browser works.

How the Project Browser works

See also

The Project Browser relies on the compiler for the items listed in its tree diagrams. Strictly speaking, each item represents a *symbol* from the compiler's *symbol table*. A single Object Pascal identifier may represent different symbols in different contexts. On the References page of the Details pane, the Browser lists occurrences of the selected *symbol*, not the selected identifier.

In practical terms, this affects the way the Browser identifies references to inherited class members. For example, consider the following source code.

```
type
  TRectangle = class
    procedure Draw; virtual;
  end;
  TSquare = class(TRectangle)
    procedure Draw; override;
  end;
...

var
  A: TRectangle;
  B: TSquare;
begin
  A := TSquare.Create;
  B := TSquare.Create;
  A.Draw;  // a reference to TRectangle.Draw (not TSquare.Draw)
  B.Draw;  // a reference to TSquare.Draw (not TRectangle.Draw)
end;
```

The first call to the Draw method (`A.Draw`) appears on the References page for TRectangle.Draw, while the second call to the Draw method (`B.Draw`) appears on the References page for TSquare.Draw. Both calls, however, execute the method in TSquare.

Project Browser options

See also

In the Environment Options dialog you can change settings that determine which items appear in the Project Browser. To change these settings, choose Tools|Environment Options and click the Explorer tab, or right-click in the Project Browser and choose Properties.

- Under Browser Scope, select
- Project Symbols Only to view items from units in the current project only, or
- All Symbols to view items from all units (including the VCL) used directly or indirectly by the current project.
- Under Explorer Categories, you can control how source elements are grouped in Project Browser tree diagrams. If a category is checked, elements in that category are grouped under a single node. If a category is unchecked, each element in that category is displayed independently on the diagram's trunk.
- Under Initial Browser View, select the tab you want to see most often (Classes, Units, or Globals).

In addition to these settings, several compiler options affect the Project Browser.

Compiler settings that affect the Project Browser

See also

Several compiler settings determine which items are displayed in the Project Browser. To change these settings, choose Project|Options and click the Compiler tab; all options that affect the Project Browser appear under Debugging. After changing compiler settings, recompile your project.

Option	Effect on Project Browser
Debug Information	Displays items declared in the **implementation** sections of units. (The Project Browser always displays items from **interface** sections.)
Local Symbols	Enables the References page of the Details pane. (Debug Information and Reference Info must also be selected.)
Reference Info	Enables the References page of the Details pane. (Debug Information and Local Symbols must also be selected.)
Definitions Only	Turn this option **off** (uncheck it) if you want the References page to show all occurrences of an item in source code. If Definitions Only is selected, the References page displays only the declaration for each identifier.

Symbol Explorer

See also

The Symbol Explorer provides information about units, classes, types, properties, methods, variables, and routines. It is functionally equivalent to the Details pane of the Project Browser.

There are three ways to open the Symbol Explorer:

Double-click on any item in the Project Browser.

Choose Search|Browse Symbol from the main menu, then enter an identifier in the Browse Symbol dialog and click OK.

In the Code editor, place the cursor on any identifier, right-click, and choose Browse Symbol at Cursor. The Browse Symbol dialog opens with the identifier already entered.

The command-line compiler

Section topics

Delphi's command-line compiler (DCC32.EXE) lets you invoke all the functions of the IDE compiler (DELPHI.EXE) from the DOS command line. Run the command-line compiler from the DOS prompt using the syntax:

```
DCC32 [options] filename [options]
```

where options are zero or more parameters that provide information to the compiler and filename is the name of the source file to compile. If you type DCC32 alone, it displays a help screen of command-line options and syntax.

If filename does not have an extension, the command-line compiler assumes .DPR, then .PAS if no .DPR is found. If the file you're compiling to doesn't have an extension, you must append a period (.) to the end of the filename.

If the source text contained in filename is a program, the compiler creates an executable file named filename.EXE. If filename contains a library, the compiler creates a file named filename.DLL. If filename contains a package, the compiler creates a file named filename.BPL. If filename contains a unit, the compiler creates a unit file named filename.DCU.

You can specify a number of options for the command-line compiler. An option consists of a slash (/) immediately followed by an option letter. In some cases, the option letter is followed by additional information, such as a number, a symbol, or a directory name. Options can be given in any order and can come before or after the file name.

Command-line compiler options

Section topics

The IDE lets you set various options through the menus; the command-line compiler gives you access to these options using the slash (/) delimiter. You can also precede options with a hyphen (-) instead of a slash (/), but those options that start with a hyphen must be separated by blanks. For example, the following two command lines are equivalent and legal:

```
DCC -IC:\DELPHI -DDEBUG SORTNAME -$R- -$U+
DCC /IC:\DELPHI/DDEBUG SORTNAME /$R-/$U+
```

The first command line uses hyphens with at least one blank separating options. The second uses slashes and no separation is needed.

The following table lists the command-line options. In addition to the listed options, all single-letter compiler directives can be specified on the command line, as described in the next topic.

Option	Description
/A*unit=alias*	Set unit alias
/B	Build all units
/CC	Console target
/CG	GUI target
/Ddefines	Define conditional symbol
/Epath	EXE directory
/Faddress	Find run-time error
/GS	Map file with segments
/GP	Map file with publics
/GD	Detailed map file
/H	Output hint messages
/Ipaths	Include directories
/J	Generate OBJ file
/JP	Generate C++ OBJ file
/K*address*	Set image base address
/LEpath	Package BPL directory
/LNpath	Package DCP directory
/LU*package*	Use packages
/M	Make modified units
/N*path*	DCU directory
/Opaths	Object directories
/P	Look for 8.3 file names also
/Q	Quiet compile
/Rpaths	Resource directories
/TXext	Target file extension
/Upaths	Unit directories
/V	Turbo Debugger debug information
/VN	Generate namespace debugging information in Giant format (used by C++Builder)
/VR	Generate RSM file for remote debugging

/W	Output warning messages
/Z	Disable implicit compilation

If you type DCC32 alone at the command line, a list of command-line compiler options appears on your screen.

Compiler directive options

Section topics

Delphi supports several compiler directives, all described in "Compiler directives." The /$ and /D command-line options allow you to change the default states of most compiler directives. Using /$ and /D on the command line is equivalent to inserting the corresponding compiler directive at the beginning of each source file compiled.

The switch directive option

The /$ option lets you change the default state of all of the switch directives. The syntax of a switch directive option is /$ followed by the directive letter, followed by a plus (+) or a minus (-). For example,

```
DCC32 MYSTUFF /$R-
```

compiles MYSTUFF.PAS with range-checking turned off, while

```
DCC32 MYSTUFF /$R+
```

compiles it with range checking turned on. Note that if a {$R+} or {$R-} compiler directive appears in the source text, it overrides the /$R command-line option.

You can repeat the /$ option in order to specify multiple compiler directives:

```
DCC32 MYSTUFF /$R-/$I-/$V-/$U+
```

Alternately, the command-line compiler lets you write a list of directives (except for $M), separated by commas:

```
DCC32 MYSTUFF /$R-,I-,V-,U+
```

Only one dollar sign ($) is needed.

Note that, because of its format, you cannot use the $M directive in a list of directives separated by commas.

The conditional defines option

The /D option lets you define conditional symbols, corresponding to the {$DEFINE symbol} compiler directive. The /D option must be followed by one or more conditional symbols separated by semicolons (;). For example, the following command line

```
DCC32 MYSTUFF /DIOCHECK;DEBUG;LIST
```

defines three conditional symbols, iocheck, debug, and list, for the compilation of MYSTUFF.PAS. This is equivalent to inserting

```
{$DEFINE IOCHECK}
{$DEFINE DEBUG}
{$DEFINE LIST}
```

at the beginning of MYSTUFF.PAS. If you specify multiple /D directives, you can concatenate the symbol lists. Therefore,

```
DCC32 MYSTUFF /DIOCHECK/DDEBUG/DLIST
```

is equivalent to the first example.

Compiler mode options

Section topics

A few options affect how the compiler itself functions. As with the other options, you can use these with either the hyphen or the slash format. Remember to separate the options with at least one blank.

The make (/M) option

The command-line compiler has built-in MAKE logic to aid in project maintenance. The /M option instructs command-line compiler to check all units upon which the file being compiled depends. Using this option results in a much quicker compile time.

A unit will be recompiled if:

- The source file for that unit has been modified since the unit file was created.
- Any file included with the $I directive, any .OBJ file linked in by the $L directive, or any .RES file referenced by the $R directive, is newer than the unit file.
- The interface section of a unit referenced in a uses statement has changed.

Units compiled with **/Z** option are excluded from the make logic.

If you were applying this option to the previous example, the command would be

```
DCC32 MYSTUFF /M
```

The build all (/B) option

Instead of relying on the /M option to determine what needs to be updated, you can tell command-line compiler to update all units upon which your program depends using the /B option. You can't use /M and /B at the same time. The /B option is slower than the /M option and is usually unnecessary.

If you were using this option in the previous example, the command would be

```
DCC32 MYSTUFF /B
```

The find error (/F) option

When a program terminates due to a runtime error, it displays an error code and the address at which the error occurred. By specifying that address in a /Faddress option, you can locate the statement in the source text that caused the error, provided your program and units were compiled with debug information enabled (via the $D compiler directive).

In order for the command-line compiler to find the run-time error with /F, you must compile the program with all the same command-line parameters you used the first time you compiled it.

As mentioned previously, you must compile your program and units with debug information enabled for the command-line compiler to be able to find run-time errors. By default, all programs and units are compiled with debug information enabled, but if you turn it off, using a {$D-} compiler directive or a /$D- option, the command-line compiler will not be able to locate run-time errors.

The use packages (/LU) option

Use the **z/LU** option to list additional runtime packages that you want to use in the application being compiled. Runtime packages already listed in Delphi's Project Options dialog need not be repeated on the command line.

The disable implicit compilation (/Z) option

The **/Z** option prevents packages and units from being implicitly recompiled later. With packages, it is equivalent to placing **{$ IMPLICITBUILD OFF}** in the .DPK file. Use **/Z** when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.

The target file extension (/TX) option

The **/TX** option lets you override the default extension for the output file. For example,

```
DCC32 MYSTUFF /TXSYS
```

generates compiled output in a file called MYSTUFF.SYS.

The quiet (/Q) option

The quiet mode option suppresses the printing of file names and line numbers during compilation. When the command-line compiler is invoked with the quiet mode option

```
DCC32 MYSTUFF /Q
```

its output is limited to the startup copyright message and the usual statistics at the end of compilation. If any errors occur, they will be reported.

Directory options

Section topics

Several options allow you to specify the directory lists used by the command-line compiler: include (**/I**), DCU input (unit search path, **/U**), resource (**/R**), object (**/O**), EXE and DCU output (**/E**), and DCU output (**/N**).

You can specify multiple directories, separated by semicolons, for some options. For example, this command line tells the command-line compiler to search for include files in C:\DELPHI\INCLUDE and D:\INC after searching the current directory:

```
DCC32 MYSTUFF /IC:\DELPHI\INCLUDE;D:\INC
```

If you specify multiple directives, the directory lists are concatenated. Therefore,

```
DCC32 MYSTUFF /IC:\DELPHI\INCLUDE /ID:\INC
```

is equivalent to the first example.

The EXE directory (/E) option

This option lets you tell the command-line compiler where to put the .EXE file it creates. It takes a directory path as its argument:

```
DCC32 MYSTUFF /EC:\DELPHI\BIN
```

You can specify only one EXE directory, which is also used for .DLL files. The **/E** option does not affect the location of .BPL (package) files.

If no such option is given, the command-line compiler creates .EXE files in the same directories as their corresponding source files.

The package BPL directory (/LE) option

This option lets you tell the command-line compiler where to put the .BPL file it creates when it compiles a package. The syntax is the same as **/E**.

The package DCP directory (/LN) option

This option lets you tell the command-line compiler where to put the .DCP file it creates when it compiles a package. The syntax is the same as **/E**.

The DCU directory (/N) option

This option lets you tell the command-line compiler where to put the .DCU files it creates when it compiles a unit. The syntax is the same as **/E**.

The include directories (/I) option

Delphi supports include files through the {$I filename} compiler directive. The /I option lets you specify a list of directories in which to search for include files.

The unit directories (/U) option

When you compile a program that uses units, the command-line compiler searches for unit files in the current directory. The /U option lets you specify additional directories in which to search for units.

The resource directories (/R) option

DCC32 searches for resource files in the current directory. The /R option lets you indicate additional directories where DCC32 should look for resource files.

The object files directories (/O) option

Using {$L filename} compiler directives, Delphi lets you link in .OBJ files containing machine code created by external assemblers or other compilers, such as Borland C++. The /O option lets you specify a list of directories in which to search for such .OBJ files.

Debug options

Section topics

The compiler has two sets of command-line options that enable you to generate external debugging information: the map file options and the debug info options.

The map file (/G) options

The /G option instructs the command-line compiler to generate a .MAP file that shows the layout of the .EXE file. Unlike the binary format of .EXE and .DCU files, a .MAP file is a legible text file that can be output on a printer or loaded into the editor. The /G option must be followed by the letter S, P, or D to indicate the desired level of information in the .MAP file. A .MAP file is divided into three sections:

- Segment
- Publics
- Line Numbers

/GS outputs only the Segment section, /GP outputs the Segment and Publics section, and /GD outputs all three sections. /GD also generates a .DRC file that contains tables of all string constants declared using the **resourcestring** keyword.

For modules (program and units) compiled in the {$D+,L+} state (the default), the Publics section shows all global variables, procedures, and functions, and the Line Numbers section shows line numbers for all procedures and functions in the module. In the {$D+,L-} state, only symbols defined in a unit's interface part are listed in the Publics section. For modules compiled in the {$D-} state, there are no entries in the Line Numbers section.

The debug info (/V) options

The /V options (/V, /VN. and /VR), which cause the compiler to generate debug information, can be combined on the command line.

The generate Turbo Debugger debug info (/V) option

When you specify the /V option on the command line, the compiler appends Turbo Debugger 5.0-compatible external debug information at the end of the .EXE file. Turbo Debugger includes both source- and machine-level debugging and powerful breakpoints.

Even though the debug information generated by /V makes the resulting .EXE file larger, it does not affect the actual code in the .EXE file, and does not require additional memory to run the program.

The extent of debug information appended to the .EXE file depends on the setting of the $D and $L compiler directives in each of the modules (program and units) that make up the application. For modules compiled in the {$D+,L+} state, which is the default, all constant, variable, type, procedure, and function symbols are known to the debugger. In the {$D+,L-} state, only symbols defined in a unit's interface section are known to the debugger. In the {$D-} state, no line-number records are generated, so the debugger cannot display source lines when you debug the application.

The IDE internal debugger does not use Turbo Debugger debug information. Because generating Turbo Debugger debug information almost doubles compile/link time, you should turn off Turbo Debugger debug information generation except when you're debugging the application in Turbo Debugger.

The generate namespace debug info (/VN) option

When you specify the /VN option on the command line, the compiler generates namespace debugging information in the Giant format used by C++Builder. This allows the C++ compiler to find Pascal symbols. Use this switch when you are compiling code that will be used by C++Builder.

The generate remote debug info (/VR) option

When you specify the /VR option on the command line, the compiler generates remote debugging information in an RSM file.

The DCC32.CFG file

Section topics See also

You can set up a list of options in a configuration file called DCC32.CFG, which will then be used in addition to the options entered on the command line. Each line in configuration file corresponds to an extra command-line argument inserted before the actual command-line arguments. Thus, by creating a configuration file, you can change the default setting of any command-line option.

The command-line compiler lets you enter the same command-line option several times, ignoring all but the last occurrence. This way, even though you've changed some settings with a configuration file, you can still override them on the command line.

When DCC32 starts, it looks for DCC32.CFG in the current directory. If the file isn't found there, DCC32 looks in the directory where DCC32.EXE resides.

Here's an example DCC32.CFG file, defining some default directories for include, object, and unit files, and changing the default states of the $O and $R compiler directives:

```
/IC:\DELPHI\INC;C:\DELPHI\SRC
/OC:\DELPHI\ASM
/UC:\DELPHI\UNITS
/$R+
/$O-
```

Now, if you type

```
DCC32 MYSTUFF
```

at the system prompt, DCC32 acts as if you had typed in the following:

```
DCC32 /IC:\DELPHI\INC;C:\DELPHI\SRC /OC:\DELPHI\ASM /UC:\DELPHI\UNITS /$R+
/$O- MYSTUFF
```

Compiler directives

See also

The following topics describe the compiler directives you can use to control the features of the Delphi compiler. Each compiler directive is classified as either a switch, parameter, or conditional compilation directive. Choose a directive from the list of compiler directives for detailed information.

A compiler directive is a comment with a special syntax. Delphi allows compiler directives wherever comments are allowed. A compiler directive starts with a $ as the first character after the opening comment delimiter, immediately followed by a name (one or more letters) that designates the particular directive. You can include comments after the directive and any necessary parameters.

Three types of directives are described in the following topics:

- **Switch directives** turn particular compiler features on or off. For the single-letter versions, you add either + or - immediately after the directive letter. For the long version, you supply the word "on" or "off."

  Switch directives are either global or local.

- Global directives affect the entire compilation and must appear before the declaration part of the program or the unit being compiled.
- Local directives affect only the part of the compilation that extends from the directive until the next occurrence of the same directive. They can appear anywhere.

  Switch directives can be grouped in a single compiler directive comment by separating them with commas with no intervening spaces. For example,

```
{$B+,R-,S-}
```

- **Parameter directives.** These directives specify parameters that affect the compilation, such as file names and memory sizes.
- **Conditional directives.** These directives control conditional compilation of parts of the source text, based on user-definable conditional symbols. See "Using conditional compilation directives" for information about using conditional directives.

All directives, except switch directives, must have at least one space between the directive name and the parameters. Here are some examples of compiler directives:

```
{$B+}
{$STACKCHECKS ON}
{$R- Turn off range checking}
{$I TYPES.INC}
{$M 32768,4096}
{$DEFINE Debug}
{$IFDEF Debug}
{$ENDIF}
```

You can insert compiler directives directly into your source code. You can also change the default directives for both the command-line compiler (DCC32.EXE) and the IDE (DELPHI32.EXE).

The Project|Options|Compiler dialog box contains many of the compiler directives; any changes you make to the settings there will affect all units whenever their source code is recompiled in subsequent compilations of that project. If you change a compiler switch and compile, none of your units will reflect the change; but if you Build All, all units for which you have source code will be recompiled with the new settings.

When using the command-line compiler, you can specify compiler directives on the command line (for example, DCC32 /$R+ MYPROG), or you can place directives in a configuration file (see "The DCC32.CFG file"). Compiler directives in the source code always override the command-line compiler directives and the IDE project options.

If you are working in the Delphi editor and want a quick way to see what compiler directives are in effect, press *Ctrl+O O*. Delphi will insert the current settings in the edit window at the top of your file.

Compiler directives (list)

See also

Align fields
Application type
Assert directives
Boolean short-circuit evaluation
Debug information
DEFINE directive
DENYPACKAGEUNIT directive
Description
DESIGNONLY directive
ELSE directive
ENDIF directive
Executable extension
Export symbols
Extended syntax
External symbols
Hints
HPP emit
Implicit Build
Imported data
IFDEF directive
IFNDEF directive
IFOPT directive
Image base address
Include file
Input/output checking
Link object file
Local symbol information
Long strings
Memory allocation sizes
Minimum enumeration size
Open String Parameters
Optimization
Overflow checking
Pentium-safe FDIV operations
Private symbol
Private unit
Range checking
Real48 compatibility
Resource file
RUNONLY directive
Runtime type information
Symbol declaration and cross-reference information
Type-checked pointers
UNDEF directive
Var-string checking
Warnings
Weak packaging
Windows stack frames
Writeable typed constants

Align fields

See also

Type	Switch
Syntax	{$A+} or {$A-} {$ALIGN ON} or {$ALIGN OFF}
Default	{$A+} {$ALIGN ON}
Scope	Local

Remarks

The **$A** directive controls alignment of fields in record types.

In the **{$A+}** state, fields in record types that are declared without the **packed** modifier are aligned. In the **{$A-}** state, fields in record types are never aligned. Record type field alignment is described in the Object Pascal Language Guide.

Regardless of the state of the **$A** directive, variables and typed constants are always aligned for optimal access. In the **{$A+}** state, execution will be faster.

Application type

See also

Type	Parameter
Syntax	{$APPTYPE GUI} or {$APPTYPE CONSOLE}
Default	{$APPTYPE GUI}
Scope	Global

Remarks

The **$APPTYPE** directive controls whether to generate a Win32 console or graphical UI application. See also **/C** compiler option in "/CC".

In the **{$APPTYPE GUI}** state, the compiler generates a graphical UI application. This is the normal state for a Delphi application.

In the **{$APPTYPE CONSOLE}** state, the compiler generates a console application. When a console application is started, Windows creates a text-mode console window through which the user can interact with the application. The *Input* and *Output* standard text files are automatically associated with the console window in a console application.

The *IsConsole* Boolean variable declared in the *System* unit can be used to detect whether a program is running as a console or graphical UI application.

The **$APPTYPE** directive is meaningful only in a program. It should not be used in a library, unit, or package.

Assert directives

See also

Type	Switch
Syntax	{$C+} or {$C-} {$ASSERTIONS ON} or {$ASSERTIONS OFF}
Default	{$C+} {$ASSERTIONS ON}
Scope	Local

Remarks

The **$C** directive enables or disables the generation of code for assertions in a source file. {**$C+**} is the default.

Since assertions are not usually used at runtime in shipping versions of a product, compiler directives that disable the generation of code for assertions are provided. {**$C-**} will disable assertions.

Boolean short-circuit evaluation

See also

Type	Switch
Syntax	{$B+} or {$B-} {$BOOLEVAL ON} or {$BOOLEVAL OFF}
Default	{$B-} {$BOOLEVAL OFF}
Scope	Local

Remarks

The $B directive switches between the two different models of code generation for the **and** and **or** Boolean operators.

In the {$B+} state, the compiler generates code for complete Boolean expression evaluation. This means that every operand of a Boolean expression built from the and and or operators is guaranteed to be evaluated, even when the result of the entire expression is already known.

In the {$B-} state, the compiler generates code for short-circuit Boolean expression evaluation, which means that evaluation stops as soon as the result of the entire expression becomes evident in left to right order of evaluation.

For further details, see the section "Boolean operators" in the Object Pascal Language Guide.

Debug information

See also

Type	Switch
Syntax	{$D+} or {$D-} {$DEBUGINFO ON} or {$DEBUGINFO OFF}
Default	{$D+} {$DEBUGINFO ON}
Scope	Global

Remarks

The $D directive enables or disables the generation of debug information. This information consists of a line-number table for each procedure, which maps object-code addresses into source text line numbers.

For units, the debug information is recorded in the unit file along with the unit's object code. Debug information increases the size of unit file and takes up additional memory when compiling programs that use the unit, but it does not affect the size or speed of the executable program.

When a program or unit is compiled in the {$D+} state, Delphi's integrated debugger lets you single-step and set breakpoints in that module.

The Include debug info (Project|Options|Linker) and Map file (Project|Options|Linker) options produce complete line information for a given module only if you've compiled that module in the {$D+} state.

The $D switch is usually used in conjunction with the $L switch, which enables and disables the generation of local symbol information for debugging. See also "The generate Turbo Debugger debug info (/V) option" and "Symbol cross-reference information."

DEFINE directive

See also

Type	Conditional compilation
Syntax	{$DEFINE name}

Remarks

Defines a conditional symbol with the given name. The symbol is recognized for the remainder of the compilation of the current module in which the symbol is declared, or until it appears in an {$UNDEF name} directive. The {$DEFINE name} directive has no effect if name is already defined.

DENYPACKAGEUNIT directive

See also

Type	Switch
Syntax	{$DENYPACKAGEUNIT ON} or {$DENYPACKAGEUNIT OFF}
Default	{$DENYPACKAGEUNIT OFF}
Scope	Local

Remarks

The **{$DENYPACKAGEUNIT ON}** directive prevents the unit in which it appears from being placed in a package.

Description

See also

Type	Parameter
Syntax	{$DESCRIPTION 'text'}
Scope	Global

Remarks

The $D directive inserts the text you specify into the module description entry in the header of an .EXE, .DLL, or .BPL. Traditionally the text is a name, version number, and copyright notice, but you may specify any text of your choosing. For example,

```
{$D 'My Application version 12.5'}
```

The string can't be longer than 256 bytes. The description is usually not visible to end users. To mark you executable files with descriptive text, version and copyright information for the benefit of end users, use version info resources.

Note: The text description must be included in quotes.

DESIGNONLY directive

See also

Type	Switch
Syntax	{$DESIGNONLY ON} or {$DESIGNONLY OFF}
Default	{$DESIGNONLY OFF}
Scope	Local

Remarks

The **{DESIGNONLY ON}** directive causes the package where it occurs to be compiled for installation in the Delphi IDE. For more information, see "Package-specific compiler directives" in the Object Pascal Language Guide.

Place the **DESIGNONLY** directive only in .DPK files.

ELSE directive

See also

Type	Conditional compilation
Syntax	{$ELSE}

Remarks

Switches between compiling and ignoring the source text delimited by the previous {$IFxxx} and the next {$ENDIF}.

ENDIF directive

See also

Type	Conditional compilation
Syntax	{$ENDIF}

Remarks

Ends the conditional compilation initiated by the last {$IFxxx} directive.

Executable extension

See also

Type	Parameter
Syntax	{$E extension}

The **$E** directive sets the extension of the executable file generated by the compiler. It is often used in conjunction with the resource-only DLL mechanism.

For example, placing **{$E DEU}** in a library module produces a DLL with the .DEU extension: filename.DEU. If you create a library module that simply references German forms and strings, you could use this directive to produce a DLL with the .DEU extension. The startup code in the runtime library looks for a DLL whose extension matches the locale of the system—for German settings, it looks for .DEU—and loads resources from that DLL.

Export symbols

See also

Type	Switch
Syntax	{$ObjExportAll On} or {$ObjExportAll Off}
Default	{$ObjExportAll Off}
Scope	Global

The **{$ObjExportAll On}** directive exports all symbols in the unit file in which it occurs. This allows C++Builder to create packages containing Pascal-generated object files.

Extended syntax

See also

Type	Switch
Syntax	{$X+} or {$X-} {$EXTENDEDSYNTAX ON} or {$EXTENDEDSYNTAX OFF}
Default	{$X+} {$EXTENDEDSYNTAX ON}
Scope	Global

Remarks

The $X directive enables or disables Delphi's extended syntax:

- Function statements. In the {$X+} mode, function calls can be used as procedure calls; that is, the result of a function call can be discarded. Generally, the computations performed by a function are represented through its result, so discarding the result makes little sense. However, in certain cases a function can carry out multiple operations based on its parameters and some of those cases might not produce a useful result. When that happens, the {$X+} extensions allow the function to be treated as a procedure.
- Null-terminated strings. A {$X+} compiler directive enables Delphi's support for null-terminated strings by activating the special rules that apply to the built-in PChar type and zero-based character arrays. For more details about null-terminated strings, see "Using null-terminated strings" in the Object Pascal Language Guide.

Note: The **$X** directive is provided for backwards compatibility with previous versions of Borland Pascal. You should not use the {**$X-**} mode when writing Delphi applications.

External symbols

See also

Type	Parameter
Syntax	{$EXTERNALSYM identifier}

The EXTERNALSYM directive prevents the specified Pascal symbol from appearing in header files generated for C++Builder. If an overloaded routine is specified, all versions of the routine are excluded from the header file.

Hints

See also

Type	Switch
Syntax	{$HINTS ON} or {$HINTS OFF}
Default	{$HINTS ON}
Scope	Local

Remarks

The **$HINTS** directive controls the generation of hint messages by the compiler.

In the {**$HINTS ON**} state, the compiler issues hint messages when detecting unused variables, unused assignments, **for** or **while** loops that never execute, and so on. In the {**$HINTS OFF**} state, the compiler generates no hint messages.

By placing code between {**$HINTS OFF**} and {**$HINTS ON**} directives, you can selectively turns off hints that you don't care about. For example,

```
{$HINTS OFF}
procedure Test;
var
 I: Integer;
begin
end;
{$HINTS ON}
```

Because of the **$HINTS** directives the compiler will not generate an unused variable hint when compiling the procedure above.

HPP emit

See also

Type	Parameter
Syntax	{$HPPEMIT 'string'}

The HPPEMIT directive adds a specified symbol to the header file generated for C++Builder. Example: `{$HPPEMIT 'typedef double Weight' }`.

HPPEMIT directives are output into the "user supplied" section at the top of the header file in the order in which they appear in the Pascal file.

Implicit Build

See also

Type	Switch
Syntax	{$IMPLICITBUILD ON} or {$IMPLICITBUILD OFF}
Default	{$IMPLICITBUILD ON}
Scope	Global

Remarks

The **{$IMPLICITBUILD OFF}** directive, intended only for packages, prevents the source file in which it occurs from being implicitly recompiled later. Use **{$IMPLICITBUILD OFF}** in .DPK files when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed. Use of **{$IMPLICITBUILD OFF}** in unit source files is not recommended. See also "The disable implicit compilation (/Z) option."

Imported data

See also

Type	Switch
Syntax	{$G+} or {$G-} {$IMPORTEDDATA ON} or {$IMPORTEDDATA OFF}
Default	{$G+} {$IMPORTEDDATA ON}
Scope	Local

Remarks

The **{$G-}** directive disables creation of imported data references. Using **{$G-}** increases memory-access efficiency, but prevents a packaged unit where it occurs from referencing variables in other packages.

IFDEF directive

See also

Type	Conditional compilation
Syntax	{$IFDEF name}

Remarks

Compiles the source text that follows it if name is defined.

IFNDEF directive

See also

Type	Conditional compilation
Syntax	{$IFNDEF name}

Remarks

Compiles the source text that follows it if name is not defined.

IFOPT directive

See also

Type	Conditional compilation
Syntax	{$IFOPT switch}

Remarks

Compiles the source text that follows it if switch is currently in the specified state. switch consists of the name of a switch option, followed by a + or a - symbol. For example, the construct

```
{$IFOPT R+}
 Writeln('Compiled with range-checking');
{$ENDIF}
```

will compile the *Writeln* statement if the $R option is currently active.

Image base address

See also

Type	Parameter
Syntax	{$IMAGEBASE number}
Default	{$IMAGEBASE $00400000}
Scope	Global

The **$IMAGEBASE** directive controls the default load address for an application, DLL, or BPL. The *number* argument must be a 32-bit integer value that specifies image base address. The *number* argument must be greater than or equal to $00010000, and the lower 16 bits of the argument are ignored and should be zero.

When a module (application or library) is loaded into the address space of a process, Windows will attempt to place the module at its default image base address. If that does not succeed, that is if the given address range is already reserved by another module, the module is *relocated* to an address determined at runtime by Windows.

There is seldom, if ever, any reason to change the image base address of an application. For a library, however, it is recommended that you use the **$IMAGEBASE** directive to specify a non-default image base address, since the default image base address of $00400000 will almost certainly never be available. The recommended address range of DLL images is $40000000 to $7FFFFFFF. Addresses in this range are always available to a process in both Windows NT and Windows 95.

When Windows succeeds in loading a DLL (or BPL) at its image base address, the load time is decreased because relocation fixups do not have to be applied. Furthermore, when the given address range is available in multiple processes that use the library, code portions of the DLL's image can be shared among the processes, thus reducing load time and memory consumption.

Note: The **$IMAGEBASE** directive overrides any value supplied with the **/K** command line compiler directive option.

Include file

See also

Type	Parameter
Syntax	{$I filename} {$INCLUDE filename}
Scope	Local

Remarks

The $I parameter directive instructs the compiler to include the named file in the compilation. In effect, the file is inserted in the compiled text right after the {$I filename} directive. The default extension for filename is .PAS. If filename does not specify a directory path, then, in addition to searching for the file in the same directory as the current module, Delphi searches in the directories specified in the Search path input box on the Directories/Conditionals page of the Project|Options dialog box (or in the directories specified in a /I option on the DCC32 command line).

To specify a file name that includes a space, surround the file name with single quotation marks: `{$I 'My file'}`.

There is one restriction to the use of include files: An include file can't be specified in the middle of a statement part. In fact, all statements between the begin and end of a statement part must exist in the same source file.

Input/output checking

See also

Type	Switch
Syntax	{$I+} or {$I-} {$IOCHECKS ON} or {$IOCHECKS OFF}
Default	{$I+} {$IOCHECKS ON}
Scope	Local

Remarks

The $I switch directive enables or disables the automatic code generation that checks the result of a call to an I/O procedure. I/O procedures are described in the Object Pascal Language Guide. If an I/O procedure returns a nonzero I/O result when this switch is on, an EInOutError exception is raised (or the program is terminated if exception handling is not enabled). When this switch is off, you must check for I/O errors by calling IOResult.

Link object file

See also

Type	Parameter
Syntax	{$L filename} {$LINK filename}
Scope	Local

Remarks

The $L parameter instructs the compiler to link the named file with the program or unit being compiled. The $L directive is used to link with code written in other languages for procedures and functions declared to be external. The named file must be an Intel relocatable object file (.OBJ file). The default extension for filename is .OBJ. If filename does not specify a directory path, then, in addition to searching for the file in the same directory as the current module, Delphi searches in the directories specified in the Search path input box on the Directories/Conditionals page of the Project|Options dialog box (or in the directories specified in a /O option on the DCC32 command line).

To specify a file name that includes a space, surround the file name with single quotation marks: `{$L 'My file'}`.

For further details about linking with assembly language, see online Help.

Local symbol information

See also

Type	Switch
Syntax	{$L+} or {$L-} {$LOCALSYMBOLS ON} or {$LOCALSYMBOLS OFF}
Default	{$L+} {$LOCALSYMBOLS ON}
Scope	Global

Remarks

The $L switch directive enables or disables the generation of local symbol information. Local symbol information consists of the names and types of all local variables and constants in a module, that is, the symbols in the module's implementation part and the symbols within the module's procedures and functions.

For units, the local symbol information is recorded in the unit file along with the unit's object code. Local symbol information increases the size of unit files and takes up additional memory when compiling programs that use the unit, but it does not affect the size or speed of the executable program.

When a program or unit is compiled in the {$L+} state, Delphi's integrated debugger lets you examine and modify the module's local variables. Furthermore, calls to the module's procedures and functions can be examined via the View|Call Stack.

The Include TDW debug info and Map file options on the Linker page of the Project|Options dialog box produce local symbol information for a given module only if that module was compiled in the {$L+} state.

The $L switch is usually used in conjunction with the $D switch, which enables and disables the generation of line-number tables for debugging. The $L directive is ignored if the compiler is in the {$D-} state.

Long strings

See also

Type	Switch
Syntax	{$H+} or {$H-} {$LONGSTRINGS ON} or {$LONGSTRINGS OFF}
Default	{$H+} {$LONGSTRINGS ON}
Scope	Local

Remarks

The **$H** directive controls the meaning of the reserved word **string** when used alone in a type declaration. The generic type **string** can represent either a long, dynamically-allocated string (the fundamental type *AnsiString*) or a short, statically-allocated string (the fundamental type *ShortString*).

By default **{$H+}**, Delphi defines the generic string type to be the long *AnsiString*. All components in the Visual Component Library are compiled in this state. If you write components, they should also use long strings, as should any code that receives data from VCL string-type properties.

The {**$H-**} state is mostly useful for using code from versions of Object Pascal that used short strings by default. You can locally override the meaning of string-type definitions to ensure generation of short strings. You can also change declarations of short string types to **string**[255] or *ShortString*, which are unambiguous and independent of the **$H** setting.

Memory allocation sizes

See also

Type	Parameter
Syntax	{$M minstacksize,maxstacksize} {$MINSTACKSIZE number} {$MAXSTACKSIZE number}
Default	{$M 16384,1048576}
Scope	Global

Remarks

The **$M** directive specifies an application's stack allocation parameters. *minstacksize* must be an integer number between 1024 and 2147483647 that specifies the minimum size of an application's stack, and *maxstacksize* must be an integer number between *minstacksize* and 2147483647 that specifies the maximum size of an application's stack.

If there is not enough memory available to satisfy an application's minimum stack requirement, Windows will report an error upon attempting to start the application.

An application's stack is never allowed to grow larger than the maximum stack size. Any attempt to grow the stack beyond the maximum stack size causes an *EStackOverflow* exception to be raised.

The **$MINSTACKSIZE** and **$MAXSTACKSIZE** directives allow the minimum and maximum stack sizes to be specified separately.

The memory allocation directives are meaningful only in a program. They should not be used in a library or a unit.

Minimum enumeration size

See also

Type	Parameter
Syntax	{$Z1} or {$Z2} or {$Z4} {$MINENUMSIZE 1} or {$MINENUMSIZE 2} or {$MINENUMSIZE 4}
Default	{$Z1} {$MINENUMSIZE 1}
Scope	Local

The **$Z** directive controls the minimum storage size of enumerated types.

An enumerated type is stored as an unsigned byte if the enumeration has no more than 256 values, and if the type was declared in the {**$Z1**} state (the default). If an enumerated type has more than 256 values, or if the type was declared in the {**$Z2**} state, it is stored as an unsigned word. Finally, if an enumerated type is declared in the {**$Z4**} state, it is stored as an unsigned double-word.

The {**$Z2**} and {**$Z4**} states are useful for interfacing with C and C++ libraries, which usually represent enumerated types as words or double-words.

Note: For backwards compatibility with earlier versions of Delphi and Borland Pascal, the directives {**$Z-**} and {**$Z+**} are also supported. They correspond to {**$Z1**} and {**$Z4**}, respectively.

Open String Parameters

See also

Type	Switch
Syntax	{$P+} or {$P-} {$OPENSTRINGS ON} or {$OPENSTRINGS OFF}
Default	{$P+} {$OPENSTRINGS ON}
Scope	Local

Remarks

The **$P** directive is meaningful only for code compiled in the {**$H-**} state, and is provided for backwards compatibility with earlier versions of Delphi and Borland Pascal. **$P** controls the meaning of variable parameters declared using the string keyword in the {**$H-**} state. In the {$P-} state, variable parameters declared using the string keyword are normal variable parameters, but in the {$P+} state, they are open string parameters. Regardless of the setting of the $P directive, the OpenString identifier can always be used to declare open string parameters.

Optimization

See also

Type	Switch
Syntax	{$O+} or {$O-} {$OPTIMIZATION ON} or {$OPTIMIZATION OFF}
Default	{$O+} {$OPTIMIZATION ON}
Scope	Local

The **$O** directive controls code optimization. In the {**$O+**} state, the compiler performs a number of code optimizations, such as placing variables in CPU registers, eliminating common subexpressions, and generating induction variables. In the {**$O-**} state, all such optimizations are disabled.

Other than for certain debugging situations, you should never have a need to turn optimizations off. All optimizations performed by Delphi's Object Pascal compiler are guaranteed not to alter the meaning of a program. In other words, Delphi performs no "unsafe" optimizations that require special awareness by the programmer.

Note: The $O directive can only turn optimization on or off for an entire procedure or function. You can’t turn optimization on or off for a single line or group of lines within a routine.

Overflow checking

See also

Type	Switch
Syntax	{$Q+} or {$Q-} {$OVERFLOWCHECKS ON} or {$OVERFLOWCHECKS OFF}
Default	{$Q-} {$OVERFLOWCHECKS OFF}
Scope	Local

Remarks

The $Q directive controls the generation of overflow checking code. In the {$Q+} state, certain integer arithmetic operations (+, -, *, Abs, Sqr, Succ, Pred, *Inc*, and *Dec*) are checked for overflow. The code for each of these integer arithmetic operations is followed by additional code that verifies that the result is within the supported range. If an overflow check fails, an EIntOverflow exception is raised (or the program is terminated if exception handling is not enabled).

The $Q switch is usually used in conjunction with the $R switch, which enables and disables the generation of range-checking code. Enabling overflow checking slows down your program and makes it somewhat larger, so use {$Q+} only for debugging.

Pentium-safe FDIV operations

See also

Type	Switch
Syntax	{$U+} or {$U-} {$SAFEDIVIDE ON} or {$SAFEDIVIDE OFF}
Default	{$U-}
Scope	Local

The **$U** directive controls generation of floating-point code that guards against the flawed FDIV instruction exhibited by certain early Pentium processors. Windows 95, Windows NT 3.51, and later contain code which corrects the Pentium FDIV bug system-wide.

In the {**$U+**} state, all floating-point divisions are performed using a runtime library routine. The first time the floating-point division routine is invoked, it checks whether the processor's FDIV instruction works correctly, and updates the *TestFDIV* variable (declared in the *System* unit) accordingly. For subsequent floating-point divide operations, the value stored in *TestFDIV* is used to determine what action to take.

Value	Meaning
-1	FDIV instruction has been tested and found to be flawed.
0	FDIV instruction has not yet been tested.
1	FDIV instruction has been tested and found to be correct.

For processors that do not exhibit the FDIV flaw, {**$U+**} results in only a slight performance degradation. For a flawed Pentium processor, floating-point divide operations may take up to three times longer in the {**$U+**} state, but they will always produce correct results.

In the {**$U-**} state, floating-point divide operations are performed using in-line FDIV instructions. This results in optimum speed and code size, but may produce incorrect results on flawed Pentium processors. You should use the {**$U-**} state only in cases where you are certain that the code is not running on a flawed Pentium processor.

Private symbol

See also

Type	Parameter
Syntax	{$NODEFINE identifier}

The NODEFINE directive prevents the specified symbol from being included in the header file generated for C++Builder, while allowing some information to be output to the OBJ file. When you use NODEFINE, it is your responsibility to define any necessary types with HPPEMIT. For example:

```
type
  Temperature = type double;
  {*$NODEFINE Temperature]
  {$HPPEMIT 'typedef double Temperature'}
```

Private unit

See also

Type	Parameter
Syntax	{$NOINCLUDE filename}

The NOINCLUDE directive prevents the specified file from being included in header files generated for C++Builder. For example, `{$NOINCLUDE Unit1}` removes `#include Unit1`.

Range checking

See also

Type	Switch
Syntax	{$R+} or {$R-} {$RANGECHECKS ON} or {$RANGECHECKS OFF}
Default	{$R-} {$RANGECHECKS OFF}
Scope	Local

Remarks

The $R directive enables or disables the generation of range-checking code. In the {$R+} state, all array and string-indexing expressions are verified as being within the defined bounds, and all assignments to scalar and subrange variables are checked to be within range. If a range check fails, an ERangeError exception is raised (or the program is terminated if exception handling is not enabled).

Enabling range checking slows down your program and makes it somewhat larger, so use the {$R+} only for debugging.

Note: Long strings are not range checked.

Real48 compatibility

See also

Type	Switch
Syntax	{$REALCOMPATIBILITY ON} or {$REALCOMPATIBILITY OFF}
Default	$REALCOMPATIBILITY OFF}
Scope	Local

Remarks

In the default {**$REALCOMPATIBILITY OFF**} state, the generic Real type is equivalent to Double.

In the {**$REALCOMPATIBILITY ON**} state, Real is equivalent to Real48.

The **REALCOMPATIBILITY** switch provides backward compatibility for legacy code in which Real is used to represent the 6-byte real type now called Real48. In new code, use Real48 when you want to specify a 6-byte real.

Double is the preferred real type for most purposes.

Resource file

See also

Type	Parameter
Syntax	{$R filename} {$RESOURCE filename} {$R *.xxx} {$R filename.RES filename.RC}
Scope	Local

Remarks

The $R directive specifies the name of a resource file to be included in an application or library. The named file must be a Windows resource file and the default extension for filenames is .RES. To specify a file name that includes a space, surround the file name with single quotation marks: `{$R 'My file'}`.

The ***** symbol has a special meaning in $R directives: it stands for the base name (without extension) of the source-code file where the directive occurs. Usually, an application's resource (.RES) file has the same name as its project (.DPR) file; in this case, including {$R *.RES} in the project file links the corresponding resource file to the application. Similarly, a form (.DFM) file usually has the same name as its unit (.PAS) file; including {$R *.DFM} in the .PAS file links the corresponding form file to the application.

{$R filename.RES filename.RC} (where the two occurrences of 'filename' match) makes the .RC file appear in Delphi's Project Manager. When the user opens the .RC file from the Project Manager, the String Table editor is invoked.

When a {$R filename} directive is used in a unit, the specified file name is simply recorded in the resulting unit file. No checks are made at that point to ensure that the filename is correct and that it specifies an existing file.

When an application or library is linked (after compiling the program or library source file), the resource files specified in all used units as well as in the program or library itself are processed, and each resource in each resource file is copied to the executable being produced. During the resource processing phase, Delphi's linker searches for .RES files in the same directory as the module containing the $R directive, and in the directories specified in the Search path input box on the Directories/Conditionals page of the Project|Options dialog box (or in the directories specified in a /R option on the DCC32 command line).

RUNONLY directive

See also

Type	Switch
Syntax	{$RUNONLY ON} or {$RUNONLY OFF}
Default	{$RUNONLY OFF}
Scope	Local

Remarks

The **{$RUNONLY ON}** directive causes the package where it occurs to be compiled as runtime only. Packages compiled with **{$RUNONLY ON}** cannot be installed as design-time packages in the Delphi IDE.

Place the **RUNONLY** directive only in .DPK files. For more information, see the *Object Pascal Language Guide*.

Runtime type information

See also

Type	Switch
Syntax	{$M+} or {$M-} {$TYPEINFO ON} or {$TYPEINFO OFF}
Default	{$M-} {$TYPEINFO OFF}
Scope	Local

The $M switch directive controls generation of runtime type information (RTTI). When a class is declared in the {$M+} state, or is derived from a class that was declared in the {$M+} state, the compiler generates runtime type information for fields, methods, and properties that are declared in a published section. If a class is declared in the {$M-} state, and is not derived from a class that was declared in the {$M+} state, published sections are not allowed in the class.

Note: The TPersistent class defined in the Classes unit of the VCL is declared in the {$M+} state, so any class derived from TPersistent will have RTTI generated for its published sections. The VCL uses the runtime type information generated for published sections to access the values of a component's properties when saving or loading form files. Furthermore, the Delphi IDE uses a component's runtime type information to determine the list of properties to show in the Object Inspector.

There is seldom, if ever, any need for an application to directly use the $M compiler switch.

Symbol declaration and cross-reference information

See also

Type	Switch
Syntax	{$Y+}, {$Y-}, or {$YD}; {$REFERENCEINFO ON}, {DEFINITIONINFO OFF} or {$REFERENCEINFO OFF}, or {DEFINITIONINFO ON}
Default	{$YD} {$DEFINITIONINFO ON}
Scope	Global

Remarks

The $Y directive controls generation of symbol reference information used by Delphi's Project Browser, Code Explorer, and Code editor. This information consists of tables that provide the source-code line numbers for all declarations of and (in the {$Y+} state) references to identifiers in a module. For units, the information is recorded in the .DCU file along with the unit's object code. Symbol reference information increases the size of .DCU files, but it does not affect the size or speed of the executable program.

When a program or unit is compiled in the default {$YD} (or {DEFINITIONINFO ON}) state, the compiler records information about where each identifier is defined. For most identifiers—variables, constants, classes, and so forth—the compiler records the location of the declaration. For procedures, functions, and methods, the compiler records the location of the implementation. This enables Code editor browsing.

When a program or unit is compiled in the {$Y+} (or {REFERENCEINFO ON}) state, the compiler records information about where every identifier is used as well as where it is defined. This enables the References page of the Project Browser.

When a program or unit is compiled in the {$Y-} (or {DEFINITIONINFO OFF} or {REFERENCEINFO OFF}) state, no symbol reference information is recorded. This disables Code editor browsing and the References page of the Project Browser.

The $Y switch is usually used in conjunction with the $D and $L switches, which control generation of debug information and local symbol information. The $Y directive has no effect unless both $D and $L are enabled.

Note: Generating full cross-reference information ({$Y+}) can slow the compile/link cycle, so you should not use this except when you need the Project Browser References page.

Type-checked pointers

See also

Type	Switch
Syntax	{$T+} or {$T-} {$TYPEDADDRESS ON} or {$TYPEDADDRESS OFF}
Default	{$T-} {$TYPEDADDRESS OFF}
Scope	Global

Remarks

The $T directive controls the types of pointer values generated by the @ operator and the compatibility of pointer types.

In the {$T-} state, the result of the @ operator is always an untyped pointer (Pointer) that is compatible with all other pointer types. When @ is applied to a variable reference in the {$T+} state, the result is a typed pointer that is compatible only with Pointer and with other pointers to the type of the variable.

In the {$T-} state, distinct pointer types other than Pointer are incompatible (even if they are pointers to the same type). In the {$T+} state, pointers to the same type are compatible.

UNDEF directive

See also

Type	Conditional compilation
Syntax	{$UNDEF name}

Remarks

Undefines a previously defined conditional symbol. The symbol is forgotten for the remainder of the compilation of the current source file or until it reappears in a {$DEFINE name} directive. The {$UNDEF name} directive has no effect if name is already undefined.

Conditional symbols defined with command-line switch or IDE Project|Options are reinstated at the start of compilation of each unit source file. Conversely, conditional symbols defined in a unit source file are forgotten when the compiler starts on another unit.

Var-string checking

See also

Type	Switch
Syntax	{$V+} or {$V-} {$VARSTRINGCHECKS ON} or {$VARSTRINGCHECKS OFF}
Default	{$V+} {$VARSTRINGCHECKS ON}
Scope	Local

Remarks

The **$V** directive is meaningful only for code that uses short strings (see Long strings compiler directive), and is provided for backwards compatibility with earlier versions of Delphi and Borland Pascal.

The $V directive controls type checking on short strings passed as variable parameters. In the {$V+} state, strict type checking is performed, requiring the formal and actual parameters to be of identical string types. In the {$V-} (relaxed) state, any short string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter.

Warnings

See also

Type	Switch
Syntax	{$WARNINGS ON} or {$WARNINGS OFF}
Default	{$WARNINGS ON}
Scope	Local

Remarks

The **$WARNINGS** directive controls the generation of compiler warnings.

In the {**$WARNINGS ON**} state, the compiler issues warning messages when detecting uninitialized variables, missing function results, construction of abstract objects, and so on. In the {**$WARNINGS OFF**} state, the compiler generates no warning messages.

By placing code between {**$WARNINGS OFF**} and {**$WARNINGS ON**} directives, you can selectively turns off warnings that you don't care about.

Weak packaging

See also

Type	Switch
Syntax	{$WEAKPACKAGEUNIT ON} or {$WEAKPACKAGEUNIT OFF}
Default	{$WEAKPACKAGEUNIT OFF}
Scope	Local

Remarks

The **$WEAKPACKAGEUNIT** directive affects the way a .DCU file is stored in a package's .DCP and .BPL files. If **{$WEAKPACKAGEUNIT ON}** appears in a unit file, the compiler omits the unit from BPLs when possible, and creates a non-packaged local copy of the unit when it is required by another application or package. A unit compiled with this directive is said to be "weakly packaged."

For example, suppose a package called PACK contains only one unit, UNIT1. Suppose UNIT1 does not use any further units, but it makes calls to RARE.DLL. If the **{$WEAKPACKAGEUNIT ON}** directive is inserted in UNIT1.PAS before compiling, UNIT1 will not be included in PACK.BPL; copies of RARE.DLL will not have to be distributed with PACK. However, UNIT1 will still be included in PACK.DCP. If UNIT1 is referenced by another package or application that uses PACK, it will be copied from PACK.DCP and compiled directly into the project.

Now suppose a second unit, UNIT2, is added to PACK. Suppose that UNIT2 uses UNIT1. This time, even if PACK is compiled with **{$WEAKPACKAGEUNIT ON}** in UNIT1.PAS, the compiler will include UNIT1 in PACK.BPL. But other packages or applications that reference UNIT1 will use the (non-packaged) copy taken from PACK.DCP.

Note: Unit files containing the **{$WEAKPACKAGEUNIT ON}** directive must not have global variables, initialization sections, or finalization sections.

The **$WEAKPACKAGEUNIT** directive is an advanced feature intended for developers who distribute their BPLs to other Delphi programmers. It can help to avoid distribution of infrequently used DLLs, and to eliminate conflicts among packages that may depend on the same external library.

For example, Delphi's PenWin unit references PENWIN.DLL. Most projects don't use PenWin, and most computers don't have PENWIN.DLL installed on them. For this reason, the PenWin unit is weakly packaged in VCL50 (which encapsulates many commonly used Delphi components). When you compile a project that uses PenWin and the VCL50 package, PenWin is copied from VCL50.DCP and bound directly into your project; the resulting executable is statically linked to PENWIN.DLL.

If PenWin were not weakly packaged, two problems would arise. First, VCL50 itself would be statically linked to PENWIN.DLL, and so could not be loaded on any computer which didn't have PENWIN.DLL installed. Second, if someone tried to create a package that contained PenWin, a compiler error would result because the PenWin unit would be contained in both VCL50 and the new package. Thus, without weak packaging, PenWin could not be included in standard distributions of VCL50.

Windows stack frames

See also

Type	Switch
Syntax	{$W+} or {$W-} {$STACKFRAMES ON} or {$STACKFRAMES OFF}
Default	{$W-} {$STACKFRAMES OFF}
Scope	Local

Remarks

The **$W** directive controls the generation of stack frames for procedures and functions. In the {**$W+**} state, stack frames are always generated for procedures and function, even when they're not needed. In the {**$W-**} state, stack frames are only generated when they're required, as determined by the routine's use of local variables.

Some debugging tools require stack frames to be generated for all procedures and functions, but other than that you should never have a need to use the {**$W+**} state.

Writeable typed constants

See also

Type	Switch
Syntax	{$J+} or {$J-} {$WRITEABLECONST ON} or {$WRITEABLECONST OFF}
Default	{$J+} {$WRITEABLECONST ON}
Scope	Local

The **$J** directive controls whether typed constants can be modified or not. In the {**$J+**} state, typed constants can be modified, and are in essence initialized variables. In the {**$J-**} state, typed constants are truly constant, and any attempt to modify a typed constant causes the compiler to report an error.

In previous versions of Delphi and Borland Pascal, typed constants were always writeable, corresponding to the {**$J+**} state. Old source code that uses writeable typed constants must be compiled in the {**$J+**} state, but for new applications it is recommended that you use initialized variables and compile your code in the {**$J-**} state.

Using conditional compilation directives

See also

Two basic conditional compilation constructs closely resemble Pascal's if statement. The first construct

```
{$IFxxx}
  ...
{$ENDIF}
```

causes the source text between {$IFxxx} and {$ENDIF} to be compiled only if the condition specified in {$IFxxx} is True. If the condition is False, the source text between the two directives is ignored.

The second conditional compilation construct:

```
{$IFxxx}
  ...
{$ELSE}
  ...
{$ENDIF}
```

causes either the source text between {$IFxxx} and {$ELSE} or the source text between {$ELSE} and {$ENDIF} to be compiled, depending on the condition specified by the {$IFxxx}.

Here are some examples of conditional compilation constructs:

```
{$IFDEF Debug}
 Writeln('X = ', X);
{$ENDIF}
{$IFDEF WIN32}
 P := SmallPointToPoint(Message.Pos);
{$ELSE}
 P := Message.Pos;
{$ENDIF}
```

You can nest conditional compilation constructs up to 16 levels deep. For every {$IFxxx}, the corresponding {$ENDIF} must be found within the same source file, which means there must be an equal number of {$IFxxx}'s and {$ENDIF}'s in every source file.

Conditional symbols

See also

Conditional compilation is based on the evaluation of conditional symbols. Conditional symbols are defined and undefined using the directives

```
{$DEFINE name}
{$UNDEF name}
```

You can also use the /D switch in the command-line compiler to define a symbol (or add the symbol to the Conditional Defines input box on the Directories/Conditionals page of the Project|Options dialog box in the IDE).

Conditional symbols are best compared to Boolean variables: They are either True (defined) or False (undefined). The {$DEFINE} directive sets a given symbol to True, and the {$UNDEF} directive sets it to False.

Conditional symbols follow the same rules as Pascal identifiers: They must start with a letter, followed by any combination of letters, digits, and underscores. They can be of any length, but only the first 255 characters are significant.

Conditional symbols and Pascal identifiers have no correlation whatsoever. Conditional symbols cannot be referenced in the actual program and the program's identifiers cannot be referenced in conditional directives. For example, the construct

```
const
 Debug = True;
begin
 {$IFDEF Debug}
 Writeln('Debug is on');
 {$ENDIF}
end;
```

will not compile the Writeln statement. Likewise, the construct

```
{$DEFINE Debug}
begin
 if Debug then
   Writeln('Debug is on');
end;
```

will result in an unknown identifier error in the if statement.

Delphi defines the following standard conditional symbols

VER130	Always defined, indicating that this is version 13.0 of the Object Pascal compiler. Each version has corresponding predefined symbols; for example, version 10.0 has VER100 defined, version 12.5 would have VER125 defined, and so on.
WIN32	Indicates that the operating environment is the Win32 API.
CPU386	Indicates that the CPU is an Intel 386 or better.
CONSOLE	Defined if an application is being compiled as a console application.

Other conditional symbols can be defined before a compilation by using the Conditional Defines input box, or the /D command-line option if you are using the command-line compiler.

Note: Source code conditional defines are evaluated only when the source code is recompiled. As mentioned earlier, changing a conditional define and recompiling the project (make), will not cause units to be recompiled from source, and therefore the units will not reflect the change in conditional defines. "Build All" is the only way to ensure everything in a project reflects changes to conditional defines.

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.

The Databases 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 Databases.
2. Do one of the following,

- Choose Object|Open.
- From the Database Explorer context menu, choose Open.

When you are connected to a database, the icon in the left pane is surrounded by a green box.

To expand a database,

- In the Databases 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 the Run button.

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

For more information, open the Help menu within the Database Explorer and choose a command from it.

Database Editor dialog box

The Database Editor dialog box sets up the properties of a database that specify the connection that should be made to a database. This dialog box allows you to specify the type of database, the connection parameters, what should happen when the user activates a connection, and whether the database is persistent.

These properties of the database component, as well as others, can also be specified using the Object inspector.

To display the Database Editor dialog box, double click on a database component.

Dialog box options

Name
Specifies the name of the database. This name refers to the database component from within the code of your application.

Alias
Specifies the BDE alias for the database. Choose a database aliases from the drop-down list. This list contains all aliases currently registered with the BDE. If you do not want to connect to a database that is registered as a BDE alias, you can set the Driver property instead. If you set the Alias property, the Driver property is cleared, as the driver type is implicit in the BDE alias.

Driver
Specifies the type of database represented by the database component. Choose a driver type such as STANDARD, ORACLE, SYBASE, or INTERBASE from the drop-down list. If the database server has an alias registered with the BDE, you can set the Alias instead. Setting the Driver automatically clears the Alias property, to avoid potential conflicts with the driver type implicit in the database alias.

Parameter overrides
Specifies the values of all login parameters when connecting to the database. The specific parameters depend on the type of database. To obtain a list of all parameters, as well as their default values, click the Defaults button. You can then modify the default values to the values you want to use.

Defaults
Press the Defaults button to set the Parameter overrides to the default values for the driver type.

Clear
Press the Clear button to remove all parameter overrides.

Login Prompt
Check the Login Prompt control to cause a login dialog to appear automatically when the user connects to the database. Uncheck the Login Prompt control to prevent the automatic login dialog. Most database servers (except for the file-based STANDARD types) require the user to supply a password when connecting to the database. For such servers, if the automatic login prompt is omitted, the application must supply the user name and password in some other manner. These can be supplied either by providing hard-coded parameter overrides, or by supplying an OnLogin event handler that sets the values for these parameters.

Keep inactive connection
Check Keep inactive connection to indicate that the application should remain connected to the database even if no datasets are currently open. For connections to remote database servers, or for applications that frequently open and close datasets, checking Keep inactive connection reduces network traffic, speeds up applications, and avoids logging in to the server each time the connection is reestablished. Uncheck Keep inactive connection to cause the database connection to be dropped when there are no open datasets. Dropping a connection releases system resources allocated to the connection, but if a dataset is later opened that uses the database, the connection must be

reestablished and initialized.

<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 attempting 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 Procedures and functions.

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 Class 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 protected 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 Unit syntax.

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 you 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 the declaration manually. For more information about the USES clause syntax, see the reserved word USES.

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.

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;
```

Manipulating files

Several common file operations are built into Object Pascal's runtime library. The procedures and functions for working with files operate at a high level: 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 FileGetAttrfunction. 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 logical operators to manipulate the individual attributes. Each attribute has a mnemonic name defined in the SysUtils unit.

For example, to set a file's read-only attribute, you would do the following:

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

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

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

Setting file attributes

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 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(FileDateToDateTime(FileAge(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;
```

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.

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.

Example

The following example declares the function GetPassword as an interface routine. The exports section includes the GetPassword routine name to ensure that the function is successfully exported.

```
unit PassForm;
interface
uses
  SysUtils, Windows, 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

Creating packages and 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.
```

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.

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 runtime, 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.

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 and choose the Add To Repository command.

   The Add To Repository dialog box appears.
2. In the Title edit box, specify a name for the object.
3. In the Description edit box, type a brief description of this object.
4. Choose the Page on which the form should appear in the New Items dialog box.
5. You can specify an Author of the form, which shows only in the detailed view of the Object Repository.
6. To specify an icon for the object, choose the Browse button.

   The Select Bitmap dialog box appears.
7. Locate and select the bitmap (if any) you want to use, and choose OK to exit the Select Bitmap dialog box.
8. 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 (Forms)

You use the Add To Repository dialog box to add new forms to the Object Repository.

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 displayed when you select the View Details option from the context menu in the New Items dialog 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 Repository. 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.

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.

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;
```

Adding a speed button to a toolbar

See also

Speed buttons are graphical buttons on toolbars 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 toolbar,

- Place the speed button component on the panel that represents the toolbar.

The default height of the toolbar 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.

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 runtime. 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 runtime.

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 toolbars

See also

Toolbars do not have to be visible all the time. In fact, it is often convenient to be able to have a number of toolbars available, but show them only when the user wants to use them. Often you create a form that has several toolbars, but hide some or all of them.

To create a hidden toolbar,

1. Add a toolbar to the form. (Be sure to set Align to alTop.)
2. Set the panel's Visible property to False.

Although the toolbar remains visible at design time so you can modify it, it remains hidden at runtime until the application specifically makes it visible.

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

Status Bars
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 you 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
OnHint event

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 runtime

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 runtime.

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 runtime. 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 runtime. See Providing an Area for Drawing at Runtime

See also

Working with graphics
Adding an Image control
Loading an image

Providing an area for drawing at runtime

See also Example

You can provide an area for the user to draw in at runtime 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

Adding an image control

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.

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

Adding an image control

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 property.

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

Adding an image control

Providing an area for drawing at runtime

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;
```

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. 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 and choose Add Fields.

Add All Fields

Choose Add All Fields in the Fields editor to create persistent fields for every field in the underlying dataset.

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 and 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 and 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 context menu.

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 context menu 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 Windows 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

The Columns editor contains a list box of defined columns, insertion and deletion buttons. Column properties are displayed in the Object Inspector.

At design time, use the DBGrid Columns editor to create a set of personal column objects for the grid. At runtime, 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 DBGrid component in the form.
2. Right-click and choose Columns editor.

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.

Add button

To create a persistent column for a grid, click on the Add button in the Columns editor. A new column will entered the list box. Default parameters for the column will be displayed in the Object Repository.

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 properties displayed on the Object Inspector. The following table summarizes the properties

Property	Purpose
Alignment	Left justifies, right justifies, or centers the field data in the column. Default source: TField.Alignment
ButtonStyle	cbsAuto: (default) Displays a drop-down list if the associated field is a lookup field, or if the column's PickList property contains data. cbsEllipsis: Displays an ellipsis (...) button to the right of the cell. Clicking on the button fires the grid's OnEditButtonClick event. cbsNone: The column uses only the normal edit control to edit data in the column
Color	Specifies the background color of the cells of the column. 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.
Font	Invoke the Font editor to define the font attibutes of the column text.
PickList	Invoke the String List editor to define 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.
PopupMenu	The PopupMenu property identifies the pop-up menu associated with the control. Assign a value to PopupMenu to make a pop-up menu appear when the user selects the control and clicks the right mouse button. If the TPopupMenu's AutoPopup property is True, the pop-up menu appears automatically. If the menu's AutoPopup property is False, display the menu with a call to its Popup method
ReadOnly	True: The data in the column cannot be edited by the user. False: (default) The data in the column can be edited.
Title	Click Title to expand the property list. Define Allignment, Caption text, Color, Font and Width for the selected column.
Width	Specifies the width of the column in screen pixels. Default Source: derived from TField.DisplayWidth

Restore Defaults

You can discard all property changes in the selected column by right clicking the Restore Defaults button. The column's properties will return to their default values.

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 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 runtime by a dataset's OnCalcFields event handler.
- **Lookup fields**, which retrieve values from a specified dataset at runtime based on search criteria you specify.

If the dataset is a client data set, a fourth field type is also available:

- **InternalCalc fields**, which retrieve calculated values that are stored with the dataset (instead of being dynamically calculated in an OnCalcFields event handler).
- **Aggregate fields**, which retrieve values that summarize the data over several records in a client dataset.

These types of persistent fields are only for display purposes. The data they contain at runtime 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 TQuery object to display the Speed menu, select Fields Editor.
2. Right-click the Fields editor list and choose New field.

   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 properties group

The Field properties group box enables you to enter general field component information.

Name

Enter the component's field name. 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.

Size

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.

Field type

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.

Lookup definition

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.

Note: 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. Remove the field from the list of persistent fields assigned for the dataset, and then choose New Field from the context menu
2. Enter the name of an existing persistent field in the Name edit box. Do not enter a new field name.
3. Choose a new data type for the field from the Type combo box. The data type you choose should be different from the data type of the field you are replacing.
4. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type **TStringField**, **TBytesField**, and **TVarBytesField**.
5. Select Data in the Field type radio group if it is not already selected.
6. Choose OK. The New Field dialog box closes, the newly defined data field replaces the existing field you specified in Step 1, and the component declaration in the data module or form's **type** declaration is updated.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector.

Defining a calculated field

A calculated field displays values calculated at runtime by a dataset's **OnCalcFields** event handler. For example, you might create a string field that displays concatenated values from other fields.

To create a calculated field in the New Field dialog box:

1. Enter a name for the calculated field in the Name edit box. Do not enter the name of an existing field.
2. Choose a data type for the field from the Type combo box.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type **TStringField**, **TBytesField**, and **TVarBytesField**.
4. Select Calculated 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 runtime 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 runtime. Specifying a lookup dataset enables the Lookup Keys and Result Field combo boxes.
6. Choose from the Key Fields drop-down list a field in the current dataset for which to match values. To match more than one field, enter field names directly instead of choosing from the drop-down list. Separate multiple field names with semicolons.
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.

See Also

Defining a data field

Defining a calculated field

Programming a calculated field

Defining a lookup field

Define Parameter dialog box

The Define Parameter dialog box allows you to create new parameters for a stored procedure when the server does not pass the information to Delphi. Do not add parameters for servers that pass parameter information to Delphi unless you are working with Oracle overloaded stored procedures.

Bring up the Define Parameter dialog box by pressing the Add button in the StoredProc Parameters Editor. After the parameter has been created using the Define Parameter dialog box, you can specify the parameter type (input, output, or results) and give the parameter a default value in the StoredProc Parameters Editor.

Using the Define Parameter dialog box

Parameter Name edit control

Use the Parameter name edit control to provide the name of the parameter.

Data type combo box

Use the Data type combo box to specify the field type for the parameter. The drop-down list provides you with selections for the field types supported by the Delphi field component. Choose the type expected for the parameter by your server.

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.

Component menu

The options on the Component menu are:

New Component	Opens the Component Expert
Install Component	Install a component into an existing or new package
Import ActiveX Control	Add type libraries of ActiveX controls to your Delphi project.
Create Component Template	Customize components and save them as a template with a new name, palette page, and icon.
Install Packages	Specify packages required by your project.
Configure Palette	Opens the Palette dialog box

Component|New Component

See Also

Use the New Component dialog to create the basic unit for a new component.

Ancestor Type	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. After you enter a base class, default entries are written to the Class Name and Unit File Name. You can accept or edit these entries.
Class Name	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.
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 on the Component palette.
Unit File Name	The name of the unit that will contain the new component. You can include a directory path with the name; otherwise, the unit will be created in the current directory. If the unit directory is not in the Search Path, it will be added at the end.
Search Path	The path Delphi will use to search for a file.
Install	Install component to a new or existing package. After identifying a package, the Package Editor dialog box will be displayed. This button is not available when the New Component dialog is invoked from the Package editor.
Create Unit	Display the component code in the Code editor. The unit will not be included in the current project unless explicitly added. This button is not available when the New Component dialog is invoked from the Package editor.

Component|Install Component

See Also

Select Component|Install Component to install a component into an existing or new package.

After entering the information required, press OK. The Package editor dialog box is displayed.

Into existing package

In this case, you write the component code, identified by the unit, name to an existing package file.

Into new package

In this case, you declare a new package and write the component code, identified by unit name, to it.

Into new package

Unit file name Enter the name of the unit you want to install. If the unit is in the Search Path, a full path name is not required. If the unit directory is not in the Search Path, it will be added to the end.

Search path The path used by Delphi to search for files.

Package file name Enter the name of the package to create. You can include a directory path with the name; otherwise, the package will reside in the current directory. To open a file/directory selection dialog box, click Browse. If you type a file name directly in the New Package dialog box, the .DPK extension will be added automatically.

Package names must be unique within a project. If you name a package “STATS”, the Package editor generates a source file for it called “STATS.DPK”; the compiler generates an executable and a binary image called “STATS.BPL” and “STATS.DCP”, respectively. Use “STATS” to refer to the package in the requires clause of another package, or when using the package in an application.

Package description A brief description of the package.

Into existing package

Unit file name	Enter the name of the unit you want to install. If the unit is in the Search Path, a full path name is not required. If the unit directory is not in the Search Path, it will be added to the end.
Search path	The path used by Delphi to search for files.
Package file name	Use the drop-down list to select the name of an installed package, or enter the name of another existing package.
Package description	A brief description of the selected package.

Component|Import ActiveX Control

See also

The Import ActiveX Control dialog displays the ActiveX controls registered on your system so you can add them to your Delphi projects. You can generate Pascal declarations in a .PAS file that let you use any of these controls as though it were a native VCL object. In effect, the ActiveX control is placed within a Delphi wrapper.

The top part of the dialog is a list of Controls that are currently registered and thus available to be imported into Delphi. This lists lets you extract the Pascal declarations from an existing control. You can also conveniently register a new control from this dialog box so that it is available to be imported.

To add and register a new ActiveX control:

1. Click Add. The Register OLE Control dialog box appears.
2. In the Register OLE Control dialog box, navigate to the disk or network location of the control file you want to add.
3. Select the new ActiveX control. It is automatically registered on your system for Delphi and immediately appears in the list of available controls in the Import ActiveX Control dialog.

Add button	The Add button lets you locate a new ActiveX control and register it in the Windows Registry, so that it will appear in the list of registered objects available to be imported into Delphi.
Remove button	To remove a registered ActiveX control, click the Remove button. The control is removed from the Windows Registry and from this list.
Class names	Shows only the ActiveX control classes in the selected library.
Palette page	Shows the Component palette location of objects associated with the selected library. Allows you to group controls by function or vendor, for example.
Unit dir name	Shows the name of the directory that will contain the unit using this control. Only the path root is shown; no file name appears. The unit name is derived from the internal type library name. Click the Browse button to move up the directory tree. If the Unit Directory name is not in the Search Path, it will be added to the end.
Search path	Shows the path Delphi will use to search for a file.
Install…	Creates a unit, opens the Package editor, and installs the unit in the package you specify. When you click Install, the Package editor appears, asking if you want to install the new unit in the default Delphi package or create another package to contain it. This button is not available when the dialog is invoked from the Package editor.
Create unit	Creates a unit and displays the unit code in the Code editor. (Does not include the unit in the current project.) This button is not available when the dialog is invoked from the Package editor.

Component|Create Component Template

See also

To create a component template,

1. Place and arrange components on a form. In the Object Inspector, set their properties and events as desired.
2 Select the components. The easiest way to select several components is to drag the mouse over all of them. Gray handles appear at the corners of each selected component.
3 Choose Component|Create Component Template.
4 Specify a name, palette page, and bitmap for the template:

Component Name	This field shows “Template” appended to the name of the first component you selected. You can change this to any valid name, but be careful not to duplicate existing component names.
Palette Page	Select the Component palette page on which you want the new template to appear.
Palette Icon	This field shows the icon of the first component you selected. To change it, click the Change button and choose a new image for the icon. The bitmap you choose must be no larger than 24 pixels by 24 pixels.

5 Click OK. Your new template appears immediately in the palette page you indicated with the new icon.

Context menus for specific components

When you right-click on a component in the Form Designer or Data Module Designer, you see a pop-up menu with shortcuts to frequently used design functions. These menus vary from component to component, but they all include the basic Form context menu options. The following additional options are available from the menus for specific components.

Action editor (TWebDispatcher, TWebModule, TMidasPageProducer)
Action List editor (TActionList)
ActiveX Control Data Bindings editor (TActiveXControl)
Add to Palette (TFrame)
Assign Local Data (TClientDataSet)
Bands editor (TCoolBar)
Clear Data(TClientDataSet)
Columns editor (TDBGrid, TListView)
Copy Object (TOleContainer)
Create DataSet (TClientDataSet)
Create Table (TTable)
Database editor (TDataBase)
Decision Cube editor (TDecisionCube)
Decision Query editor (TDecisionQuery)
Delete Object (TOleContainer)
Delete Table (TTable)
Display Sparse Rows/Columns (TDecisionSource)
Execute (TBatchMove)
Explore (TDatabase, TTable, TQuery, TStoredProc)
Fetch Params(TClientDataSet)
Fields editor (TClientDataSet, TTable, TQuery, TStoredProc)
ImageList editor (TImageList)
Input Mask editor (TMaskEdit)
Insert Object (TOleContainer)
Items editor (TListView)
Items editor (TTreeView)
Load from File (TClientDataSet)
Masked Text editor (TMaskEdit)
Menu Designer (TMainMenu, TPopupMenu)
New Button (TToolBar)
New Page (TPageControl, TTabSheet)
New Separator (TToolBar)
Next Frame (TAnimate)
Next Page (TPageControl, TTabSheet)
Object Properties (TOleContainer)
Panels editor (TStatusBar)
Paste Special (TOleContainer)
Previous Frame (TAnimate)
Previous Page (TPageControl, TTabSheet)
Query Builder (TQuery)

Rename Table (TTable)
Response editor (TDataSetTableProducer, TQueryTableProducer)
Save to File (TClientDataSet)
Sections editor (THeaderControl)
Subtotals on/off (TDecisionGrid)
UpdateSQL editor (TUpdateSQL)
Update Table Definition (TTable; calls TFieldDefs.Update and TIndexDefs.Update)

Your installation of Delphi may have third-party components with additional menu options.

Form context menu

See also

Use the Form context menu to manipulate components in the Form Designer or Data Module Designer.

To display the Form context menu:

- Right-click anywhere in the client area of a form, frame, or data module, or select the form and press *Alt+F10*.
- Right-click on a specific component, or select a component and press *Alt+F10*.
- Select several components (by dragging the mouse across them or holding down the *Shift* key while clicking each one), then right-click or press *Alt+F10*.

The following commands appear on some context menus in either the Form Designer or Data Module Designer:

Align To Grid
Bring To Front
Send To Back
Revert To Inherited
Align
Size
Scale
Tab Order
Creation Order
Add To Repository
View As Text
Show Field Info
Auto Height
Color
Remove From Diagram
Remove Relationship
Fill Color
Starts With
Ends With
Print

Additional commands appear on the context menus for specific components.

Query Builder **(Form context menu)**

Choose Query Builder from the Form context menu 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.

Execute (Form context menu)

See also

Choose Execute from the Form context menu 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.
- Append and Update data from a dataset.
- Delete records in a dataset from a table.

To run this process at runtime, you must call the Execute method for BatchMove.

Next Page **(Form context menu)**

See also

Choose Next Page from the Form context menu to change the ActivePage property of a TPageControl to the next TTabSheet. Before you can change the active page, you must add the pages to the page control by choosing New Page from the Form context menu.

Previous Page **(Form context menu)**

See also

Choose Previous Page from the Form context menu to change the ActivePage property of a TPageControl to the previous TTabSheet. Before you can change the active page, you must add the pages to the page control by choosing New Page from the Form context menu.

Align To Grid **(Form context menu)**

Choose Align To Grid from the Form context menu to align the selected components to the closest grid point.

You can specify the size of the grid on the Preferences page of the Tools|Environment Options dialog box.

This command works the same as Edit|Align To Grid.

Bring To Front **(Form context menu)**

Choose Bring To Front from the Form context menu to move a selected component in front of all other components on the form.

This is called changing the component's z-order

This command works the same as Edit|Bring To Front.

Send To Back **(Form context menu)**

Choose Send To Back from the Form context menu to move a selected component behind all other components on the form.

This is called changing the component's z-order.

This command works the same as Edit|Send To Back.

Revert To Inherited **(Form or Object Inspector context menu)**

If a form inherits design features and properties from another form, you can choose Revert To Inherited from the Form context menu to restore the form to its original state. For example, if a form inherits a certain button placement from another form and you then move the button, Revert To Inherited returns the button to its original position.

Align **(Form context menu)**

Choose Align from the Form context menu to open the Alignment dialog box.

Use this dialog box to line up selected components in relation to each other or to the form:

This command works the same as Edit|Align.

Size (Form context menu)

Choose Size from the Form context menu to open the Size dialog box.

Use this dialog box to resize multiple components to be exactly the same height or width.

This command works the same as Edit|Size.

Scale **(Form context menu)**

Choose Scale from the Form context menu to open the Scale dialog box.

Use this dialog box to proportionally resize the form and all of its components.

This command works the same as Edit|Scale.

Tab Order **(Form context menu)**

See also

Choose Tab Order from the Form context menu to open the Edit Tab Order dialog box.

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.

This command works the same as Edit|Tab Order.

See also
Setting the tab order

Creation Order **(Form context menu)**

Choose Creation Order from the Form context menu to open the Creation Order dialog box.

Use this dialog box to specify the order in which your application will create nonvisual components.

This command works the same as Edit|Creation Order.

Add To Repository **(Form context menu)**

Choose Add To Repository from the Form context menu 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 context menu)**

Use this command to view a text description of the form's attributes.

Note: This command changes to View As Form when you view the form as text.

View As Form **(Code editor context menu)**

Use this command to view a unit as a form. This option is available only for units that can produce a form when the form is not already visible in the IDE.

Note: This command changes to View As Text when you view the unit as a form.

Text as DFM **(Form context menu)**

The Text as DFM command toggles the format in which this particular form file is saved. The form files in your project can be saved in one of two formats: binary or text. Text files can be modified more easily by other tools and managed by a version control system. Binary files are backward compatible with earlier versions of Delphi. For individual forms, this setting overrides the New Forms as Text check box on the Tools|Environment Options|Preferences page.

Select Icon dialog box

Use the Select Icon 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.

Show Hints

Choose Show Hints to toggle the display of Help Hints. When this command is checked, Help Hints are enabled.

Show Hints is available from the following context menus:

- Alignment Palette context menu
- Component palette context menu
- Object Inspector context menu
- Toolbar context menu

Assign Local Data dialog box

See Also

Copies the current set of records from a BDE dataset or client dataset to the selected client dataset. This is useful when populating client datasets for use as lookup tables, or when testing client datasets at design-time. Select the dataset you want to copy from the list of BDE datasets available to the current form, then click OK.

To clear the records in a client dataset at design-time, make sure the dataset's Active property is set to *True*, then right-click the client dataset and choose Clear Data.

Ole Container context menu

See also

Right-click an Ole container in the form designer to display the Ole Container context menu. In addition to the basic Form context menu options, additional items appear in this menu, depending on the state of the Ole container. These include the following:

Menu Item	Meaning	When Present
Insert Object	Brings up the Insert Object dialog box to create a new OLE object or load an existing object from a file.	Always
Paste Special	Brings up the Paste Special dialog box to load an OLE object from the Clipboard	When the clipboard contains an OLE object.
Copy Object	Copies the currently loaded OLE object to the clipboard.	When an OLE object has been loaded by Create Object or Paste Special.
Delete Object	deletes the currently loaded OLE object from the Ole container, and frees all associated memory.	When an OLE object has been loaded by Create Object or Paste Special.
Object Properties	Displays the property sheet for the currently loaded OLE object.	When the OLE object that was loaded by Create Object or Paste Special includes a property sheet.
Other commands	Additional Verbs may be added by the OLE server application. These appear before any other context menu commands.	When the OLE object that was loaded by Create Object or Paste Special includes additional verbs.

ActiveX Control Data Bindings editor

See also

After installing a data-aware ActiveX control in the ActiveX tab of the Palette, and placing the control in the form designer, right-click the data-aware ActiveX control to display a list of options. In addition to the basic Form context menu options, the additional DataBindings item appears.

Note: You must set the data source property to the data source component on the form before invoking the Data Bindings editor. In doing so, the dialog supplies the Field Name and Property fields from the data source component. The editor lists only those properties from the data source component that can be data-bound properties of the ActiveX control.

Field name lists the fields in the active database. Property Name lists those properties of the ActiveX control that can be bound to a database field. The DispID of the property is in parentheses.

To bind a field to a property,

1. Select a field name and a property name.

   Field Name lists the fields of the database and Property Name lists the ActiveX control properties that can be bound to a database field. The DispID of the property is in parentheses, for example, Value(12).

1. Click Bind and OK.

Note: If no properties appear in the dialog, the ActiveX control contains no data-aware properties. To enable simple data binding for a property of an ActiveX control, use the type library.

For a complete example that walks you through importing a data-aware control and using this dialog, see Enabling simple data binding of ActiveX controls in the container.

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 runtime, 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 runtime,

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 runtime 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 runtime, 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 runtime, 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 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 runtime

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 event handlers

Manipulating menu items at runtime

Merging menus

Disabling menu items

Opening a dialog box with a menu command

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.

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 runtime.

Choose a topic for more information.

Setting text alignment and word wrap

Using the Clipboard with text

See also

Displaying and editing text in a memo control
Displaying and editing fields in an edit box
Setting component properties.
TDBMemo component
TDBEdit component

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 runtime with a simple assignment statement.

The following topics discuss ways to set text alignment and word wrap at runtime.

Setting text alignment

Setting word wrap

See also
Displaying and editing text in a memo control

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 runtime.

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

Displaying and editing text in a memo control

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 runtime.

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

Displaying and editing text in a memo control

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

Displaying and editing text in a memo control

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

Displaying and editing text in a memo control

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

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

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;
```

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 focus 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 runtime, 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 Project|Add To Repository, which opens the Save Project Template dialog box.
3. In the Title edit box, enter a project title.

   The title for the template will appear in the Object Repository window.
4. In the Description field, enter text that describes the template.

   This text will appear in the Object Repository window's status bar.
5. 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.
6. 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.
7. Choose Browse to select an icon to represent this template in the Object Repository.
8. 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,

1. Right click on the form and choose Add To Repository.
2. In the Add To Repository dialog, select the form you want to add from the list on the left.
3. In the Title edit box, enter a title for the form.

   The title for the template will appear in the Object Repository window.
4. In the Description field, enter text that describes the template.

   This text will appear in the Object Repository window's status bar.
5. In the Page field, choose the name of the page in the New Items dialog box (probably Forms) you want the template to appear on.
6. 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.
7. Choose Browse to select an icon to represent this template in the Object Repository.
8. Choose OK to save the form as a template.

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), choose Tools|Environment Options|Preferences and set the Shared Repository Directory.

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 want 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.

Runtime errors

I/O errors Fatal errors Operating System errors Compiler error messages

Certain errors at runtime cause the program to display an error message and terminate:

Runtime error nnn at xxxxxxxx

where nnn is the runtime error number, and xxxxxxxx is the runtime error address.

Delphi applications that use the SysUtils unit map most runtime errors to Exceptions, which enable your application to resolve the error without terminating. This is called "exception handling".

The runtime errors are divided into three categories:

- I/O errors, numbered 100 through 149
- fatal errors, numbered 200 through 255
- Operating system errors

I/O errors

Fatal errors Operating System errors Compiler error messages

These errors cause termination if the particular statement was compiled in the {$I+} state. In the {$I-} state, the program continues to execute, and the error is reported by the IOResult function.

Number	Name	Description
100	Disk read error	Reported by Read on a typed file if you attempt to read past the end of the file.
101	Disk write error	Reported by CloseFile, Write, WriteIn, or Flush if the disk becomes full.
102	File not assigned	Reported by Reset, Rewrite, Append, Rename, or Erase if the file variable has not been assigned a name through a call to Assign or AssignFile.
103	File not open	Reported by CloseFile, Read Write, Seek, Eof, FilePos, FileSize, Flush, BlockRead, or BlockWrite if the file is not open.
104	File not open for input	Reported by Read, Readln, Eof, Eoln, SeekEof, or SeekEoln on a text file if the file is not open for input.
105	File not open for output	Reported by Write or WriteIn on a text file if you do not generate a Console application.
106	Invalid numeric format	Reported by Read or Readln if a numeric value read from a text file does not conform to the proper numeric format.

Fatal errors

I/O errors Operating System errors Compiler error messages

These errors always immediately terminate the program.

In applications that use the SysUtils unit (as most Delphi applications do), these errors are mapped to exceptions. For a description of the conditions that produce each error, see the documentation for the exception.

Number	Name	Exception
200	Division by zero	EDivByZero
201	Range check error	ERangeError
202	Stack overflow	EStackOverflow
203	Heap overflow error	EOutOfMemory
204	Invalid pointer operation	EInvalidPointer
205	Floating point overflow	EOverflow
206	Floating point underflow	EUnderflow
207	Invalid floating point operation	EInvalidOp
210	Abstract Method Error	EAbstractError
215	Arithmetic overflow (integer only)	EIntOverflow
216	Access violation	EAccessViolation
217	Control-C	EControlC
218	Privileged instruction	EPrivilege
219	Invalid typecast	EInvalidCast
220	Invalid variant typecast	EVariantError
221	Invalid variant operation	EVariantError
222	No variant method call dispatcher	EVariantError
223	Cannot create variant array	EVariantError
224	Variant does not contain array	EVariantError
225	Variant array bounds error	EVariantError
226	TLS initialization error	
227	Assertion failed	EAssertionFailed
228	Interface Cast Error	EIntfCastError
229	Safecall error	Windows E_UNEXPECTED error

Operating system errors

I/O errors Fatal errors Compiler error messages

All errors other than I/O errors and fatal errors are reported with the error codes returned by the Win32 error function, GetLastError. The error code values are dependent on the operating system, but you can see a list of them in the Win32 documentation.

Error reading symbol file

Delphi symbol files from earlier versions may not be compatible with later versions of Delphi. If you see this message when opening a Delphi application, close the message box and rebuild the application.

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 Toolbar buttons to display the current form, or to open the View Form dialog box.

Displaying 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.

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.

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 runtime, set its FormStyle property to fsStayOnTop.

To display a form modally,

- Call its ShowModal method.

See also

Setting component properties

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 runtime.

See also

Setting the component focus in a form

Enabling and disabling components

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 runtime.

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 runtime.

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 runtime.

To disable a component at runtime,

- Type the following code in an event handler for the component:

```
<componentn>.Enabled := False;
```

where <component*n*> 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

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 runtime corresponds to the ActiveControl property of the form.

If no component is specified as the active control, Delphi gives initial focus to any button component whose Default property is set to *True*.

If no button 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 runtime
- 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 component you want to have focus when the form first opens.

To change the active component during runtime,

- Call the SetFocus method from an event handler, for example,

```
<component n>.SetFocus;
```

where <component*n*> 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 runtime, any default button you might have specified.

See also
Setting the tab order

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 component focus in a form

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 runtime 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 runtime 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 runtime. 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 runtime,

- Call the button's SetFocus method.

See also
Executing button code on *Esc*

Setting form properties for a dialog box

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 runtime 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-resizable, 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
biMaximize	*False*	Removes Maximize button
BorderStyle	bsSizable	Enables the user to resize the form border
	bsSingle	Provides a single outline, non-resizable border
	bsNone	No distinguishable border; not resizable
	bsDialog	Window has a border but not resizable
	bsToolWindow	Makes title bar small; window is not resizable
	bsSizeToolWindow	Makes title bar small; window is resizable

Note: Changing these settings does not change the design-time appearance of the form; these property settings become visible at runtime.

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.

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 it to. 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 toolbar.

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
Compiling, building, and running projects

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

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

MDI applications

Designing menus

Adding a separator bar to a menu

- Enter a hyphen as the caption of the menu item.

Specifying keyboard shortcuts

See also

- Enter a value for the ShortCut property, 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.

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 context menu

Menu Designer context menu

See also

The Menu Designer context menu provides quick access to the most common Menu Designer commands, and to the menu template options.

To display the Menu Designer context menu,

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 context menu 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 context menu 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 context menu)

Choose Insert from the Menu Designer context menu to add a menu item placeholder before the selected menu item.

Delete (Menu Designer context menu)

Choose Delete from the Menu Designer context menu to remove the selected menu item.

Create Submenu (Menu Designer context menu)

Choose Create Submenu from the Menu Designer context menu 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 context menu)

See also

Choose Select Menu from the Menu Designer context menu 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 context menu)

See also

Choose Save As Template from the Menu Designer context menu 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 context menu)**

See also

Choose Insert From Template from the Menu Designer context menu 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 context menu)

See also

Choose Delete Templates from the Menu Designer context menu 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 context menu)

See also

Right-click the Menu Designer and choose Insert From Resource to display 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 context menu or the Object Inspector to easily select and move among them.

To use the context menu to switch among menus in a form,

1. Right-click in the Menu Designer and choose Select Menu.

   The Select Menu dialog box appears. This dialog box lists all the menus associated with the form whose menu is currently open in the Menu Designer.
2. From the list in the Select Menu dialog box, choose the menu you want to view or edit.

To use the Object Inspector to switch 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 context menu

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 DELPHI32.DMT. The menu templates shipped with Delphi also reside in this file. In a default installation, this file is in the \BIN directory. If you want to store the DELPHI32.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 and choose Insert From Template.

   (If there are no templates, the Insert From Template option appears dimmed in the context menu.)

   The Insert Template dialog box opens, displaying a list of available menu templates.
2. Select the menu template you want to insert, then press *Enter* or choose OK.

   This inserts the menu into your form at the cursor's location. For example, if your cursor is on a menu item in a list, the menu template is inserted above the selected item. If your cursor is on the menu bar, the menu template is inserted to the left of the cursor.

To delete a menu template,

1. Right-click the Menu Designer window and choose Delete Templates.

   (If there are no templates, the Delete Templates option appears dimmed in the context menu.)

   The Delete Templates dialog box opens, displaying a list of available templates.
2. Select the menu template you want to delete, and press *Del*.

   Delphi deletes the template from the templates list and from your hard disk.

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 DELPHI32.DMT file in the BIN directory.

You edit the template file by using the template commands from the Menu Designer context menu.

To save a menu as a template,

1. Choose Save As Template from the Menu Designer context menu 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. 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, because 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. Right-click and choose Insert From Resource.
   The Insert Menu From Resource dialog box appears.
3. In the dialog box, select the resource file you want to load.
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

Link not found

The topic you requested is either not available or not linked to this Help system. This can occur if you launched this Help file from a system on which Delphi has not yet been installed, or if the subject matter you are requesting is not available in your edition of Delphi.

The topic you requested is now loading. If it does not appear within a few seconds, the topic is either not available or not linked to this Help system. This can occur if you launched this Help file from a system on which Delphi has not yet been installed, or if the subject matter you are requesting is not available in your edition of Delphi.

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4112

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4113

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4114

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4116

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4117

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4118

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4119

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4110

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4121

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4122

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4123

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4124

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4126

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4127

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4128

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4129

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4131

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4133

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4134

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4135

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4136

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4137

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4138

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4139

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4130

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4141

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4142

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4143

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4144

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4145

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4146

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4147

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4148

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4149

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4140

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4151

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4152

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4153

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4154

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4155

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4156

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4157

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4158

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4159

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4150

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4161

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4162

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4163

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4164

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4165

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4166

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4167

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4168

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4169

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4160

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4172

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4173

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4174

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4175

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4176

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4177

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4178

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4179

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4170

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4181

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4182

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4183

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4184

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4185

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4186

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4187

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4188

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4189

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4180

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4191

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4192

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4193

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4194

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4195

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4196

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4197

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4198

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4190

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4101

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4102

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4103

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4104

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4105

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4106

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4107

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4108

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4109

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4100

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4011

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4012

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4013

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4014

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4015

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4016

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4017

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4018

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4019

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4010

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4021

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4022

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4023

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4024

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4025

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4026

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4027

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4028

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4029

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4020

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4031

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4032

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4033

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4034

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4035

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4036

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4037

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4038

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4039

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4030

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4041

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4042

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4043

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4044

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4045

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4046

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4047

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4048

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4049

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4040

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4051

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4052

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4053

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4054

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4055

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4056

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4057

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4058

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4059

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4050

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4061

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4062

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4063

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4064

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4065

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4066

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4067

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4068

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4069

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4071

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4072

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4073

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4074

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4075

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4076

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4077

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4078

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4079

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4070

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4081

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4082

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4083

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4084

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4085

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4086

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4088

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4089

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4080

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4091

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4092

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4093

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4094

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4095

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4096

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4097

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4098

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4099

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4090

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4001

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4002

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4003

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4004

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4005

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4006

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4007

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

4008

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_THRLMEditSource

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_DSpeedBarEditor

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_OEnvEditorDisplay

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_DBILMRange

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MWClearCompilerMessages

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_mCollectionMoveUp

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_FindHeaderFileDialog

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_mViewsViewAsText

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_ELMGotoAddress

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_DAllUnitsUsed

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGMissingDefaultMakeFile

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_FirstRunDlg

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_GBArrageByData

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_GBArrageByName

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_PMViewMakefile

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MHelpWhatsNew

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_RangeExpression

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGDuplicateContains

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_DOpenAddRequires

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_GBArrageByDescription

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MWEditSource

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_NewThreadObjectDialog

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_OProjectCppCpp

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_DFileDateChanged

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_ELMViewAsForm

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGInvalidStackBreakpoint

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_GBProperties

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_RedirectLinkDlg

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_ELMNewEditWindow

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_OProjectStaticLib

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_ConsoleWizard

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_OProjectAssembler

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSearchGotoAddress

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_PMCompile

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MWClearSearchResults

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_RedirectLinkError

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGMissingMakeSymbol

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MsgDeleteAxRegInfo

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_CompilerErrorFirst

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_PIPerviousPageAlt

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_mCollectionMoveDown

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_ViewModuleDialog

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGNoInMemoryExeProject

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_DNewInheritedForm

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_mViewsMakefile

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_mDBGridColnRestoreDefaults

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MViewsPropInsp

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_DLinkerWarnings

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_ELMMessageView

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_DCompilerWarnings

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MWindowCloseEditor

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_BPLMZoomWindow

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_DBILMShowDynamicProperties

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_ELMViewCPU

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_GBNextPage

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_mRunMTXInstall

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_DOpenAddContains

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGNotAllowedOnPasForm

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_mViewsProjectGroupSource

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_ELMProperties

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_PILocalMenu

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_NewExpression

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_ELMInspect

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGRequiresError

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_CompilerErrorLast

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGMakefileUpdated

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_OProjectATL

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGWrongProjExtension

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_DCompilerOptimizations

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGWrongUnitExtension

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_OProjectPascal

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_ELMSwapCppHdrFiles

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MWViewSource

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_EditPageNameDialog

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGNoInMemoryExeUnderWin95

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_BPLMLocalMenu

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_OProjectCppAdvCompiler

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_PINextPage

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGInHardMode

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGCantCompileHostPackage

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MFileNewDataModule

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_mCollectionAdd

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_mCollectionDelete

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MWSaveMessages

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_DResetWarning

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_mCollectionSelectAll

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGContainsError

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_PIPreviousPage

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_RCImport

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_DOpenPackage

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_OProjectCppCompiler

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_DDebuggeeFaulted

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_EditClassNameDialog

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_mDBGridColnAddAllFields

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_GBArrageByAuthor

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_mCollectionShowButtons

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_GBViweSmallIcons

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGDuplicateRequires

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_PIRevertToInherited

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGFormVarsError

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MWindow

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGDirectivesError

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_UseUnitDialog

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_PINextPageAlt

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_GBPreviousPage

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_NewInheritedFormDialog

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_GBViewDetails

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_GBViewLargeIcons

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGWrongFormExtension

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_THRLMMakeCurrent

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_GBViewList

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGMissingFormHdr

No Help topic is associated with the control or object you selected. Click here to search the index for the word or phrase you need.

HC_MSGCreateFormError

Compiler error messages

Complete list of compiler error messages

The most convenient way to get information on a message you receive in the Integrated Development Environment (IDE) is to highlight the message in the message window and press F1.

0. Ordinal type required

Complete list of compiler error messages

The compiler required an ordinal type at this point. Ordinal types are the predefined types Integer, Char, WideChar, Boolean, and declared enumerated types.

Ordinal types are required in several different situations:

- The index type of an array must be ordinal.
- The low and high bounds of a subrange type must be constant expressions of ordinal type.
- The element type of a set must be an ordinal type.
- The selection expression of a case statement must be of ordinal type.
- The first argument to the standard procedures Inc and Dec must be a variable of either ordinal or pointer type.

```
program Produce;
type
  TByteSet = set of 0..7;
var
  BitCount: array [TByteSet] of Integer;
begin
end.
```

The index type of an array must be an ordinal type - type TByteSet is a set, not an ordinal.

```
program Solve;
type
  TByteSet = set of 0..7;
var
  BitCount: array [Byte] of Integer;
begin
end.
```

Supply an ordinal type as the array index type.

1. File type not allowed here

Complete list of compiler error messages

File types are not allowed as value parameters and as the base type of a file type itself. They are also not allowed as function return types, and you cannot assign them - those errors will however produce a different error message.

```
program Produce;

procedure WriteInteger(T: Text; I: Integer);
begin
  Writeln(T, I);
end;

begin
end.
```

In this example, the problem is that T is value parameter of type Text, which is a file type. Recall that whatever gets written to a value parameter has no effect on the caller's copy of the variable - declaring a file as a value parameter therefore makes little sense.

```
program Solve;

procedure WriteInteger(var T: Text; I: Integer);
begin
  Writeln(T, I);
end;

begin
end.
```

Declaring the parameter as a var parameter solves the problem.

2. Undeclared identifier: '<name>'

Complete list of compiler error messages

The compiler could not find the given identifier - most likely it has been misspelled either at the point of declaration or the point of use. It might be from another unit that has not mentioned a uses clause.

```
program Produce;
var
  Counter: Integer;
begin
  Count := 0;
  Inc(Count);
  Writeln(Count);
end.
```

In the example the variable has been declared as "Counter", but used as "Count". The solution is to either change the declaration or the places where the variable is used.

```
program Solve;
var
  Count: Integer;
begin
  Count := 0;
  Inc(Count);
  Writeln(Count);
end.
```

In the example we have chosen to change the declaration - that was less work.

3. Identifier redeclared: '<name>'

Complete list of compiler error messages

The given identifier has already been declared in this scope - you are trying to reuse its name for something else.

```
program Tests;
var
  Tests: Integer;
begin
end.
```

Here the name of the program is the same as that of the variable - we need to change one of them to make the compiler happy.

```
program Tests;
var
  TestCnt: Integer;
begin
end.
```

4. '<name>' is not a type identifier

Complete list of compiler error messages

This error message occurs when the compiler expected the name of a type, but the name it found did not stand for a type.

```
program Produce;
type
  TMyClass = class
    Field: Integer;
  end;
var
  MyClass: TMyClass;

procedure Proc(C: MyClass);           (*<-- Error message here*)
begin
end;

begin
end.
```

The example erroneously uses the name of the variable, not the name of the type, as the type of the argument.

```
program Solve;
type
  TMyClass = class
    Field: Integer;
  end;
var
  MyClass: TMyClass;

procedure Proc(C: TMyClass);
begin
end;

begin
end.
```

Make sure the offending identifier is indeed a type - maybe it was misspelled, or another identifier of the same name hides the one you meant to refer to.

5. PACKED not allowed here

Complete list of compiler error messages

The packed keyword is only legal for set, array, record, object, class and file types. In contrast to the 16-bit version of Delphi, packed will affect the layout of record, object and class types.

```
program Produce;
type
  SmallReal = packed Real;
begin
end.
```

Packed can not be applied to a real type - if you want to conserve storage, you need to use the smallest real type, type Single.

```
program Solve;
type
  SmallReal = Single;
begin
end.
```

6. Constant or type identifier expected

Complete list of compiler error messages

This error message occurs when the compiler expects a type, but finds a symbol that is neither a constant (a constant could start a subrange type), nor a type identifier.

```
program Produce;
var
  c : ExceptionClass; (*ExceptionClass is a variable in System*)
begin
end.
```

Here, ExceptionClass is a variable, not a type.

```
program Solve;
program Produce;
var
  c : Exception; (*Exception is a type in SysUtils*)
begin
end.
```

You need to make sure you specify a type. Maybe the identifier is misspelled, or it is hidden by some other identifier, for example from another unit.

7. Incompatible types

Complete list of compiler error messages

This error message occurs when the compiler expected two types to be compatible (meaning very similar), but in fact, they turned out to be different. This error occurs in many different situations - for example when a read or write clause in a property mentions a method whose parameter list does not match the property, or when a parameter to a standard procedure or function is of the wrong type.

This error can also occur when two units both declare a type of the same name. When a procedure from an imported unit has a parameter of the same-named type, and a variable of the same-named type is passed to that procedure, the error could occur.

```
unit unit1;
interface
  type
    ExportedType = (alpha, beta, gamma);

implementation
begin
end.

unit unit2;
interface
  type
    ExportedType = (alpha, beta, gamma);

  procedure ExportedProcedure(v : ExportedType);

implementation
  procedure ExportedProcedure(v : ExportedType);
  begin
  end;

begin
end.

program Produce;
uses unit1, unit2;

var
  A: array [0..9] of char;
  I: Integer;
  V : ExportedType;
begin
  ExportedProcedure(v);
  I:= Hi(A);
end.
```

The standard function Hi expects an argument of type Integer or Word, but we supplied an array instead. In the call to ExportedProcedure, V actually is of type unit1.ExportedType since unit1 is imported prior to unit2, so an error will occur.

```
unit unit1;
interface
  type
    ExportedType = (alpha, beta, gamma);

implementation
begin
end.

unit unit2;
interface
  type
    ExportedType = (alpha, beta, gamma);

  procedure ExportedProcedure(v : ExportedType);

implementation
  procedure ExportedProcedure(v : ExportedType);
  begin
  end;

begin
end.

program Solve;
uses unit1, unit2;
var
  A: array [0..9] of char;
  I: Integer;
  V : unit2.ExportedType;
begin
  ExportedProcedure(v);
  I:= High(A);
end.
```

We really meant to use the standard function High, not Hi. For the ExportedProcedure call, there are two alternative solutions. First, you could alter the order of the uses clause, but it could also cause similar errors to occur. A more robust solution is to fully qualify the type name with the unit which declared the desired type, as has been done with the declaration for V above.

8. Incompatible types: <text>

Complete list of compiler error messages

The compiler has detected a difference between the declaration and use of a procedure.

```
program Produce;

  type
    ProcedureParm0 = procedure; stdcall;
    ProcedureParm1 = procedure(VAR x : Integer);

  procedure WrongConvention; register;
  begin
  end;

  procedure WrongParms(x, y, z : Integer);
  begin
  end;

  procedure TakesParm0(p : ProcedureParm0);
  begin
  end;

  procedure TakesParm1(p : ProcedureParm1);
  begin
  end;

begin
  TakesParm0(WrongConvention);
  TakesParm1(WrongParms);
end.
```

The call of 'TakesParm0' will elicit an error because the type 'ProcedureParm0' expects a 'stdcall' procedure, whereas 'WrongConvention' is declared with the 'register' calling convention. Similarly, the call of 'TakesParm1' will fail because the parameter lists do not match.

```
program Solve;

  type
    ProcedureParm0 = procedure; stdcall;
    ProcedureParm1 = procedure(VAR x : Integer);

  procedure RightConvention; stdcall;
  begin
  end;

  procedure RightParms(VAR x : Integer);
  begin
  end;

  procedure TakesParm0(p : ProcedureParm0);
  begin
  end;

  procedure TakesParm1(p : ProcedureParm1);
  begin
  end;

begin
  TakesParm0(RightConvention);
  TakesParm1(RightParms);
end.
```

The solution to both of these problems is to ensure that the calling convention or the parameter lists matches the declaration.

9. Incompatible types: '<name>' and '<name>'

Complete list of compiler error messages

This error message results when the compiler expected two types to be compatible (i.e. similar), but they turned out to be different.

```
program Produce;

procedure Proc(I: Integer);
begin
end;

begin
  Proc( 22 / 7 ); (*Result of / operator is Real*)
end.
```

Here a C++ programmer thought the division operator / would give him an integral result - not the case in Pascal.

```
program Solve;

procedure Proc(I: Integer);
begin
end;

begin
  Proc( 22 div 7 ); (*The div operator gives result type
Integer*)
end.
```

The solution in this case is to use the integral division operator div - in general, you have to look at your program very careful to decide how to resolve type incompatibilities.

10. Low bound exceeds high bound

Complete list of compiler error messages

This error message is given when either the low bound of a subrange type is greater than the high bound, or the low bound of a case label range is greater than the high bound.

```
program Produce;
type
  SubrangeType = 1..0;              (*Gets: Low bound exceeds
high bound *)
begin
  case True of
  True..False:                      (*Gets: Low bound exceeds
high bound *)
    Writeln('Expected result');
  else
    Writeln('Unexpected result');
  end;
end.
```

In the example above, the compiler gives an error rather than treating the ranges as empty. Most likely, the reversal of the bounds was not intentional.

```
program Solve;
type
  SubrangeType = 0..1;
begin
  case True of
  False..True:
    Writeln('Expected result');
  else
    Writeln('Unexpected result');
  end;
end.
```

Make sure you have specified the bounds in the correct order.

11. Type of expression must be BOOLEAN

Complete list of compiler error messages

This error message is output when an expression serves as a condition and must therefore be of Boolean type. This is the case for the controlling expression of the if, while and repeat statements, and for the expression that controls a conditional breakpoint.

```
program Produce;
var
  P: Pointer;
begin
  if P then
    Writeln('P <> nil');
end.
```

Here, a C++ programmer just used a pointer variable as the condition of an if statement.

```
program Solve;
var
  P: Pointer;
begin
  if P <> nil then
    Writeln('P <> nil');
end.
```

In Pascal, you need to be more explicit in this case.

12. Type of expression must be INTEGER

Complete list of compiler error messages

This error message is only given when the constant expression that specifies the number of characters in a string type is not of type integer.

```
program Produce;
type
  color = (red,green,blue);
var
  S3 : string[Succ(High(color))];
begin
end.
```

The example tries to specify the number of elements in a string as dependent on the maximum element of type color - unfortunately, the element count is of type color, which is illegal.

```
program Solve;
type
  color = (red,green,blue);
var
  S3 : string[ord(High(color))+1];
begin
end.
```

13. Statement expected, but expression of type '<type>' found

Complete list of compiler error messages

The compiler was expecting to find a statement, but instead it found an expression of the specified type.

```
program Produce;
  var
    a : Integer;
begin
   (3 + 4);
end.
```

In this example, the compiler is expecting to find a statement, such as an IF, WHILE, REPEAT, but instead it found the expression (3+4).

```
program Produce;
  var
    a : Integer;
begin
   a := (3 + 4);
end.
```

The solution here was to assign the result of the expression (3+4) to the variable 'a'. Another solution would have been to remove the offending expression from the source code - the choice depends on the situation.

14. Operator not applicable to this operand type

Complete list of compiler error messages

This error message is given whenever an operator cannot be applied to the operands it was given - for instance if a boolean operator is applied to a pointer.

```
program Produce;
var
  P: ^Integer;
begin
  if P and P^ > 0 then
    Writeln('P points to a number greater 0');
end.
```

Here a C++ programmer was unclear about operator precedence in Pascal - P is not a boolean expression, and the comparison needs to be parenthesized.

```
program Solve;
var
  P: ^Integer;
begin
  if (P <> nil) and (P^ > 0) then
    Writeln('P points to a number greater 0');
end.
```

If we explicitly compare P to nil and use parentheses, the compiler is happy.

15. Array type required

Complete list of compiler error messages

This error message is given if you either index into an operand that is not an array, or if you pass an argument that is not an array to an open array parameter.

```
program Produce;
var
  P: ^Integer;
  I: Integer;
begin
  Writeln(P[I]);
end.
```

We try to apply an index to a pointer to integer - that would be legal in C, but is not in Pascal.

```
program Solve;
type
  TIntArray = array [0..MaxInt DIV sizeof(Integer)-1] of Integer;
var
  P: ^TIntArray;
  I: Integer;
begin
  Writeln(P^[I]);   (*Actually, P[I] would also be legal in
Delphi32*)
end.
```

In Pascal, we must tell the compiler that we intend P to point to an array of integers.

16. Pointer type required

Complete list of compiler error messages

This error message is given when you apply the dereferencing operator '^' to an operand that is not a pointer, and, as a very special case, when the second operand in a 'Raise <exception> at <address>' statement is not a pointer.

```
program Produce;
var
  C: TObject;
begin
  C^.Destroy;
end.
```

Even though class types are implemented internally as pointers to the actual information, it is illegal to apply the dereferencing operator to class types at the source level. It is also not necessary - the compiler will dereference automatically whenever it is appropriate.

```
program Solve;
var
  C: TObject;
begin
  C.Destroy;
end.
```

Simply leave off the dereferencing operator—the compiler will do the right thing automatically.

17. Record, object or class type required

Complete list of compiler error messages

The compiler was expecting to find the type name which specified a record, object or class but did not find one.

```
program Produce;

  type
    RecordDesc = class
      ch : Char;
    end;

  var
    pCh : PChar;
    r : RecordDesc;

  procedure A;
  begin
    pCh.ch := 'A';    (* case 1 *)

    with pCh do begin (* case 2 *)
    end;
  end;
end.
```

There are two causes for the same error in this program. The first is the application of '.' to a object that is not a record. The second case is the use of a variable which is of the wrong type in a WITH statement.

```
program Solve;

  type
    RecordDesc = class
      ch : Char;
    end;

  var
    r : RecordDesc;

  procedure A;
  begin
    r.ch := 'A';    (* case 1 *)

    with r do begin (* case 2 *)
    end;
  end;
end.
```

The easy solution to this error is to always make sure that the '.' and WITH are both applied only to records, objects or class variables.

18. Object type required

Complete list of compiler error messages

This error is given whenever an object type is expected by the compiler. For instance, the ancestor type of an object must also be an object type.

```
type
  MyObject = object(TObject)
  end;
begin
end.
```

Confusingly enough, TObject in the unit System has a class type, so we cannot derive an object type from it.

```
program Solve;
type
  MyObject = class  (*Actually, this means: class(TObject)*)
  end;
begin
end.
```

Make sure the type identifier really stands for an object type - maybe it is misspelled, or maybe is hidden by an identifier from another unit.

19. Object or class type required

Complete list of compiler error messages

This error message is given when the syntax 'Typename.Methodname' is used, but the typename does not refer to an object or class type.

```
program Produce;
type
  TInteger = class
    Value: Integer;
  end;
var
  V: TInteger;
begin
  V := Integer.Create;
end.
```

Type Integer does not have a Create method, but TInteger does.

```
program Solve;
type
  TInteger = class
    Value: Integer;
  end;
var
  V: TInteger;
begin
  V := TInteger.Create;
end.
```

Make sure the identifier really refers to an object or class type - maybe it is misspelled or it is hidden by an identifier from another unit.

20. Class type required

Complete list of compiler error messages

In certain situations the compiler requires a class type:

- As the ancestor of a class type
- In the on-clause of a try-except statement
- As the first argument of a raise statement
- As the final type of a forward declared class type

```
program Produce;
begin
  raise 'This would work in C++, but does not in Delphi';
end.
```

```
program Solve;
uses SysUtils;
begin
  raise Exception.Create('There is a simple workaround,
however');
end.
```

21. Function needs result type

Complete list of compiler error messages

You have declared a function, but have not specified a return type.

```
program Produce;

function Sum(A: array of Integer);
var I: Integer;
begin
  Result := 0;
  for I := 0 to High(A) do
    Result := Result + A[I];
end;

begin
end.
```

Here Sum is meant to be function, we have not told the compiler about it.

```
program Solve;

function Sum(A: array of Integer): Integer;
var I: Integer;
begin
  Result := 0;
  for I := 0 to High(A) do
    Result := Result + A[I];
end;

begin
end.
```

Just make sure you specify the result type.

22. Invalid function result type

Complete list of compiler error messages

File types are not allowed as function result types.

```
program Produce;

function OpenFile(Name: string): File;
begin
end;

begin
end.
```

You cannot return a file from a function.

```
program Solve;

procedure OpenFile(Name: string; var F: File);
begin
end;

begin
end.
```

You can 'return' the file as a variable parameter. Alternatively, you can also allocate a file dynamically and return a pointer to it.

23. Procedure cannot have a result type

Complete list of compiler error messages

You have declared a procedure, but given it a result type. Either you really meant to declare a function, or you should delete the result type.

```
program Produce;

procedure DotProduct(const A,B: array of Double): Double;
var
  I: Integer;
begin
  Result := 0.0;
  for I := 0 to High(A) do
    Result := Result + A[I]*B[I];
end;

const
  C: array [1..3] of Double = (1,2,3);

begin
  Writeln( DotProduct(C,C) );
end.
```

Here DotProduct was really meant to be a function, we just happened to use the wrong keyword...

```
program Solve;

function DotProduct(const A,B: array of Double): Double;
var
  I: Integer;
begin
  Result := 0.0;
  for I := 0 to High(A) do
    Result := Result + A[I]*B[I];
end;

const
  C: array [1..3] of Double = (1,2,3);

begin
  Writeln( DotProduct(C,C) );
end.
```

Just make sure you specify a result type when you declare a function, and no result type when you declare a procedure.

24. Text after final 'END.' - ignored by compiler

Complete list of compiler error messages

This warning is given when there is still source text after the final end and the period that constitute the logical end of the program. Possibly the nesting of begin-end is inconsistent (there is one `end` too many somewhere). Check whether you intended the source text to be ignored by the compiler - maybe it is actually quite important.

```
program Produce;

begin
end.

Text here is ignored by Delphi 16-bit - Delphi 32-bit gives a
warning.


program Solve;

begin
end.
```

25. Constant expression expected

Complete list of compiler error messages

The compiler expected a constant expression here, but the expression it found turned out not to be constant.

```
program Produce;
const
  Message = 'Hello World!';
  WPosition = Pos('W', Message);
begin
end.
```

The call to Pos is not a constant expression to the compiler, even though its arguments are constants, and it could in principle be evaluated at compile time.

```
program Solve;
const
  Message = 'Hello World!';
  WPosition = 7;
begin
end.
```

So in this case, we just have to calculate the right value for WPosition ourselves.

26. Constant expression violates subrange bounds

Complete list of compiler error messages

This error message occurs when the compiler can determine that a constant is outside the legal range. This can occur for instance if you assign a constant to a variable of subrange type.

```
program Produce;
var
  Digit: 1..9;
begin
  Digit := 0;  (*Get message: Constant expression violates
subrange bounds*)
end.
```

```
program Solve;
var
  Digit: 0..9;
begin
  Digit := 0;
end.
```

27. Duplicate tag value

Complete list of compiler error messages

This error message is given when a constant appears more than once in the declaration of a variant record.

```
program Produce;
type
  VariantRecord = record
    case Integer of
    0: (IntField: Integer);
    0: (RealField: Real);     (*<-- Error message here*)
  end;

begin
end.
```

```
program Solve;
type
  VariantRecord = record
    case Integer of
    0: (IntField: Integer);
    1: (RealField: Real);
  end;

begin
end.
```

28. Sets may have at most 256 elements

Complete list of compiler error messages

This error message appears when you try to declare a set type of more than 256 elements. More precisely, the ordinal values of the upper and lower bounds of the base type must be within the range 0..255.

```
program Produce;
type
  BigSet = set of 1..256;  (*<-- error message given here*)
begin
end.
```

In the example, BigSet really only has 256 elements, but is still illegal.

```
program Solve;
type
  BigSet = set of 0..255;
begin
end.
```

We need to make sure the upper and lower bounds and in the range 0..255.

29. <Token1> expected but <token2> found

Complete list of compiler error messages

This error message appears for syntax errors. There is probably a typo in the source, or something was left out. When the error occurs at the beginning of a line, the actual error is often on the previous line.

```
program Produce;
var
  I: Integer
begin               (*<-- Error message here: ';' expected but
'BEGIN' found*)
end.
```

After the type Integer, the compiler expects to find a semicolon to terminate the variable declaration. It does not find the semicolon on the current line, so it reads on and finds the 'begin' keyword at the start of the next line. At this point it finally knows something is wrong...

```
program Solve;
var
  I: Integer;       (*Semicolon was missing*)
begin
end.
```

In this case, just the semicolon was missing - a frequent case in practice. In general, have a close look at the line where the error message appears, and the line above it to find out whether something is missing or misspelled.

30. Duplicate case label

Complete list of compiler error messages

This error message occurs when there is more than one case label with a given value in a case statement.

```
program Produce;

function DigitCount(I: Integer): Integer;
begin
  case Abs(I) of
  0:                    DigitCount := 1;
  0        ..9:         DigitCount := 1;   (*<-- Error message
here*)
  10       ..99:        DigitCount := 2;
  100      ..999:       DigitCount := 3;
  1000     ..9999:      DigitCount := 4;
  10000    ..99999:     DigitCount := 5;
  100000   ..999999:    DigitCount := 6;
  1000000  ..9999999:   DigitCount := 7;
  10000000 ..99999999:  DigitCount := 8;
  100000000..999999999: DigitCount := 9;
  else                  DigitCount := 10;
  end;
end;

begin
  Writeln( DigitCount(12345) );
end.
```

Here we did not pay attention and mentioned the case label 0 twice.

```
program Solve;

function DigitCount(I: Integer): Integer;
begin
  case Abs(I) of
  0        ..9:         DigitCount := 1;
  10       ..99:        DigitCount := 2;
  100      ..999:       DigitCount := 3;
  1000     ..9999:      DigitCount := 4;
  10000    ..99999:     DigitCount := 5;
  100000   ..999999:    DigitCount := 6;
  1000000  ..9999999:   DigitCount := 7;
  10000000 ..99999999:  DigitCount := 8;
  100000000..999999999: DigitCount := 9;
  else                  DigitCount := 10;
  end;
end;

begin
  Writeln( DigitCount(12345) );
end.
```

In general, the problem might not be so easy to spot when you have symbolic constants and ranges of case labels - you might have to write down the real values of the constants to find out what is wrong.

31. Label expected

Complete list of compiler error messages

This error message occurs if the identifier given in a goto statement or used as a label in inline assembly is not declared as a label.

```
program Produce;

begin
  if 2*2 <> 4 then
    goto Exit; (*<-- Error message here: Exit is also a standard
procedure*)
  (*...*)
Exit:               (*Additional error messages here*)
end.
```

```
program Solve;
label
  Exit;             (*Labels must be declared in Pascal*)
begin
  if 2*2 <> 4 then
    goto Exit;
  (*...*)
Exit:
end.
```

32. For loop control variable must be simple local variable

Complete list of compiler error messages

This error message is given when the control variable of a for statement is not a simple variable (but a component of a record, for instance), or if it is not local to the procedure containing the for statement.

For backward compatibility reasons, it is legal to use a global variable as the control variable - the compiler gives a warning in this case. Note that using a local variable will also generate more efficient code.

```
program Produce;

var
  I: Integer;
  A: array [0..9] of Integer;

procedure Init;
begin
  for I := Low(A) to High(a) do  (*<-- Warning given here*)
    A[I] := 0;
end;

begin
  Init;
end.
```

```
program Solve;
var
  A: array [0..9] of Integer;

procedure Init;
var
  I: Integer;
begin
  for I := Low(A) to High(a) do
    A[I] := 0;
end;

begin
  Init;
end.
```

33. For loop control variable must have ordinal type

Complete list of compiler error messages

The control variable of a for loop must have type Boolean, Char, WideChar, Integer, an enumerated type, or a subrange type.

```
program Produce;
var
  x: Real;
begin (*Plot sine wave*)
  for x := 0 to 2*pi/0.2 do                              (*<--
Error message here*)
    Writeln( '*': Round((Sin(x*0.2) + 1)*20) + 1 );
end.
```

The example uses a variable of type Real as the for loop control variable, which is illegal.

```
program Solve;
var
  x: Integer;
begin (*Plot sine wave*)
  for x := 0 to Round(2*pi/0.2) do
    Writeln( '*': Round((Sin(x*0.2) + 1)*20) + 1 );
end.
```

Instead, use the Integer ordinal type.

You may see this warning if a FOR loop uses an Int64 control variable. This results from a limitation in the compiler which you can work around by replacing the FOR loop with a WHILE loop.

34. Types of actual and formal var parameters must be identical

Complete list of compiler error messages

For a variable parameter, the actual argument must be of the exact type of the formal parameter.

```
program Produce;

procedure SwapBytes(var B1, B2: Byte);
var
  Temp: Byte;
begin
  Temp := B1; B1 := B2; B2 := Temp;
end;

var
  C1, C2: 0..255;     (*Similar to a byte, but NOT identical*)
begin
  SwapBytes(C1,C2);   (*<-- Error message here*)
end.
```

Arguments C1 and C2 are not acceptable to SwapBytes, although they have the exact memory representation and range that a Byte has.

```
program Solve;

procedure SwapBytes(var B1, B2: Byte);
var
  Temp: Byte;
begin
  Temp := B1; B1 := B2; B2 := Temp;
end;

var
  C1, C2: Byte;
begin
  SwapBytes(C1,C2);   (*<-- Error message here*)
end.
```

So you actually have to declare C1 and C2 as Bytes to make this example compile.

35. Too many actual parameters

Complete list of compiler error messages

This error message occurs when a procedure or function call gives more parameters than the procedure or function declaration specifies.

Additionally, this error message occurs when an OLE automation call has too many (more than 255), or too many named parameters.

```
program Produce;

function Max(A,B: Integer): Integer;
begin
  if A > B then Max := A else Max := B
end;

begin
  Writeln( Max(1,2,3) );    (*<-- Error message here*)
end.
```

It would have been convenient for Max to accept three parameters...

```
program Solve;

function Max(const A: array of Integer): Integer;
var
  I: Integer;
begin
  Result := Low(Integer);
  for I := 0 to High(A) do
    if Result < A[I] then
      Result := A[I];
end;

begin
  Writeln( Max([1,2,3]) );
end.
```

Normally, you would change to call site to supply the right number of parameters. Here, we have chose to show you how to implement Max with an unlimited number of arguments. Note that now you have to call it in a slightly different way.

36. Not enough actual parameters

Complete list of compiler error messages

This error message occurs when a call to procedure or function gives less parameters than specified in the procedure or function declaration.

This can also occur for calls to standard procedures or functions.

```
program Produce;
var
  X: Real;
begin
  Val('3.141592', X);    (*<-- Error message here*)
end.
```

The standard procedure Val has one additional parameter to return an error code in. The example did not supply that parameter.

```
program Solve;
var
  X: Real;
  Code: Integer;
begin
  Val('3.141592', X, Code);
end.
```

Typically, you will check the call against the declaration of the procedure called or the help, and you will find you forgot about a parameter you need to supply.

37. Variable required

Complete list of compiler error messages

This error message occurs when you try to take the address of an expression or a constant.

```
program Produce;
var
  I: Integer;
  PI: ^Integer;
begin
  PI := Addr(1);
end.
```

A constant like 1 does not have a memory address, so you cannot apply the operator or the Addr standard function to it.

```
program Solve;
var
  I: Integer;
  PI: ^Integer;
begin
  PI := Addr(I);
end.
```

You need to make sure you take the address of variable.

38. Declaration of <Name> differs from previous declaration

Complete list of compiler error messages

This error message occurs when the declaration of a procedure, function, method, constructor or destructor differs from its previous (forward) declaration.

This error message also occurs when you try to override a virtual method, but the overriding method has a different parameter list, calling convention etc.

```
program Produce;

type
  MyClass = class
    procedure Proc(Inx: Integer);
    function Func: Integer;
    procedure Load(const Name: string);
    procedure Perform(Flag: Boolean);
    constructor Create;
    destructor Destroy(Msg: string); override;      (*<-- Error
message here*)
    class function NewInstance: MyClass; override;  (*<-- Error
message here*)
  end;

procedure MyClass.Proc(Index: Integer);             (*<-- Error
message here*)
begin
end;

function MyClass.Func: Longint;                     (*<-- Error
message here*)
begin
end;

procedure MyClass.Load(Name: string);               (*<-- Error
message here*)
begin
end;

procedure MyClass.Perform(Flag: Boolean); cdecl;    (*<-- Error
message here*)
begin
end;

procedure MyClass.Create;                           (*<-- Error
message here*)
begin
end;

function MyClass.NewInstance: MyClass;              (*<-- Error
message here*)
begin
end;

begin
end.
```

As you can see, there are a number of reasons for this error message to be issued.

```
program Solve;

type
  MyClass = class
    procedure Proc(Inx: Integer);
    function Func: Integer;
    procedure Load(const Name: string);
    procedure Perform(Flag: Boolean);
    constructor Create;
    destructor Destroy; override;                    (*No
parameters*)
    class function NewInstance: TObject; override;  (*Result type
*)
  end;

procedure MyClass.Proc(Inx: Integer);               (*Parameter
name  *)
begin
end;

function MyClass.Func: Integer;                     (*Result type
*)
begin
end;

procedure MyClass.Load(const Name: string);         (*Parameter
kind  *)
begin
end;

procedure MyClass.Perform(Flag: Boolean);           (*Calling
convention*)
begin
end;

constructor MyClass.Create;
(*constructor*)
begin
end;

class function MyClass.NewInstance: TObject;        (*class
function*)
begin
end;

begin
end.
```

You need to carefully compare the 'previous declaration' with the one that causes the error to determine what is different between the two.

39. Illegal character in input file: '<char>' ($<hex>)

Complete list of compiler error messages

The compiler found a character that is illegal in Pascal programs.

This error message is caused most often by errors with string constants or comments.

```
program Produce;

begin
  Writeln("Hello world!");   (*<-- Error messages here*)
end.
```

Here a programmer fell back to C++ habits and quoted a string with double quotes.

```
program Solve;

begin
  Writeln('Hello world!');   (*Need single quotes in Pascal*)
end.
```

The solution is to use single quotes. In general, you need to delete the illegal character.

40. File not found: <Filename>

Complete list of compiler error messages

This error message occurs when the compiler cannot find an input file. This can be a source file, a compiled unit file (.dcu file), an include, an object file or a resource file.

Check the spelling of the name and the relevant search path.

```
program Produce;
uses SysUtilss;       (*<-- Error message here*)
begin
end.
```

```
program Solve;
uses SysUtils;      (*Fixed typo*)
begin
end.
```

41. Could not create output file <Filename>

Complete list of compiler error messages

The compiler could not create an output file. This can be a compiled unit file (.dcu file), an executable file, a map file or an object file.

Most likely causes are a nonexistent directory or a write protected file or disk.

42. Seek error on <Filename>

Complete list of compiler error messages

The compiler encountered a seek error on an input or output file.

This should never happen - if it does, the most likely cause is corrupt data.

43. Read error on <Filename>

Complete list of compiler error messages

The compiler encountered a read error on an input file.

This should never happen - if it does, the most likely cause is corrupt data.

44. Write error on <Filename>

Complete list of compiler error messages

The compiler encountered a write error while writing to an output file.

Most likely, the output disk is full.

45. Close error on <Filename>

Complete list of compiler error messages

The compiler encountered an error while closing an input or output file.

This should rarely happen. If it does, the most likely cause is a full or bad disk.

46. Bad file format: <Filename>

Complete list of compiler error messages

This error occurs if an object file loaded with a $L or $LINK directive is not of the correct format. Several restrictions must be met:

- Check the naming restrictions on segment names in the help file
- Not more than 10 segments
- Not more than 255 external symbols
- Not more than 50 local names in LNAMES records
- LEDATA and LIDATA records must be in offset order
- No THREAD subrecords are supported in FIXU32 records
- Only 32-bit offsets can be fixed up
- Only segment and self relative fixups
- Target of a fixup must be a segment, a group or an EXTDEF
- Object must be 32-bit object file
- Various internal consistency condition that should only fail if the object file is corrupted.

47. Out of memory

Complete list of compiler error messages

The compiler ran out of memory.

This should rarely happen. If it does, make sure your swap file is large enough and that there is still room on the disk.

48. Circular unit reference to <Unitname>

Complete list of compiler error messages

One or more units use each other in their interface parts.

As the compiler has to translate the interface part of a unit before any other unit can use it, the compiler must be able to find a compilation order for the interface parts of the units.

Check whether all the units in the uses clauses are really necessary, and whether some can be moved to the implementation part of a unit instead.

```
unit A;
interface
uses B;           (*A uses B, and B uses A*)
implementation
end.

unit B;
interface
uses A;
implementation
end.
```

The problem is caused because A and B use each other in their interface sections.

```
unit A;
interface
uses B;          (*Compilation order: B.interface, A,
B.implementation*)
implementation
end.

unit B;
interface
implementation
uses A;          (*Moved to the implementation part*)
end.
```

You can break the cycle by moving one or more uses to the implementation part.

49. Bad unit format: <Filename>

Complete list of compiler error messages

This error occurs if a compiled unit file (.dcu file) has a bad format.

Most likely, the .dcu file has been corrupted. Recompile the file or reinstall Delphi32.

50. Label declaration not allowed in interface part

Complete list of compiler error messages

This error occurs when you declare a label in the interface part of a unit.

```
unit Produce;
interface
label 99;
implementation
begin
99:
end.
```

It is just illegal to declare a label in the interface section of a unit.

```
unit Solve;
interface
implementation
label 99;
begin
99:
end.
```

You have to move it to the implementation section.

51. Statements not allowed in interface part

Complete list of compiler error messages

The interface part of a unit can only contain declarations, not statements.

Move the bodies of procedures to the implementation part.

```
unit Produce;

interface

procedure MyProc;
begin                (*<-- Error message here*)
end;

implementation

begin
end.
```

We got carried away and gave MyProc a body right in the interface section.

```
unit Solve;

interface

procedure MyProc;

implementation

procedure MyProc;
begin
end;

begin
end.
```

We need move the body to the implementation section - then it's fine.

52. Unit <Unit1> was compiled with a different version of <Unit2>

Complete list of compiler error messages

This error occurs when the declaration of symbol declared in the interface part of a unit has changed, and the compiler cannot recompile a unit that relies on this declaration because the source is not available to it.

There are several possible solutions - recompile Unit1 (assuming you have the source code available), use an older version of Unit2 or change Unit2, or get a new version of Unit1 from whoever has the source code to it.

53. Unterminated string

Complete list of compiler error messages

The compiler did not find a closing apostrophe at the end of a character string.

Note that character strings cannot be continued onto the next line - however, you can use the '+' operator to concatenate two character strings on separate lines.

```
program Produce;

begin
  Writeln('Hello world!);   (*<-- Error message here -*)
end.
```

We just forgot the closing quote at the string - no big deal, happens all the time.

```
program Solve;

begin
  Writeln('Hello world!');
end.
```

So we supplied the closing quote, and the compiler is happy.

54. Syntax error in real number

Complete list of compiler error messages

This error message occurs if the compiler finds the beginning of a scale factor (an 'E' or 'e' character) in a number, but no digits follow it.

```
program Produce;
const
  SpeedOfLight = 3.0E 8;     (*<-- Error message here*)
begin
end.
```

In the example, we put a space after '3.0E' - now for the compiler the number ends here, and it is incomplete.

```
program Solve;
const
  SpeedOfLight = 3.0E+8;
begin
end.
```

We could have just deleted the blank, but we put in a '+' sign because it looks a little nicer.

55. Illegal type in Write/Writeln statement

Complete list of compiler error messages

This error occurs when you try to output a type in a Write or Writeln statement that is not legal.

```
program Produce;
type
  TColor = (red,green,blue);
var
  Color : TColor;
begin
  Writeln(Color);
end.
```

It would have been convenient to use a writeln statement to output Color, wouldn't it?

```
program Solve;
type
  TColor = (red,green,blue);
var
  Color : TColor;
const
  ColorString : array [TColor] of string = ('red', 'green',
'blue');
begin
  Writeln(ColorString[Color]);
end.
```

Unfortunately, that is not legal, and we have to do it with an auxiliary table.

56. Illegal type in Read/Readln statement

Complete list of compiler error messages

This error occurs when you try to read a variable in a Read or Readln that is not of a legal type.

Check the type of the variable and make sure you are not missing a dereferencing, indexing or field selection operator.

```
program Produce;
type
  TColor = (red,green,blue);
var
  Color : TColor;
begin
  Readln(Color);     (*<-- Error message here*)
end.
```

We cannot read variables of enumerated types directly.

```
program Solve;
type
  TColor = (red,green,blue);
var
  Color : TColor;
  InputString: string;
const
  ColorString : array [TColor] of string = ('red', 'green',
'blue');
begin
  Readln(InputString);
  Color := red;
  while (color < blue) and (ColorString[color] <> InputString) do
    Inc(color);
end.
```

The solution is to read a string, and look up that string in an auxiliary table. In the example above, we didn't bother to do error checking - any string will be treated as 'blue'. In practice, we would probably output an error message and ask the user to try again.

57. Strings may have at most 255 elements

Complete list of compiler error messages

This error message occurs when you declare a string type with more than 255 elements, if you assign a string literal of more than 255 characters to a variable of type ShortString, or when you have more than 255 characters in a single character string.

Note that you can construct long string literals spanning more than one line by using the '+' operator to concatenate several string literals.

```
program Produce;
var
  LongString : string[256];  (*<-- Error message here*)
begin
end.
```

In the example above, the length of the string is just one beyond the limit.

```
program Solve;
var
  LongString : AnsiString;
begin
end.
```

The most convenient solution is to use the new long strings - then you don't even have to spend any time thinking about what a reasonable maximum length would be.

58. Unexpected end of file in comment started on line <Number>

Complete list of compiler error messages

This error occurs when you open a comment, but do not close it.

Note that a comment started with '{' must be closed with '}', and a comment started with '(*' must be closed with '*)'.

```
program Produce;
(*Let's start a comment here but forget to close it
begin
end.
```

So the example just didn't close the comment.

```
program Solve;
(*Let's start a comment here and not forget to close it*)
begin
end.
```

Doing so fixes the problem.

59. Invalid compiler directive: '<Directive>'

Complete list of compiler error messages

This error message means there is an error in a compiler directive or in a command line option. Here are some possible error situations:

- An external declaration was syntactically incorrect.

- A command line option or an option in a DCC32.CFG file was not recognized by the compiler or was invalid. For example, '-$M100' is invalid because the minimum stack size must be at least 1024.

- The compiler found a $XXXXX directive, but could not recognize it. It was probably misspelled.

- The compiler found a $ELSE or $ENDIF directive, but no preceding $IFDEF, $IFNDEF or $IFOPT directive.

- (*$IFOPT*) was not followed by a switch option and a + or -.

- The long form of a switch directive was not followed by ON or OFF.

- A directive taking a numeric parameter was not followed by a valid number.

- The $DESCRIPTION directive was not followed by a string.

- The $APPTYPE directive was not followed by CONSOLE or GUI.

- The $ENUMSIZE directive (short form $Z) was not followed by 1,2 or 4.

```
(*$Description Copyright Borland International 1996*)     (*<--
Error here*)
program Produce;
(*$AppType Console*)                                       (*<--
Error here*)

begin
(*$If O+*)                                                 (*<--
Error here*)
  Writeln('Optimizations are ON');
(*$Else*)                                                  (*<--
Error here*)
  Writeln('Optimizations are OFF');
(*$Endif*)                                                 (*<--
Error here*)
  Writeln('Hello world!');
end.
```

The example shows three typical error situations, and the last two errors are caused by the compiler not having recognized $If.

```
(*$Description 'Copyright Borland International 1996'*)  (*Need
string*)
program Solve;
(*$AppType Console*)
(*AppType*)

begin
(*$IfOpt O+*)
(*IfOpt*)
  Writeln('Optimizations are ON');
(*$Else*)                                                 (*Now
fine*)
  Writeln('Optimizations are OFF');
(*$Endif*)                                                (*Now
fine*)
  Writeln('Hello world!');
end.
```

So $Description needs a quoted string, we need to spell $AppType right, and checking options is done with $IfOpt. With these changes, the example compiles fine.

60. Bad global symbol definition: '<Name>' in object file '<Filename>'

Complete list of compiler error messages

This warning is given when an object file linked in with a $L or $LINK directive contains a definition for a symbol that was not declared in Pascal as an external procedure, but as something else (e.g. a variable).

The definition in the object will be ignored in this case.

61. Class or object types only allowed in type section

Complete list of compiler error messages

Class or object types must always be declared with an explicit type declaration in a type section - unlike record types, they cannot be anonymous.

The main reason for this is that there would be no way you could declare the methods of that type - after all, there is no type name.

```
program Produce;

var
  MyClass : class
    Field: Integer;
  end;

begin
end.
```

The example tries to declare a class type within a variable declaration - that is not legal.

```
program Solve;

type
  TMyClass = class
    Field: Integer;
  end;

var
  MyClass : TMyClass;

begin
end.
```

The solution is to introduce a type declaration for the class type. Alternatively, you could have changed the class type to a record type.

62. Local class or object types not allowed

Complete list of compiler error messages

Class and object cannot be declared local to a procedure.

```
program Produce;

  procedure MyProc;
  type
    TMyClass = class
      Field: Integer;
    end;
  begin
  (*...*)
  end;

begin
end.
```

So MyProc tries to declare a class type locally, which is illegal.

```
program Solve;

  type
    TMyClass = class
      Field: Integer;
    end;

  procedure MyProc;
  begin
  (*...*)
  end;

begin
end.
```

The solution is to move out the declaration of the class or object type to the global scope.

63. Virtual constructors are not allowed

Complete list of compiler error messages

Unlike class types, object types can only have static constructors.

```
program Produce;

type
  TMyObject = object
    constructor Init; virtual;
  end;

constructor TMyObject.Init;
begin
end;

begin
end.
```

The example tries to declare a virtual constructor, which does not really make sense for object types and is therefore illegal.

```
program Solve;

type
  TMyObject = object
    constructor Init;
  end;

constructor TMyObject.Init;
begin
end;

begin
end.
```

The solution is to either make the constructor static, or to use a new-style class type which can have a virtual constructor.

64. Could not compile used unit '<Unitname>'

Complete list of compiler error messages

This fatal error is given when a unit used by another could not be compiled. In this case, the compiler gives up compilation of the dependent unit because it is likely very many errors will be encountered as a consequence.

65. Left side cannot be assigned to

Complete list of compiler error messages

This error message is given when you try to modify a read-only object like a constant, a constant parameter, or the return value of function.

```
program Produce;

const
  c = 1;

procedure p(const s: string);
begin
  s := 'changed';            (*<-- Error message here*)
end;

function f: PChar;
begin
  f := 'Hello';              (*This is fine - we are setting the
return value*)
end;

begin
  c := 2;                    (*<-- Error message here*)
  f := 'h';                  (*<-- Error message here*)
end.
```

The example assigns to constant parameter, to a constant, and to the result of a function call. All of these are illegal.

```
program Solve;

var
  c : Integer = 1;           (*Use an initialized variable*)

procedure p(var s: string);
begin
  s := 'changed';            (*Use variable parameter*)
end;

function f: PChar;
begin
  f := 'Hello';              (*This is fine - we are setting the
return value*)
end;

begin
  c := 2;
  f^ := 'h';                 (*This compiles, but will crash at
runtime*)
end.
```

There two ways you can solve this kind of problem: either you change the definition of whatever you are assigning to, so the assignment becomes legal, or you eliminate the assignment.

66. Unsatisfied forward or external declaration: '<Procedurename>'

Complete list of compiler error messages

This error message appears when you have a forward or external declaration of a procedure or function, or a declaration of a method in a class or object type, and you don't define the procedure, function or method anywhere.

Maybe the definition is really missing, or maybe its name is just misspelled.

Note that a declaration of a procedure or function in the interface section of a unit is equivalent to a forward declaration - you have to supply the implementation (the body of the procedure or function) in the implementation section.

Similarly, the declaration of a method in a class or object type is equivalent to a forward declaration.

```
program Produce;

type
  TMyClass = class
  constructor Create;
  end;

function Sum(const a: array of Double): Double; forward;

function Summ(const a: array of Double): Double;
var
  i: Integer;
begin
  Result := 0.0;
  for i:= 0 to High(a) do
  Result := Result + a[i];
end;

begin
end.
```

The definition of Sum in the above example has an easy-to-spot typo.

```
program Solve;

type
  TMyClass = class
  constructor Create;
  end;

constructor TMyClass.Create;
begin
end;

function Sum(const a: array of Double): Double; forward;

function Sum(const a: array of Double): Double;
var
  i: Integer;
begin
  Result := 0.0;
  for i:= 0 to High(a) do
  Result := Result + a[i];
end;

begin
end.
```

The solution: make sure the definitions of your procedures, functions and methods are all there, and spelled correctly.

67. Missing operator or semicolon

Complete list of compiler error messages

This error message appears if there is no operator between two subexpressions, or no semicolon between two statements.

Often, a semicolon is missing on the previous line.

```
program Produce;
var
  I: Integer;
begin
  I := 1 2                  (*<-- Error message here*)
  if I = 3 then             (*<-- Error message here*)
  Writeln('Fine')
end.
```

The first statement in the example has two errors - a '+' operator and a semicolon are missing. The first error is reported on this statement, the second on the following line.

```
program Solve;
var
  I: Integer;
begin
  I := 1 + 2;               (*We were missing a '+' operator and a
semicolon*)
  if I = 3 then
  Writeln('Fine')
end.
```

The solution is to make sure the necessary operators and semicolons are there.

68. Missing parameter type

Complete list of compiler error messages

This error message is issued when a parameter list gives no type for a value parameter.

Leaving off the type is legal for constant and variable parameters.

```
program Produce;

procedure P(I;J: Integer);                                (*<-- Error
message here*)
begin
end;

function ComputeHash(Buffer; Size: Integer): Integer; (*<-- Error
message here*)
begin
end;


begin
end.
```

We intended procedure P to have two integer parameters, but we put a semicolon instead of a comma after the first parameters. The function ComputeHash was supposed to have an untyped first parameter, but untyped parameters must be either variable or constant parameters - they cannot be value parameters.

```
program Solve;

procedure P(I,J: Integer);
begin
end;

function ComputeHash(const Buffer; Size: Integer): Integer;
begin
end;

begin
end.
```

The solution in this case was to fix the type in P's parameter list, and to declare the Buffer parameter to ComputeHash as a constant parameter, because we don't intend to modify it.

69. Illegal reference to symbol '<Name>' in object file '<Filename>'

Complete list of compiler error messages

This error message is given if an object file loaded with a $L or $LINK directive contains a reference to a Pascal symbol that is not a procedure, function, variable, typed constant or thread local variable.

70. Line too long (more than 255 characters)

Complete list of compiler error messages

This error message is given when the length of a line in the source file exceeds 255 characters.

Usually, you can divide the long line into two shorter lines.

If you need a really long string constant, you can break it into several pieces on consecutive lines that you concatenate with the '+' operator.

71. Unknown directive: '<Directive>'

Complete list of compiler error messages

This error message appears when the compiler encounters an unknown directive in a procedure or function declaration.

The directive is probably misspelled, or a semicolon is missing.

```
program Produce;

procedure P; stcall;
begin
end;

procedure Q forward;

function GetLastError: Integer external 'kernel32.dll';

begin
end.
```

In the declaration of P, the calling convention "stdcall" is misspelled. In the declaration of Q and GetLastError, we're missing a semicolon.

```
program Solve;

procedure P; stdcall;
begin
end;

procedure Q; forward;

function GetLastError: Integer; external 'kernel32.dll';

begin
end.
```

The solution is to make sure the directives are spelled correctly, and that the necessary semicolons are there.

72. This type cannot be initialized

Complete list of compiler error messages

File types (including type Text), and the type Variant cannot be initialized, that is, you cannot declare typed constants or initialized variables of these types.

```
program Produce;

var
  V: Variant = 0;

begin
end.
```

The example tries to declare an initialized variable of type Variant, which illegal.

```
program Solve;

var
  V: Variant;

begin
  V := 0;
end.
```

The solution is to initialize a normal variable with an assignment statement.

73. Number of elements differs from declaration

Complete list of compiler error messages

This error message appears when you declare a typed constant or initialized variable of array type, but do not supply the appropriate number of elements.

```
program Produce;

var
  A : array [1..10] of Integer = (1,2,3,4,5,6,7,8,9);

begin
end.
```

The example declares an array of 10 elements, but the initialization only supplies 9 elements.

```
program Solve;

var
  A : array [1..10] of Integer = (1,2,3,4,5,6,7,8,9,10);

begin
end.
```

We just had to supply the missing element to make the compiler happy. When initializing bigger arrays, it can be sometimes hard to see whether you have supplied the right number of elements. To help with that, you layout the source file in a way that makes counting easy (e.g. ten elements to a line), or you can put the index of an element in comments next to the element itself.

74. Label already defined: '<Labelname>'

Complete list of compiler error messages

This error message is given when a label is set on more than one statement.

```
program Produce;
label 1;
begin
1:
  goto 1;
1:        (*<-- Error message here*)
end.
```

The example just tries to set label 1 twice.

```
program Solve;
label 1;
begin
1:
  goto 1;
end.
```

Make sure every label is set exactly once.

75. Label declared and referenced, but not set: '<label>'

Complete list of compiler error messages

You declared and used a label in your program, but the label definition was not encountered in the source code.

```
program Produce;

  procedure Labeled;
  label 10;
  begin
    goto 10;
  end;

begin
end.
```

Label 10 is declared and used in the procedure 'Labeled', but the compiler never finds a definition of the label.

```
program Produce;

  procedure Labeled;
  label 10;
  begin
    goto 10;
    10:
  end;

begin
end.
```

The simple solution is to ensure that a declared and used label has a definition, in the same scope, in your program.

76. This form of method call only allowed in methods of derived types

Complete list of compiler error messages

This error message is issued if you try to make a call to a method of an ancestor type, but you are in fact not in a method.

```
program Produce;

type
  TMyClass = class
    constructor Create;
  end;

procedure Create;
begin
  inherited Create;      (*<-- Error message here*)
end;

begin
end.
```

The example tries to call an inherited constructor in procedure Create, which is not a method.

```
program Solve;

type
  TMyClass = class
    constructor Create;
  end;

constructor TMyclass.Create;
begin
  inherited Create;
end;

begin
end.
```

The solution is to make sure you are in fact in a method when using this form of call.

77. This form of method call only allowed for class methods

Complete list of compiler error messages

You were trying to call a normal method by just supplying the class type, not an actual instance.

This is only allowed for class methods and constructors, not normal methods and destructors.

```
program Produce;

type
  TMyClass = class
  (*...*)
  end;
var
  MyClass: TMyClass;

begin
  MyClass := TMyClass.Create;  (*Fine, constructor*)
  Writeln(TMyClass.ClassName); (*Fine, class method*)
  TMyClass.Destroy;            (*<-- Error message here*)
end.
```

The example tries to destroy the type TMyClass - this doesn't make sense and is therefore illegal.

```
program Solve;
type
  TMyClass = class
  (*...*)
  end;
var
  MyClass: TMyClass;

begin
  MyClass := TMyClass.Create;  (*Fine, constructor*)
  Writeln(TMyClass.ClassName); (*Fine, class method*)
  MyClass.Destroy;             (*Fine, called on instance*)
end.
```

As you can see, we really meant to destroy the instance of the type, not the type itself.

78. Variable '<Name>' might not have been initialized

Complete list of compiler error messages

This warning is given if a variable has not been assigned a value on every code path leading to a point where it is used.

```
program Produce;
(*$WARNINGS ON*)
var
  B: Boolean;
  C: (Red,Green,Blue);

procedure Simple;
var
  I : Integer;
begin
  Writeln(I);        (*<-- Warning here*)
end;

procedure IfStatement;
var
  I : Integer;
begin
  if B then
    I := 42;
  Writeln(I);        (*<-- Warning here*)
end;

procedure CaseStatement;
var
  I: Integer;
begin
  case C of
  Red..Blue: I := 42;
  end;
  Writeln(I);        (*<-- Warning here*)
end;

procedure TryStatement;
var
  I: Integer;
begin
  try
    I := 42;
  except
    Writeln('Should not get here!');
  end;
  Writeln(I);        (*<-- Warning here*)
end;

begin
  B := False;
end.
```

In an if statement, you have to make sure the variable is assigned in both branches. In a case statement, you need to add an else part to make sure the variable is assigned a value in every conceivable case. In a try-except construct, the compiler assumes that assignments in the try

part may in fact not happen, even if they are at the very beginning of the try part and so simple that they cannot conceivably cause an exception.

```
program Solve;
(*$WARNINGS ON*)
var
  B: Boolean;
  C: (Red,Green,Blue);

procedure Simple;
var
  I : Integer;
begin
  I := 42;
  Writeln(I);
end;

procedure IfStatement;
var
  I : Integer;
begin
  if B then
    I := 42
  else
    I := 0;
  Writeln(I);        (*Need to assign I in the else part
end;

procedure CaseStatement;
var
  I: Integer;
begin
  case C of
  Red..Blue: I := 42;
  else       I := 0;
  end;
  Writeln(I);        (*Need to assign I in the else part*)
end;

procedure TryStatement;
var
  I: Integer;
begin
  I := 0;
  try
    I := 42;
  except
    Writeln('Should not get here!');
  end;
  Writeln(I);        (*Need to assign I before the try*)
end;

begin
  B := False;
end.
```

The solution is to either add assignments to the code paths where they were missing, or to add an assignment before a conditional statement or a try-except construct.

79. Value assigned to '<Name>' never used

Complete list of compiler error messages

The compiler gives this hint message if the value assigned to a variable is not used. If optimization is enabled, the assignment is eliminated.

This can happen because either the variable is not used anymore, or because it is reassigned before it is used.

```
program Produce;
(*$HINTS ON*)

procedure Simple;
var
  I: Integer;
begin
  I := 42;                (*<-- Hint message here*)
end;

procedure Propagate;
var
  I: Integer;
  K: Integer;
begin
  I := 0;                 (*<-- Hint message here*)
  Inc(I);                 (*<-- Hint message here*)
  K := 42;
  while K > 0 do begin
    if Odd(K) then
      Inc(I);             (*<-- Hint message here*)
    Dec(K);
  end;
end;

procedure TryFinally;
var
  I: Integer;
begin
  I := 0;                 (*<-- Hint message here*)
  try
    I := 42;
  finally
    Writeln('Reached finally');
  end;
  Writeln(I);             (*Will always write 42 - if an
exception happened,
          we wouldn't get here*)
end;

begin
end.
```

In procedure Propagate, the compiler is smart enough to realize that as variable I is not used after the while loop, it does not need to be incremented inside the while, and therefore the increment and the assignment before the while loop are also superfluous.

In procedure TryFinally, the assignment to I before the try-finally construct is not necessary. If an exception happens, we don't execute the Writeln statement at the end, so the value of I does not

matter. If no exception happens, the value of I seen by the Writeln statement is always 42. So the first assignment will not change the behavior of the procedure, and can therefore be eliminated.

This hint message does not indicate your program is wrong - it just means the compiler has determined there is an assignment that is not necessary.

You can usually just delete this assignment - it will be dropped in the compiled code anyway if you compile with optimizations on.

Sometimes, however, the real problem is that you assigned to the wrong variable, e.g. to meant to assign J but instead assigned I. So it is worthwhile to check the assignment in question carefully.

80. Return value of function '<Functionname>' might be undefined

Complete list of compiler error messages

This warning is given if the return value of a function has not been assigned a value on every code path.

To put it a little differently, the function could execute in a way that never assigns anything to the return value.

```
program Produce;
(*$WARNINGS ON*)
var
  B: Boolean;
  C: (Red,Green,Blue);

function Simple: Integer;
begin
end;                          (*<-- Warning here*)

function IfStatement: Integer;
begin
  if B then
    Result := 42;
end;                            (*<-- Warning here*)

function CaseStatement: Integer;
begin
  case C of
  Red..Blue: Result := 42;
  end;
end;                            (*<-- Warning here*)

function TryStatement: Integer;
begin
  try
    Result := 42;
  except
    Writeln('Should not get here!');
  end;
end;                            (*<-- Warning here*)

begin
  B := False;
end.
```

The problem with procedure IfStatement and CaseStatement is that the result is not assigned in every code path. In TryStatement, the compiler assumes that an exception could happen before Result is assigned.

```
program Solve;
(*$WARNINGS ON*)
var
  B: Boolean;
  C: (Red,Green,Blue);

function Simple: Integer;
begin
  Result := 42;
end;

function IfStatement: Integer;
begin
  if B then
    Result := 42
  else
    Result := 0;
end;

function CaseStatement: Integer;
begin
  case C of
  Red..Blue: Result := 42;
  else       Result := 0;
  end;
end;

function TryStatement: Integer;
begin
  Result := 0;
  try
    Result := 42;
  except
    Writeln('Should not get here!');
  end;
end;

begin
  B := False;
end.
```

The solution is to make sure there is an assignment to the result variable in every possible code path.

81. Procedure FAIL only allowed in constructor

Complete list of compiler error messages

The standard procedure Fail can only be called from within a constructor - it is illegal otherwise.

82. Procedure NEW needs constructor

Complete list of compiler error messages

This error message is issued when an identifier given in the parameter list to New is not a constructor.

```
program Produce;

type
  PMyObject = ^TMyObject;
  TMyObject = object
  F: Integer;
  constructor Init;
  destructor Done;
  end;

constructor TMyObject.Init;
begin
  F := 42;
end;

destructor TMyObject.Done;
begin
end;

var
  P: PMyObject;

begin
  New(P, Done);               (*<-- Error message here*)
end.
```

By mistake, we called New with the destructor, not the constructor.

```
program Solve;

type
  PMyObject = ^TMyObject;
  TMyObject = object
  F: Integer;
  constructor Init;
  destructor Done;
  end;

constructor TMyObject.Init;
begin
  F := 42;
end;

destructor TMyObject.Done;
begin
end;

var
  P: PMyObject;

begin
  New(P, Init);
end.
```

Make sure you give the New standard function a constructor, or no additional argument at all.

83. Procedure DISPOSE needs destructor

Complete list of compiler error messages

This error message is issued when an identifier given in the parameter list to Dispose is not a destructor.

```
program Produce;

type
  PMyObject = ^TMyObject;
  TMyObject = object
  F: Integer;
  constructor Init;
  destructor Done;
  end;

constructor TMyObject.Init;
begin
  F := 42;
end;

destructor TMyObject.Done;
begin
end;

var
  P: PMyObject;

begin
  New(P, Init);
  (*...*)
  Dispose(P, Init);        (*<-- Error message here*)
end.
```

In this example, we passed the constructor to Dispose by mistake.

```
program Solve;

type
  PMyObject = ^TMyObject;
  TMyObject = object
  F: Integer;
  constructor Init;
  destructor Done;
  end;

constructor TMyObject.Init;
begin
  F := 42;
end;

destructor TMyObject.Done;
begin
end;

var
  P: PMyObject;

begin
  New(P, Init);
  Dispose(P, Done);
end.
```

The solution is to either pass a destructor to Dispose, or to eliminate the second argument.

84. Assignment to FOR-Loop variable '<Name>'

Complete list of compiler error messages

It is illegal to assign a value to the for loop control variable inside the for loop.

If the purpose is to leave the loop prematurely, use a break or goto statement.

```
program Produce;

var
  I: Integer;
  A: array [0..99] of Integer;
begin
  for I := 0 to 99 do begin
    if A[I] = 42 then
      I := 99;
  end;
end.
```

In this case, the programmer thought that assigning 99 to I would cause the program to exit the loop.

```
program Solve;

var
  I: Integer;
  A: array [0..99] of Integer;
begin
  for I := 0 to 99 do begin
    if A[I] = 42 then
      Break;
  end;
end.
```

Using a break statement is a cleaner way to exit out of a for loop.

85. FOR-Loop variable '<Name>' may be undefined after loop

Complete list of compiler error messages

This warning is issued if the value of a for loop control variable is used after the loop.

You can only rely on the final value of a for loop control variable if the loop is left with a goto or exit statement.

The purpose of this restriction is to enable the compiler to generate efficient code for the for loop.

```
program Produce;
(*$WARNINGS ON*)

function Search(const A: array of Integer; Value: Integer):
Integer;
begin
  for Result := 0 to High(A) do
    if A[Result] = Value then
      break;
end;

const
  A : array [0..9] of Integer = (1,2,3,4,5,6,7,8,9,10);

begin
  Writeln( Search(A,11) );
end.
```

In the example, the Result variable is used implicitly after the loop, but it is undefined if we did not find the value - hence the warning.

```
program Solve;
(*$WARNINGS ON*)

function Search(const A: array of Integer; Value: Integer):
Integer;
begin
  for Result := 0 to High(A) do
    if A[Result] = Value then
      exit;
  Result := High(a)+1;
end;

const
  A : array [0..9] of Integer = (1,2,3,4,5,6,7,8,9,10);

begin
  Writeln( Search(A,11) );
end.
```

The solution is to assign the intended value to the control variable for the case where we don't exit the loop prematurely.

86. TYPEOF can only be applied to object types with a VMT

Complete list of compiler error messages

This error message is issued if you try to apply the standard function TypeOf to an object type that does not have a virtual method table.

A simple workaround is to declare a dummy virtual procedure to force the compiler to generate a VMT.

```
program Produce;

type
  TMyObject = object
    procedure MyProc;
  end;

procedure TMyObject.MyProc;
begin
  (*...*)
end;

var
  P: Pointer;
begin
  P := TypeOf(TMyObject);     (*<-- Error message here*)
end.
```

The example tries to apply the TypeOf standard function to type TMyObject which does not have virtual functions, and therefore no virtual function table (VMT).

```
program Solve;

type
  TMyObject = object
    procedure MyProc;
    procedure Dummy; virtual;
  end;

procedure TMyObject.MyProc;
begin
  (*...*)
end;

procedure TMyObject.Dummy;
begin
end;

var
  P: Pointer;
begin
  P := TypeOf(TMyObject);
end.
```

The solution is to introduce a dummy virtual function, or to eliminate the call to TypeOf.

87. Order of fields in record constant differs from declaration

Complete list of compiler error messages

This error message occurs if record fields in a typed constant or initialized variable are not initialized in declaration order.

```
program Produce;

type
  TPoint = record
    X, Y: Integer;
  end;

var
  Point : TPoint = (Y: 123; X: 456);

begin
end.
```

The example tries to initialize first Y, then X, in the opposite order from the declaration.

```
program Solve;

type
  TPoint = record
    X, Y: Integer;
  end;

var
  Point : TPoint = (X: 456; Y: 123);

begin
end.
```

The solution is to adjust the order of initialization to correspond to the declaration order.

88. Internal error: <ErrorCode>

Complete list of compiler error messages

You should never get this error message - it means there is a programming error in the compiler.

If you do, please call Inprise Developer Support and let us know the ErrorCode (e.g. "C1196") that appears in the error message. This will give us a rough indication what went wrong. It is even more helpful if you can give us an example program that produces this message.

89. Unit name mismatch: '<Unitname>'

Complete list of compiler error messages

This error message is issued if after loading a unit mentioned in a uses clause, the compiler finds that it does not have the requested name.

This can happen for instance because file names may be truncated to the MSDOS 8.3 filename format.

```
----- Contents of MY_UNIT_.PAS
-------------------------------------------
unit My_Unit_With_A_Long_Name;
interface
implementation
end.
----- End of MY_UNIT_.PAS
-----------------------------------------------

program Produce;
uses My_Unit_With_Another_Long_Name; (*Will find MY_UNIT_.PAS if
-P command line
               switch is active - but it's the wrong unit.*)
begin
end.
```

In this case, the problem is that the compiler found the wrong unit, because the filenames were truncated to 8 characters.

The solution is to use long filenames or to make sure the filenames differ in the first 8 characters. Also, you need to make sure the filename of a unit corresponds to the unit name.

90. Type '<Name>' is not yet completely defined

Complete list of compiler error messages

This error occurs if there is either a reference to a type that is just being defined, or if there is a forward declared class type in a type section and no final declaration of that type.

```
program Produce;

type
  TListEntry = record
    Next: ^TListEntry;                    (*<-- Error message
here*)
    Data: Integer;
  end;
  TMyClass = class;                       (*<-- Error message
here*)
  TMyClassRef = class of TMyClass;
  TMyClasss = class                       (*<-- Typo ...*)
    (*...*)
  end;

begin
end.
```

The example tries to refer to record type before it is completely defined. Also, because of a typo, the compiler never sees a complete declaration for TMyClass.

```
program Solve;

type
  PListEntry = ^TListEntry;
  TListEntry = record
    Next: PListEntry;
    Data: Integer;
  end;
  TMyClass = class;
  TMyClassRef = class of TMyClass;
  TMyClass = class
    (*...*)
  end;

begin
end.
```

The solution for the first problem is to introduce a type declaration for an auxiliary pointer type. The second problem is fixed by spelling TMyClass correctly.

91. This Demo Version has been patched

Complete list of compiler error messages

This error message is currently unused.

92. Integer constant or variable name expected

Complete list of compiler error messages

This error message is issued if you try to declare an absolute variable, but the absolute directive is not followed by an integer constant or a variable name.

```
program Produce;

var
  I : Integer;
  J : Integer absolute Addr(I);    (*<-- Error message here*)

begin
end.
```

```
program Solve;

const
  Addr = 0;

var
  I : Integer;
  J : Integer absolute I;

begin
end.
```

93. Invalid typecast

Complete list of compiler error messages

This error message is issued for type casts not allowed by the rules. The following kinds of casts are allowed:

- Ordinal or pointer type to another ordinal or pointer type

- A character, string, array of character or pchar to a string

- An ordinal, real, string or variant to a variant

- A variant to an ordinal, real, string or variant

- A variable reference to any type of the same size.

Note that casting real types to integer can be performed with the standard functions Trunc and Round.

There are other transfer functions like Ord and Chr that might make your intention clearer.

```
program Produce;

begin
  Writeln( Integer(Pi) );
end.
```

This programmer thought he could cast a floating point constant to Integer, like in C.

```
program Solve;

begin
  Writeln( Trunc(Pi) );
end.
```

In Pascal, we have separate Transfer functions to convert floating point values to integer.

94. User break - compilation aborted

Complete list of compiler error messages

This message is currently unused.

95. Assignment to typed constant '<Name>'

Complete list of compiler error messages

This warning message is currently unused.

96. "Segment/Offset pairs not supported in Borland 32-bit Pascal

Complete list of compiler error messages

32-bit code no longer uses the segment/offset addressing scheme that 16-bit code used.

In 16-bit versions of Borland Pascal, segment/offset pairs were used to declare absolute variables, and as arguments to the Ptr standard function.

Note that absolute addresses should not be used in 32-bit protected mode programs. Instead appropriate Win32 API functions should be called.

```
program Produce;

var
  VideoMode : Integer absolute $0040:$0049;

begin
  Writeln( Byte(Ptr($0040,$0049)^) );
end.
```

```
program Solve;
(*This version will compile, but will not run; absolute addresses
are to be carefully avoided*)
var
  VideoMode : Integer absolute $0040*16+$0049;

begin
  Writeln( Byte(Ptr($0040*16+$0049)^) );
end.
```

97. Program or unit '<name>' recursively uses itself

Complete list of compiler error messages

An attempt has been made for a unit to use itself.

```
unit Produce;
interface
  uses Produce;
implementation

begin
end.
```

In the above example the uses clause specifies the same unit, which causes the compiler to emit an error message.

```
unit Solve;
interface
implementation

begin
end.
```

The only solution to this problem is to remove the offending uses clause.

98. Label '<Name>' is not declared in current procedure

Complete list of compiler error messages

In contrast to Standard Pascal, Borland Pascal does not allow a goto to jump out of the current procedure.

However, his construct is mainly useful for error handling, and Borland Pascal provides a more general and structured mechanism to deal with errors: exception handling.

```
program Produce;

label 99;

procedure MyProc;
begin
  (*Something goes very wrong...*)
  goto 99;
end;

begin
  MyProc;
  99:
    Writeln('Fatal error');
end.
```

The example above tries to halt computation by doing a non-local goto.

```
program Solve;

uses SysUtils;

procedure MyProc;
begin
  (*Something goes very wrong...*)
  raise Exception.Create('Fatal error');
end;

begin
  try
    MyProc;
  except
    on E: Exception do Writeln(E.Message);
  end;
end.
```

In our solution, we used exception handling to stop the program. This has the advantage that we can also pass an error message. Another solution would be to use the standard procedures Halt or RunError.

99. "Local procedure/function '<Name>' assigned to procedure variable

Complete list of compiler error messages

This error message is issued if you try to assign a local procedure to a procedure variable, or pass it as a procedural parameter.

This is illegal, because the local procedure could then be called even if the enclosing procedure is not active. This situation would cause the program to crash if the local procedure tried to access any variables of the enclosing procedure.

```
program Produce;

var
  P: Procedure;

procedure Outer;

  procedure Local;
  begin
    Writeln('Local is executing');
  end;

begin
  P := Local;       (*<-- Error message here*)
end;

begin
  Outer;
  P;
end.
```

The example tries to assign a local procedure to a procedure variable. This is illegal because it is unsafe at run time.

```
program Solve;

var
  P: Procedure;

procedure NonLocal;
begin
  Writeln('NonLocal is executing');
end;

procedure Outer;

begin
  P := NonLocal;
end;

begin
  Outer;
  P;
end.
```

The solution is to move the local procedure out of the enclosing one.

100. Missing ENDIF directive

Complete list of compiler error messages

This error message is issued if the compiler does not find a corresponding $ENDIF directive after an $IFDEF, $IFNDEF or $IFOPT directive.

```
program Produce;
(*$APPTYPE CONSOLE*)
begin
(*$IfOpt O+*)
  Writeln('Compiled with optimizations');
(*$Else*)
  Writeln('Compiled without optimizations');
(*Endif*)
end.                                          (*<-- Error
message here*)
```

In this example, we left out the $ character in the (*$Endif*) directive, so the compiler mistook it for a comment.

```
program Solve;
(*$APPTYPE CONSOLE*)
begin
(*$IfOpt O+*)
  Writeln('Compiled with optimizations');
(*$Else*)
  Writeln('Compiled without optimizations');
(*$Endif*)
end.
```

The solution is to make sure all the conditional directives have a valid $ENDIF directive.

101. Method identifier expected

Complete list of compiler error messages

This error message will be issued in several different situations:

- Properties in an automated section must use methods for access, they cannot use fields in their read or write clauses.

- You tried to call a class method with the "ClassType.MethodName" syntax, but "MethodName" was not the name of a method.

- You tried calling an inherited with the "Inherited MethodName" syntax, but "MethodName" was not the name of a method.

```
program Produce;

type
  TMyBase = class
    Field: Integer;
  end;
  TMyDerived = class (TMyBase)
    Field: Integer;
    function Get: Integer;
  Automated
    property Prop: Integer read Field;     (*<-- Error message
here*)
  end;

function TMyDerived.Get: Integer;
begin
  Result := TMyBase.Field;                 (*<-- Error message
here*)
end;

begin
end.
```

The example tried to declare an automated property that accesses a field directly. The second error was caused by trying to get at a field of the base class - this is also not legal.

```
program Solve;

type
  TMyBase = class
    Field: Integer;
  end;
  TMyDerived = class (TMyBase)
    Field: Integer;
    function Get: Integer;
  Automated
    property Prop: Integer read Get;
  end;

function TMyDerived.Get: Integer;
begin
  Result := TMyBase(Self).Field;
end;

begin
  Writeln( TMyDerived.Create.Prop );
end.
```

The first problem is fixed by accessing the field via a method. The second problem can be fixed by casting the Self pointer to the base class type, and accessing the field off of that.

102. FOR-Loop variable '<name>' cannot be passed as var parameter

Complete list of compiler error messages

An attempt has been made to pass the control variable of a FOR-loop to a procedure or function which takes a var parameter. This is an error because the procedure which receives the control variable is able to modify it, thereby changing the semantics of the FOR-loop which issued the call.

```
program Produce;

  procedure p1(var x : Integer);
  begin
  end;

  procedure p0;
    var
      i : Integer;
  begin
    for i := 0 to 1000 do
      p1(i);
  end;

begin
end.
```

In this example, the loop control variable, i, is passed to a procedure which receives a var parameter. This is the main cause of the error.

```
program Solve;
  procedure p1(x : Integer);
  begin
  end;

  procedure p0;
    var
      i : Integer;
  begin
    i := 0;
    while i <= 1000 do
      p1(i);
  end;


begin
end.
```

The easiest way to approach this problem is to change the parameter into a by-value parameter. However, there may be a good reason that it was a by-reference parameter in the begging, so you must be sure that this change of semantics in your program does not affect other code. Another way to approach this problem is change the for loop into an equivalent while loop, as is done in the above program.

103. Typed constant '<name>' passed as var parameter

Complete list of compiler error messages

This error message is reserved.

104. BREAK or CONTINUE outside of loop

Complete list of compiler error messages

The compiler has found a BREAK or CONTINUE statement which is not contained inside a WHILE or REPEAT loop. These two constructs are only legal in loops.

```
program Produce;

  procedure Error;
    var i : Integer;
  begin
    i := 0;
    while i < 100 do
      INC(i);
      if odd(i) then begin
        INC(i);
  continue;
      end;
  end;

begin
end.
```

The example above shows how a continue statement could seem to be included in the body of a looping construct but, due to the compound-statement nature of Pascal, it really is not.

```
program Solve;

  procedure Error;
    var i : Integer;
  begin
    i := 0;
    while i < 100 do begin
      INC(i);
      if odd(i) then begin
        INC(i);
  continue;
      end;
    end;
  end;

begin
end.
```

Often times it is a simple matter to create compound statement out of the looping construct to ensure that your CONTINUE or BREAK statements are included.

105. Division by zero

Complete list of compiler error messages

The compiler has detected a constant division by zero in your program.

Check your constant expressions and respecify them so that a division by zero error will not occur.

106. Overflow in conversion or arithmetic operation

Complete list of compiler error messages

The compiler has detected an overflow in an arithmetic expression: the result of the expression is too large to be represented in 32 bits.

Check your computations to ensure that the value can be represented by the computer hardware.

107. Data type too large: exceeds 2 GB

Complete list of compiler error messages

You have specified a data type which is too large for the compiler to represent. The compiler will generate this error for datatypes which are greater or equal to 2 GB in size. You must decrease the size of the description of the type.

```
program Produce;

  type
    EnormousArray = array [0..MaxLongint] OF Longint;
    BigRecord = record
      points : array [1..10000] of Extended;
    end;

  var
    data : array [0..500000] of BigRecord;

begin
end.
```

It is easily apparent to see why these declarations will elicit error messages.

```
program Solve;
  type
    EnormousArray = array [0..MaxLongint DIV 8] OF Longint;

    DataPoints = ^DataPointDesc;
    DataPointDesc = array [1..10000] of Extended;
    BigRecord = record
      points : DataPoints;
    end;

  var
    data : array [0..500000] OF BigRecord;


begin
end.
```

The easy solution to avoid this error message is to make sure that the size of your data types remain under 2Gb in size. A more complicated method would involve the restructuring of your data, as has been begun with the BigRecord declaration.

108. Integer constant too large

Complete list of compiler error messages

You have specified an integer constant that requires more than 64 bits to represent.

```
program Produce;

  const
    VeryBigHex = $80000000000000001;

begin
end.
```

The constant in the above example is too large to represent in 64 bits, thus the compiler will output an error.

```
program Solve;

  const
    BigHex = $8000000000000001;

begin
end.
```

Check the constants that you have specified and ensure that they are representable in 64 bits.

109. 16-Bit fixup encountered in object file '<name>'

Complete list of compiler error messages

A 16-bit fixup has been found in one of the object modules linked to your program with the $L compiler directive. The compiler only supports 32 bit fixups in linked object modules.

Make sure that the linked object module is a 32 bit object module.

110. Inline assembler syntax error

Complete list of compiler error messages

You have entered an expression which the inline assembler is unable to interpret as a valid assembly instruction.

```
program Produce;

  procedure Assembly;
  asm
    adx  eax, 151
  end;

begin
end.
```

```
program Solve;

  procedure Assembly;
  asm
    add  eax, 151
  end;

begin
end.
```

Examine the offending inline assembly statement and ensure that it conforms to the proper syntax.

111. Inline assembler stack overflow

Complete list of compiler error messages

Your inline assembler code has exceeded the capacity of the inline assembler.

Contact Inprise if you encounter this error.

112. Operand size mismatch

Complete list of compiler error messages

The size required by the instruction operand does not match the size given.

```
program Produce;

  var
    v : Integer;

  procedure Assembly;
  asm
    db offset v
  end;

begin
end.
```

In the sample above, the compiler will complain because the 'offset' operator produces a 'dword', but the operator is expecting a 'byte'.

```
program Solve;

  var
    v : Integer;

  procedure Assembly;
  asm
    dd offset v
  end;

begin
end.
```

The solution, for this example, is to change the operator to receive a 'dword'. In the general case you will need to closely examine your code and ensure that the operator and operand sizes match.

113. Memory reference expected

Complete list of compiler error messages

The inline assembler has expected to find a memory reference expression but did not find one.

Ensure that the offending statement is indeed a memory reference.

114. Constant expected

Complete list of compiler error messages

The inline assembler was expecting to find a constant but did not find one.

```
program Produce;

  procedure Assembly(x : Integer);
  asm
    mov   ax, x MOD 10
  end;

begin
end.
```

The inline assembler is not capable of performing a MOD operation on a Pascal variable, thus the above code will cause an error.

Many of the inline assembler expressions require constants to assemble correctly. Change the offending statement to have a assemble-time constant.

115. Type expected

Complete list of compiler error messages

Contact Inprise if you receive this error.

116. Cannot add or subtract relocatable symbols

Complete list of compiler error messages

The inline assembler is not able to add or subtract memory address which may be changed by the linker.

```
program Produce;

  var
    a : array [1..10] of Integer;
    endOfA : Integer;

  procedure Relocatable;
  begin
  end;

  procedure Assembly;
  asm
    mov eax, a + endOfA
  end;

begin
end.
```

Global variables fall into the class of items which produce relocatable addresses, and the inline assembler is unable to add or subtract these.

Make sure you don't try to add or subtract relocatable addresses from within your inline assembler statements.

117. Invalid register combination

Complete list of compiler error messages

You have specified an illegal combination of registers in a inline assembler Please refer to an assembly language guide for more information on addressing modes allowed on the Intel 80x86 family. statement.

```
program Produce;

  procedure AssemblerExample;
  asm
    mov eax, [ecx + esp * 4]
  end;

begin
end.
```

The right operand specified in this mov instruction is illegal.

```
program Solve;

  procedure AssemblerExample;
  asm
    mov eax, [ecx + ebx * 4]
  end;

begin
end.
```

The addressing mode specified by the right operand of this mov instruction is allowed.

118. Numeric overflow

Complete list of compiler error messages

The inline assembler has detected a numeric overflow in one of your expressions.

```
program Produce;

  procedure AssemblerExample;
  asm
    mov eax, $0fffffffffffffffffffff
  end;

begin
end.
```

Specifying a number which requires more than 32bits to represent will elicit this error.

```
program Solve;

  procedure AssemblerExample;
  asm
    mov al, $0ff
  end;

begin
end.
```

Make sure that your numbers all fit in 32bits.

119. String constant too long

Complete list of compiler error messages

The inline assembler has not found the end of the string that you specified. The most likely cause is a misplaced closing quote.

```
program Produce;

  procedure AssemblerExample;
  asm
    db 'Hello world.  I am an inline assembler statement
  end;

begin
end.
```

The inline assembler is unable to find the end of the string, before the end of the line, so it reports that the string is too long.

```
program Solve;

  procedure AssemblerExample;
  asm
    db 'Hello world.  I am an inline assembler statement'
  end;

begin
end.
```

Adding the closing quote will vanquish this error.

120. Error in numeric constant

Complete list of compiler error messages

The inline assembler has found an error in the numeric constant you entered.

```
program Produce;

  procedure AssemblerExample;
  asm
    mov al, $z0f0
  end;

begin
end.
```

In the example above, the inline assembler was expecting to parse a hexadecimal constant, but it found an erroneous character.

```
program Solve;

  procedure AssemblerExample;
  asm
    mov al, $f0
  end;

begin
end.
```

Make sure that the numeric constants you enter conform to the type that the inline assembler is expecting to parse.

121. Invalid combination of opcode and operands

Complete list of compiler error messages

You have specified an inline assembler statement which is not correct.

```
program Produce;

  procedure AssemblerExample;
  asm
    mov al, $0f0 * 16
  end;

begin
end.
```

The inline assembler is not capable of storing the result of $f0 * 16 into the 'al' register—it simply won't fit.

```
program Solve;
  procedure AssemblerExample;
  asm
    mov al, $0f * 16
  end;

begin
end.
```

Make sure that the type of both operands are compatible.

122. 486/487 instructions not enabled

Complete list of compiler error messages

You should not receive this error as 486 instructions are always enabled.

123. Division by zero

Complete list of compiler error messages

The inline assembler has encountered an expression which results in a division by zero.

```
program Produce;

  procedure AssemblerExample;
  asm
    dw  1000 / 0
  end;

begin
end.
```

If you are using program constants instead of constant literals this error might not be quite so obvious.

```
program Solve;

  procedure AssemblerExample;
  asm
    dw  1000 / 10
  end;

begin
end.
```

The solution, as when programming in high level languages, is to make sure that you don't divide by zero.

124. Structure field identifier expected

Complete list of compiler error messages

The inline assembler recognized an identifier on the right side of a '.', but it was not a field of the record found on the left side of the '.'. One common, yet difficult to realize, error of this sort is to use a record with a field called 'ch' - the inline assembler will always interpret 'ch' to be a register name.

```
program Produce;

  type
    Data = record
      x : Integer;
    end;

  procedure AssemblerExample(d : Data; y : Char);
  asm
    mov  eax, d.y
  end;

begin
end.
```

In this example, the inline assembler has recognized that 'y' is a valid identifier, but it has not found 'y' to be a member of the type of 'd'.

```
program Solve;

  type
    Data = record
      x : Integer;
    end;

  procedure AssemblerExample(d : Data; y : Char);
  asm
    mov  eax, d.x
  end;

begin
end.
```

By specifying the proper variable name, the error will go away.

125. LOOP/JCXZ distance out of range

Complete list of compiler error messages

You have specified a LOOP or JCXZ destination which is out of range. You should not receive this error as the jump range is 2Gb for LOOP and JCXZ instructions.

126. Procedure or function name expected

Complete list of compiler error messages

You have specified an identifier which does not represent a procedure or function in an EXPORTS clause.

```
library Produce;

  var
   y : procedure;

exports y;
begin
end.
```

It is not possible to export variables from a Delphi library, even though the variable is of 'procedure' type.

```
program Solve;

  procedure ExportMe;
  begin
  end;

exports ExportMe;
begin
end.
```

Always be sure that all the identifiers listed in an EXPORTS clause truly represent procedures.

127. PROCEDURE or FUNCTION expected

Complete list of compiler error messages

This error message is produced by two different constructs, but in both cases the compiler is expecting to find the keyword 'procedure' or the keyword 'function'.

```
program Produce;

  type
    Base = class
      class AProcedure; (*case 1*)
    end;

  class Base.AProcedure; (*case 2*)
  begin
  end;

begin
end.
```

In both cases above, the word 'procedure' should follow the keyword 'class'.

```
program Solve;

  type
    Base = class
      class procedure AProcedure;
    end;

  class procedure Base.AProcedure;
  begin
  end;

begin
end.
```

As can be seen, adding the keyword 'procedure' removes the error from this program.

128. Instance variable '<name>' inaccessible here

Complete list of compiler error messages

You are attempting to reference a instance variable from within a class procedure.

```
program Produce;

  type
    Base = class
      Title : String;

      class procedure Init;
    end;

  class procedure Base.Init;
  begin
    Self.Title := 'Does not work';
    Title := 'Does not work';
  end;

begin
end.
```

Class procedures do not have an instance pointer, so they cannot access any methods or instance data of the class.

```
program Solve;

  type
    Base = class
      Title : String;

      class procedure Init;
    end;

  class procedure Base.Init;
  begin
  end;

begin
end.
```

The only solution to this error is to not access any member data or methods from within a class method.

129. EXCEPT or FINALLY expected

Complete list of compiler error messages

The compiler was expecting to find a FINALLY or EXCEPT keyword, during the processing of exception handling code, but did not find either.

```
program Produce;

begin
  try
  end;
end.
```

In the code above, the 'except' or 'finally' clause of the exception handling code is missing, so the compiler will issue an error.

```
program Solve;

begin
  try
  except
  end;
end.
```

By adding the missing clause, the compiler will be able to complete the compilation of the code. In this case, the 'except' clause will easily allow the program to finish.

130. Cannot BREAK, CONTINUE or EXIT out of a FINALLY clause

Complete list of compiler error messages

Because a FINALLY clause may be entered and exited through Delphi's exception handling mechanism or through normal program control, the explicit control flow of your program may not be followed. When the FINALLY is entered through the exception handling mechanism, it is not possible to exit the clause with BREAK, CONTINUE, or EXIT - when the finally clause is being executed by the exception handling system, control must return to the exception handling system.

```
program Produce;

  procedure A0;
  begin
    try
      (* try something that might fail *)
    finally
      break;
    end;
  end;

  begin
  end.
```

The program above attempts to exit the finally clause with a break statement. It is not legal to exit a FINALLY clause in this manner.

```
program Solve;

  procedure A0;
  begin
    try
      (* try something that might fail *)
    finally
    end;
  end;

  begin
  end.
```

The only solution to this error is to restructure your code so that the offending statement does not appear in the FINALLY clause.

131. 'GOTO <label>' leads into or out of TRY statement

Complete list of compiler error messages

The GOTO statement cannot jump into or out of an exception handling statement.

```
program Produce;

label 1, 2;

begin
  goto 1;
  try
1:
  except
    goto 2;
  end;
2:
end.
```

Both GOTO statements in the above code are incorrect. It is not possible to jump into, or out of, exception handling blocks.

The ideal solution to this problem is to avoid using GOTO statements altogether, however, if that is not possible you will have to perform more detailed analysis of the program to determine the correct course of action.

132. <clause1> clause expected, but <clause2> found

Complete list of compiler error messages

The compiler was, due to the Pascal syntax, expecting to find a clause1 in your program, but instead found clause2.

```
program Produce;

  type
    CharDesc = class
      vch : Char;

property Ch : Char;
    end;
end.
```

The first declaration of a property must specify a read and write clause, and since both are missing on the 'Ch' property, an error will result when compiling. In the case of properties, the original intention might have been to hoist a property defined in a base class to another visibility level - for example, from public to private. In this case, the most probable cause of the error is that the property name was not found in the base class. Make sure that you have spelled the property name correctly and that it is actually contained in one of the parent classes.

```
program Produce;

  type
    CharDesc = class
      vch : Char;

property Ch : Char read vch write vch;
    end;
end.
```

The solution is to ensure that all the proper clauses are specified, where required.

133. Cannot assign to a read-only property

Complete list of compiler error messages

The property to which you are attempting to assign a value did not specify a 'write' clause, thereby causing it to be a read-only property.

```
program Produce;

  type
    Base = class
      s : String;

      property Title : String read s;
    end;

  var
    c : Base;

  procedure DiddleTitle
  begin
    if c.Title = '' then
      c.Title := 'Super Galactic Invaders with Turbo Gungla
Sticks';

      (*perform other work on the c.Title*)
  end;

begin
end.
```

If a property does not specify a 'write' clause, it effectively becomes a read-only property; it is not possible to assign a value to a property which is read-only, thus the compiler outputs an error on the assignment to 'c.Title'.

```
program Solve;

  type
    Base = class
      s : String;

      property Title : String read s;
    end;

  var
    c : Base;

  procedure DiddleTitle
    var title : String;
  begin
    title := c.Title;
    if Title = '' then
      Title := 'Super Galactic Invaders with Turbo Gungla
Sticks';
      (*perform other work on title*)
  end;

begin
end.
```

One solution, if you have source code, is to provide a write clause for the read-only property - of course, this could dramatically alter the semantics of the base class and should not be taken lightly. Another alternative would be to introduce an intermediate variable which would contain the value of the read-only property - it is this second alternative which is shown in the code above.

134. Cannot read a write-only property

Complete list of compiler error messages

The property from which you are attempting to read a value did not specify a 'read' clause, thereby causing it to be a write-only property.

```
program Produce;

  type
    Base = class
      s : String;

      property Password : String write s;
    end;

  var
    c : Base;
    s : String;

begin
  s := c.Password;
end.
```

Since c.Password has not specified a read clause, it is not possible to read its value.

```
program Solve;

  type
    Base = class
      s : String;

      property Password : String read s write s;
    end;

  var
    c : Base;
    s : String;

begin
  s := c.Password;
end.
```

One easy solution to this problem, if you have source code, would be to add a read clause to the write-only property. But, adding a read clause is not always desirable and could lead to holes in a security system - consider, for example, a write-only property called 'Password', as in this example: you certainly wouldn't want to casually allow programs using this class to read the stored password. If a property was created as write-only, there is probably a good reason for it and you should reexamine why you need to read this property.

135. Class already has a default property

Complete list of compiler error messages

You have tried to assign a default property to a class which already has defined a default property.

```
program Produce;

  type
    Base = class
      function GetV(i : Integer) : Char;
      procedure SetV(i : Integer; const x : Char);

      property Data[i : Integer] : Char read GetV write SetV;
default;
      property Access[i : Integer] : Char read GetV write SetV;
default;
    end;

  function Base.GetV(i : Integer) : Char;
  begin GetV := 'A';
  end;

  procedure Base.SetV(i : Integer; const x : Char);
  begin
  end;

begin
end.
```

The Access property in the code above attempts to become the default property of the class, but Data has already been specified as the default. There can be only one default property in a class.

```
program Solve;

  type
    Base = class
      function GetV(i : Integer) : Char;
      procedure SetV(i : Integer; const x : Char);

      property Data[i : Integer] : Char read GetV write SetV;
default;
    end;

  function Base.GetV(i : Integer) : Char;
  begin GetV := 'A';
  end;

  procedure Base.SetV(i : Integer; const x : Char);
  begin
  end;

begin
end.
```

The solution is to remove the incorrect default property specifications from the program source.

136. Default property must be an array property

Complete list of compiler error messages

The default property which you have specified for the class is not an array property. Default properties are required to be array properties.

```
program Produce;

  type
    Base = class
      function GetV : Char;
      procedure SetV(x : Char);

      property Data : Char read GetV write SetV; default;
    end;

  function Base.GetV : Char;
  begin GetV := 'A';
  end;

  procedure Base.SetV(x : Char);
  begin
  end;

begin
end.
```

When specifying a default property, you must make sure that it conforms to the array property syntax. The 'Data' property in the above code specifies a 'Char' type rather than an array.

```
program Solve;

  type
    Base = class
      function GetV(i : Integer) : Char;
      procedure SetV(i : Integer; const x : Char);

      property Data[i : Integer] : Char read GetV write SetV;
default;
    end;

  function Base.GetV(i : Integer) : Char;
  begin GetV := 'A';
  end;

  procedure Base.SetV(i : Integer; const x : Char);
  begin
  end;

begin
end.
```

By changing the specification of the offending property to an array, or by removing the 'default' directive, you can remove this error.

137. TYPEINFO standard function expects a type identifier

Complete list of compiler error messages

You have attempted to obtain type information for an identifier which does not represent a type.

```
program Produce;

  var
    p : Pointer;

  procedure NotType;
  begin
  end;


begin
  p := TypeInfo(NotType);
end.
```

The TypeInfo standard procedure requires a type identifier as it's parameter. In the code above, 'NotType' does not represent a type identifier.

```
program Solve;

  type
    Base = class
    end;

  var
    p : Pointer;

begin
  p := TypeInfo(Base);
end.
```

By ensuring that the parameter used for TypeInfo is a type identifier, you will avoid this error.

138. Type '<name>' has no type info

Complete list of compiler error messages

You have applied the TypeInfo standard procedure to a type identifier which does not have any run-time type information associated with it.

```
program Produce;

  type
    Data = record
    end;

  var
    v : Pointer;

begin
  v := TypeInfo(Data);
end.
```

Record types do not generate type information, so this use of TypeInfo is illegal.

```
program Solve;

  type
    Base = class
    end;

  var
    v : Pointer;

begin
  v := TypeInfo(Base);
end.
```

A class does generate RTTI, so the use of TypeInfo here is perfectly legal.

139. FOR or WHILE loop executes zero times - deleted

Complete list of compiler error messages

The compiler has determined that the specified looping structure will not ever execute, so as an optimization it will remove it. Example:

```
program Produce;
(*$HINTS ON*)

  var
    i : Integer;

begin
  i := 0;
  WHILE FALSE AND (i < 100) DO
    INC(i);
end.
```

The compiler determines that 'FALSE AND (i < 100)' always evaluates to FALSE, and then easily determines that the loop will not be executed.

```
program Solve;
(*$HINTS ON*)

  var
    i : Integer;

begin
  i := 0;
  WHILE i < 100 DO
    INC(i);
end.
```

The solution to this hint is to check the boolean expression used to control while statements is not always FALSE. In the for loops you should make sure that (upper bound - lower bound) >= 1.

You may see this warning if a FOR loop increments its control variable from a value within the range of Longint to a value outside the range of Longint. For example:

```
var I: Cardinal;
begin
  For I := 0 to $FFFFFFFF do
...
```

This results from a limitation in the compiler which you can work around by replacing the FOR loop with a WHILE loop.

140. No definition for abstract method '<name>' allowed

Complete list of compiler error messages

You have declared <name> to be abstract, but the compiler has found a definition for the method in the source file. It is illegal to provide a definition for an abstract declaration.

```
program Produce;

  type
    Base = class
      procedure Foundation; virtual; abstract;
    end;

    procedure Base.Foundation;
    begin
    end;

begin
end.
```

Abstract methods cannot be defined. An error will appear at the point of Base.Foundation when you compile this program.

```
program Solve;

  type
    Base = class
      procedure Foundation; virtual; abstract;
    end;

    Derived = class (Base)
      procedure Foundation; override;
    end;

    procedure Derived.Foundation;
    begin
    end;

begin
end.
```

Two steps are required to solve this error. First, you must remove the definition of the abstract procedure which is declared in the base class. Second, you must extend the base class, declare the abstract procedure as an 'override' in the extension, and then provide a definition for the newly declared procedure.

141. Method '<name>' not found in base class

Complete list of compiler error messages

You have applied the 'override' directive to a method, but the compiler is unable to find a procedure of the same name in the base class.

```
program Produce;

  type
    Base = class
      procedure Title; virtual;
    end;

    Derived = class (Base)
      procedure Titl; override;
    end;

    procedure Base.Title;
    begin
    end;

    procedure Derived.Titl;
    begin
    end;

begin
end.
```

A common cause of this error is a simple typographical error in your source code. Make sure that the name used as the 'override' procedure is spelled the same as it is in the base class. In other situations, the base class will not provide the desired procedure: it is those situations which will require much deeper analysis to determine how to solve the problem.

```
program Solve;

  type
    Base = class
      procedure Title; virtual;
    end;

    Derived = class (Base)
      procedure Title; override;
    end;

    procedure Base.Title;
    begin
    end;

    procedure Derived.Title;
    begin
    end;

begin
end.
```

The solution (in this example) was to correct the spelling of the procedure name in Derived.

142. Invalid message parameter list

Complete list of compiler error messages

A message procedure can take only one, VAR, parameter; it's type is not checked.

```
program Produce;

  type
    Base = class
      procedure Msg1(x : Integer); message 151;
      procedure Msg2(VAR x, y : Integer); message 152;
    end;

  procedure Base.Msg1(x : Integer);
  begin
  end;

  procedure Base.Msg2(VAR x, y : Integer);
  begin
  end;

begin
end.
```

The obvious error in the first case is that the parameter is not VAR. The error in the second case is that more than one parameter is declared.

```
program Solve;

  type
    Base = class
      procedure Msg1(VAR x : Integer); message 151;
      procedure Msg2(VAR y : Integer); message 152;
    end;

  procedure Base.Msg1(VAR x : Integer);
  begin
  end;

  procedure Base.Msg2(VAR y : Integer);
  begin
  end;

begin
end.
```

The solution in both cases was to only specify one, VAR, parameter in the message method declaration.

143. Illegal message method index

Complete list of compiler error messages

You have specified value for your message index which <= 0.

```
program Produce;

  type
    Base = class
      procedure Dynamo(VAR x : Integer); message -151;
    end;

  procedure Base.Dynamo(VAR x : Integer);
  begin
  end;

begin
end.
```

The specification of -151 as the message index is illegal in the above example.

```
program Solve;

  type
    Base = class
      procedure Dynamo(VAR x : Integer); message 151;
    end;

  procedure Base.Dynamo(VAR x : Integer);
  begin
  end;

begin
end.
```

Always make sure that your message index values are >= 1.

144. Duplicate dynamic method index

Complete list of compiler error messages

You have specified an index for a dynamic method which is already used by another dynamic method.

```
program Produce;

  type
    Base = class
      procedure First(VAR x : Integer); message 151;
      procedure Second(VAR x : Integer); message 151;
    end;

  procedure Base.First(VAR x : Integer);
  begin
  end;

  procedure Base.Second(VAR x : Integer);
  begin
  end;

begin
end.
```

The declaration of 'Second' attempts to reuse the same message index which is used by 'First'; this is illegal.

```
program Solve;

  type
    Base = class
      procedure First(VAR x : Integer); message 151;
      procedure Second(VAR x : Integer); message 152; (*change to
unique index*)
    end;

    Derived = class (Base)
      procedure First(VAR x : Integer); override; (*override base
class behavior*)
    end;

  procedure Base.First(VAR x : Integer);
  begin
  end;

  procedure Base.Second(VAR x : Integer);
  begin
  end;

  procedure Derived.First(VAR x : Integer);
  begin
  end;

begin
end.
```

There are two straightforward solutions to this problem. First, if you really do not need to use the

same message value, you can change the message number to be unique. Alternatively, you could derive a new class from the base and override the behavior of the message handler declared in the base class. Both options are shown in the above example.

145. Bad file format '<name>'

Complete list of compiler error messages

The compiler state file has become corrupted. It is not possible to reload the previous compiler state.

Delete the corrupt file.

146. Inaccessible value

Complete list of compiler error messages

You have tried to view a value that is not accessible from within the integrated debugger. Certain types of values, such as a 0 length Variant-type string, cannot be viewed within the debugger.

147. Destination cannot be assigned to

Complete list of compiler error messages

The integrated debugger has determined that your assignment is not valid in the current context.

148. Expression has no value

Complete list of compiler error messages

You have attempted to assign the result of an expression, which did not produce a value, to a variable.

149. Destination is inaccessible

Complete list of compiler error messages

The address to which you are attempting to put a value is inaccessible from within the IDE.

150. Re-raising an exception only allowed in exception handler

Complete list of compiler error messages

You have used the syntax of the raise statement which is used to reraise an exception, but the compiler has determined that this reraise has occurred outside of an exception handler block. A limitation of the current exception handling mechanism disallows reraising exceptions from nested exception handlers. for the exception.

```
program Produce;

  procedure RaiseException;
  begin
    raise;            (*case 1*)
    try
      raise;          (*case 2*)
    except
      try
        raise;        (*case 3*)
      except
      end;
      raise;
    end;
  end;


begin
end.
```

There are several reasons why this error might occur. First, you might have specified a raise with no exception constructor outside of an exception handler. Secondly, you might be attempting to reraise an exception in the try block of an exception handler. Thirdly, you might be attempting to reraise the exception in an exception handler nested in another exception handler.

```
program Solve;
  uses SysUtils;

  procedure RaiseException;
  begin
    raise Exception.Create('case 1');
    try
      raise Exception.Create('case 2');
    except
      try
        raise Exception.Create('case 3');
      except
      end;
      raise;
    end;
  end;

begin
end.
```

One solution to this error is to explicitly raise a new exception; this is probably the intention in situations like 'case 1' and 'case 2'. For the situation of 'case 3', you will have to examine your code to determine a suitable workaround which will provide the desired results.

151. Default values must be of ordinal, pointer or small set type

Complete list of compiler error messages

You have declared a property containing a default clause, but the type property type is incompatible with default values.

```
program Produce;

  type
    VisualGauge = class
      pos : Single;
property Position : Single read pos write pos default 0.0;
    end;

begin
end.
```

The program above creates a property and attempts to assign a default value to it, but since the type of the property does not allow default values, an error is output.

```
program Produce;

  type
    VisualGauge = class
      pos : Integer;
property Position : Integer read pos write pos default 0;
    end;

begin
end.
```

When this error is encountered, there are two easy solutions: the first is to remove the default value definition, and the second is to change the type of the property to one which allows a default value. Your program, however, may not be as simple to fix; consider when you have a set property which is too large - it is this case which will require you to carefully examine your program to determine the best solution to this problem.

152. Property '<name>' does not exist in base class

Complete list of compiler error messages

The compiler believes you are attempting to hoist a property to a different visibility level in a derived class, but the specified property does not exist in the base class.

```
program Produce;

  type
    Base = class
    private
      a : Integer;
      property BaseProp : integer read a write a;
    end;

    Derived = class (Base)
      ch : Char;
      property Alpha read ch write ch; (*case 1*)
      property BesaProp; (*case 2*)
    end;

begin
end.
```

There are two basic causes of this error. The first is the specification of a new property without specifying a type; this usually is not supposed to be a movement to a new visibility level. The second is the specification of a property which should exist in the base class, but is not found by the compiler; the most likely cause for this is a simple typo (as in "BesaProp"). In the second form, the compiler will also output errors that a read or write clause was expected. of a proper

```
program Solve;

  type
    Base = class
    private
      a : Integer;
      property BaseProp : integer read a write a;
    end;

    Derived = class (Base)
      ch : Char;
    public
      property Alpha : Char read ch write ch; (*case 1*)
      property BaseProp; (*case 2*)
    end;

begin
end.
```

The solution for the first case is to supply the type of the property. The solution for the second case is to check the spelling of the property name.

153. Dynamic method or message handler not allowed here

Complete list of compiler error messages

Dynamic and message methods cannot be used as accessor functions for properties.

```
program Produce;

  type
    Base = class
      v : Integer;
      procedure SetV(x : Integer); dynamic;
      function GetV : Integer; message;
      property Velocity : Integer read GetV write v;
      property Value : Integer read v write SetV;
    end;

  procedure Base.SetV(x : Integer);
  begin v := x;
  end;

  function Base.GetV : Integer;
  begin GetV := v;
  end;

begin
end.
```

Both 'Velocity' and 'Value' above are in error since they both have illegal accessor functions assigned to them.

```
program Solve;

  type
    Base = class
      v : Integer;
      procedure SetV(x : Integer);
      function GetV : Integer;
      property Velocity : Integer read GetV write v;
      property Value : Integer read v write SetV;
    end;

  procedure Base.SetV(x : Integer);
  begin v := x;
  end;

  function Base.GetV : Integer;
  begin GetV := v;
  end;

begin
end.
```

The solution taken in this is example was to remove the offending compiler directives from the procedure declarations; this may not be the right solution for you. You may have to closely examine the logic of your program to determine how best to provide accessor functions for your properties.

154. Class does not have a default property

Complete list of compiler error messages

You have used a class instance variable in an array expression, but the class type has not declared a default array property.

```
program Produce;

  type
    Base = class
    end;

  var
    b : Base;

  procedure P;
    var ch : Char;
  begin
    ch := b[1];
  end;

begin
end.
```

The example above elicits an error because 'Base' does not declare an array property, and 'b' is not an array itself.

```
program Solve;

  type
    Base = class
      function GetChar(i : Integer) : Char;
      property data[i : Integer] : Char read GetChar; default;
    end;

  var
    b : Base;

  function Base.GetChar(i : Integer) : Char;
  begin GetChar := 'A';
  end;

  procedure P;
    var ch : Char;
  begin
    ch := b[1];
    ch := b.data[1];
  end;

begin
end.
```

When you have declared a default property for a class, you can use the class instance variable in array expression, as if the class instance variable itself were actually an array. Alternatively, you can use the name of the property as the actual array accessor. Note: if you have hints turned on, you will receive two warnings about the value assigned to 'ch' never being used.

155. Bad argument type in variable type array constructor

Complete list of compiler error messages

You are attempting to construct an array using a type which is not allowed in variable arrays.

```
program Produce;

  type
    Fruit = (apple, orange, pear);
    Data = record
      x : Integer;
      ch : Char;
    end;

  var
    f : Fruit;
    d : Data;

  procedure Examiner(v : array of TVarRec);
  begin
  end;

begin
  Examiner([d]);
  Examiner([f]);
end.
```

Both calls to Examiner will fail because enumerations and records are not supported in array constructors.

```
program Solve;

  var
    i : Integer;
    r : Real;
    v : Variant;

  procedure Examiner(v : array of TVarRec);
  begin
  end;

begin
  i := 0; r := 0; v := 0;
  Examiner([i, r, v]);
end.
```

Many data types, like those in the example above, are allowed in array constructors.

156. Could not load RLINK32.DLL

Complete list of compiler error messages

RLINK32.DLL could not be found. Please ensure that it is on the path.

Contact Inprise if you encounter this error.

157. Wrong or corrupted version of RLINK32.DLL

Complete list of compiler error messages

The internal consistency check performed on the RLINK32.DLL file has failed.

Contact Inprise if you encounter this error.

158. '';' not allowed before 'ELSE'

Complete list of compiler error messages

You have placed a ';' directly before an ELSE in an IF-ELSE statement. The reason for this is that the ';' is treated as a statement separator, not a statement terminator - IF-ELSE is one statement, a ';' cannot appear in the middle (unless you use compound statements).

```
program Produce;

  var
    b : Integer;

begin
  if b = 10 then
    b := 0;
  else
    b := 10;
end.
```

Pascal does not allow a ';' to be placed directly before an ELSE statement. In the code above, an error will be flagged because of this fact.

```
program Solve;

  var
    b : Integer;

begin
  if b = 10 then
    b := 0
  else
    b := 10;

  if b = 10 then begin
    b := 0;
  end
  else begin
    b := 10;
  end;

end.
```

There are two easy solutions to this problem. The first is to remove the offending ';'. The second is to create compound statements for each part of the IF-ELSE. If $HINTS are turned on, you will receive a hint about the value assigned to 'b' is never used. statement.

159. Type '<name>' needs finalization - not allowed in variant record

Complete list of compiler error messages

Certain types are treated specially by the compiler on an internal basis in that they must be correctly finalized to release any resources that they might currently own. Because the compiler cannot determine what type is actually stored in a record's variant section at runtime, it is not possible to guarantee that these special data types are correctly finalized.

```
program Produce;

  type
    Data = record
      case kind:Char of
      'A': (str : String);
    end;

begin
end.
```

String is one of those types which requires special treatment by the compiler to correctly release the resources. As such, it is illegal to have a String in a variant section.

```
program Solve;

  type
    Data = record
      str : String;
    end;

begin
end.
```

One solution to this error is to move all offending declarations out of the variant section. Another solution would be to use pointer types (^String, for example) and manage the memory by yourself.

160. Type '<name>' needs finalization - not allowed in file type

Complete list of compiler error messages

Certain types are treated specially by the compiler on an internal basis in that they must be correctly finalized to release any resources that they might currently own. Because the compiler cannot determine what type is actually stored in a record's variant section at runtime, it is not possible to guarantee that these special data types are correctly finalized.

```
program Produce;

  type
    Data = record
      name : string;
    end;

  var
    inFile : file of Data;

begin
end.
```

String is one of those data types which need finalization, and as such they cannot be stored in a File type.

```
program Solve;

  type
    Data = record
      name : array [1..25] of Char;
    end;

  var
    inFile : file of Data;

begin
end.
```

One simple solution, for the case of String, is to redeclare the type as an array of characters. For other cases which require finalization, it becomes increasingly difficult to maintain a binary file structure with standard Pascal features, such as 'file of'. In these situations, it is probably easier to write specialized file I/O routines.

161. Expression too complicated

Complete list of compiler error messages

The compiler has encounter an expression in your source code that is too complicated for it to handle.

Reduce the complexity of your expression by introducing some temporary variables.

162. Element 0 inaccessible - use 'Length' or 'SetLength'

Complete list of compiler error messages

The Delphi32 String type does not store the length of the string in element 0. The old method of changing, or getting, the length of a string by accessing element 0 does not work with long strings.

```
program Produce;

  var
    str : String;
    len : Integer;

begin
  str := 'Kojo no tsuki';
  len := str[0];
end.
```

Here the program is attempting to get the length of the string by directly accessing the first element. This is not legal.

```
program Solve;

  var
    str : String;
    len : Integer;

begin
  str := 'Kojo no tsuki';
  len := Length(str);
end.
```

You can use the SetLength and Length standard procedures to provide the same functionality as directly accessing the first element of the string. If hints are turned on, you will receive a warning about the value of 'len' not being used.

163. System unit out of date or corrupted: missing '<name>'

Complete list of compiler error messages

The compiler is looking for a special function which resides in System.dcu but could not find it. Your System unit is either corrupted or obsolete.

Make sure there are no conflicts in your library search path which can point to another System.dcu. Try reinstalling System.dcu. If neither of these solutions work, contact Inprise Developer Support.

164. Type not allowed in OLE Automation call

Complete list of compiler error messages

If a data type cannot be converted by the compiler into a Variant, then it is not allowed in an OLE automation call.

```
program Produce;

  type
    Base = class
      x : Integer;
    end;

  var
    B : Base;
    V : Variant;

begin
  V.Dispatch(B);
end.
```

A class cannot be converted into a Variant type, so it is not allowed in an OLE call.

```
program Solve;


  type
    Base = class
      x : Integer;
    end;

  var
    B : Base;
    V : Variant;

begin
  V.Dispatch(B.i);
end.
```

The only solution to this problem is to manually convert these data types to Variants or to only use data types that can automatically be converted into a Variant.

165. RLINK32 error

Complete list of compiler error messages

RLINK32 has encountered an error. Contact Inprise Developer Support if you encounter this error.

166. RLINK32 error

Complete list of compiler error messages

RLINK32 has encountered an error. Contact Inprise Developer Support if you encounter this error.

167. Too many conditional symbols

Complete list of compiler error messages

You have exceeded the memory allocated to conditional symbols defined on the command line (including configuration files). There are 256 bytes allocated for all the conditional symbols. Each conditional symbol requires 1 extra byte when stored in conditional symbol area.

The only solution is to reduce the number of conditional compilation symbols contained on the command line (or in configuration files).

168. Method '<name>' hides virtual method of base type '<name>'

Complete list of compiler error messages

You have declared a method which has the same name as a virtual method in the base class. Your new method is not a virtual method; it will hide access to the base's method of the same name.

```
program Produce;

  type
    Base = class
      procedure VirtuMethod; virtual;
      procedure VirtuMethod2; virtual;
    end;

    Derived = class (Base)
      procedure VirtuMethod;
      procedure VirtuMethod2;
    end;

  procedure Base.VirtuMethod;
  begin
  end;

  procedure Base.VirtuMethod2;
  begin
  end;

  procedure Derived.VirtuMethod;
  begin
  end;

  procedure Derived.VirtuMethod2;
  begin
  end;

begin
end.
```

Both methods declared in the definition of Derived will hide the virtual functions of the same name declared in the base class.

```
program Solve;

  type
    Base = class
      procedure VirtuMethod; virtual;
      procedure VirtuMethod2; virtual;
    end;

    Derived = class (Base)
      procedure VirtuMethod; override;
      procedure Virtu2Method;
    end;

  procedure Base.VirtuMethod;
  begin
  end;

  procedure Base.VirtuMethod2;
  begin
  end;

  procedure Derived.VirtuMethod;
  begin
  end;

  procedure Derived.Virtu2Method;
  begin
  end;

begin
end.
```

There are three alternatives to take when solving this warning.

First, you could specify override to make the derived class' procedure also virtual, and thus allowing inherited calls to still reference the original procedure.

Secondly, you could change the name of the procedure as it is declared in the derived class. Both methods are exhibited in this example.

Finally, you could add the `reintroduce` directive to the procedure declaration to cause the warning to be silenced for that particular method.

Virtual Methods Static Methods Overriding Methods

169. Variable '<name>' is declared but never used in '<name>'

Complete list of compiler error messages

You have declared a variable in a procedure, but you never actually use it. -H

```
program Produce;
(*$HINTS ON*)

  procedure Local;
    var i : Integer;
  begin
  end;

begin
end.
```

```
program Solve;

(*$HINTS ON*)

  procedure Local;
  begin
  end;

begin
end.
```

One simple solution is to remove any unused variable from your procedures. However, unused variables can also indicate an error in the implementation of your algorithm.

170. Compile terminated by user

Complete list of compiler error messages

You pressed Ctrl-Break during a compile.

171. Unnamed arguments must precede named arguments in OLE Automation call

Complete list of compiler error messages

You have attempted to follow named OLE Automation arguments with unnamed arguments.

```
program Produce;

  var
    ole : variant;

begin ole.dispatch(filename:='FrogEggs', 'Tapioca');
end.
```

The named argument, 'filename' must follow the unnamed argument in this OLE dispatch.

```
program Solve;

  var
    ole : variant;

begin ole.dispatch('Tapioca', filename:='FrogEggs');
end.
```

This solution, reversing the parameters, is the most straightforward but it may not be appropriate for your situation. Another alternative would be to provide the unnamed parameter with a name.

172. Abstract methods must be virtual or dynamic

Complete list of compiler error messages

When declaring an abstract method in a base class, it must either be of regular virtual or dynamic virtual type.

```
program Produce;

  type
    Base = class
      procedure DaliVision; abstract;
      procedure TellyVision; abstract;
    end;

begin
end.
```

The declaration above is in error because abstract methods must either be virtual or dynamic.

```
program Solve;

  type
    Base = class
      procedure DaliVision; virtual; abstract;
      procedure TellyVision; dynamic; abstract;
    end;

begin
end.
```

It is possible to remove this error by either specifying 'virtual' or 'dynamic', whichever is most appropriate for your application.

173. Case label outside of range of case expression

Complete list of compiler error messages

You have provided a label inside a case statement which cannot be produced by the case statement control variable. -W

```
program Produce;
(*$WARNINGS ON*)

  type
    CompassPoints = (n, e, s, w, ne, se, sw, nw);
    FourPoints = n..w;

  var
    TatesCompass : FourPoints;

begin

   TatesCompass := e;
   case TatesCompass OF
   n:    Writeln('North');
   e:    Writeln('East');
   s:    Writeln('West');
   w:    Writeln('South');
   ne:   Writeln('Northeast');
   se:   Writeln('Southeast');
   sw:   Writeln('Southwest');
   nw:   Writeln('Northwest');
   end;
end.
```

It is not possible for a TatesCompass to hold all the values of the CompassPoints, and so several of the case labels will elicit errors.

```
program Solve;
(*$WARNINGS ON*)

  type
    CompassPoints = (n, e, s, w, ne, se, sw, nw);
    FourPoints = n..w;

  var
    TatesCompass : CompassPoints;

begin

   TatesCompass := e;
   case TatesCompass OF
   n:    Writeln('North');
   e:    Writeln('East');
   s:    Writeln('West');
   w:    Writeln('South');
   ne:   Writeln('Northeast');
   se:   Writeln('Southeast');
   sw:   Writeln('Southwest');
   nw:   Writeln('Northwest');
   end;
end.
```

After examining your code to determine what the intention was, there are two alternatives. The first is to change the type of the case statement's control variable so that it can produce all the case labels. The second alternative would be to remove any case labels that cannot be produced by the control variable. The first alternative is shown in this example.

174. Field or method identifier expected

Complete list of compiler error messages

You have specified an identifier for a read or write clause to a property which is not a field or method.

```
program Produce;

  var
    r : string;

  type
    Base = class
      t : string;
      property Title : string read Title write Title;
      property Caption : string read r write r;

    end;

begin
end.
```

The two properties in this code both cause errors. The first causes an error because it is not possible to specify the property itself as the read & write methods. The second causes an error because 'r' is not a member of the Base class.

```
program Solve;

  type
    Base = class
      t : string;
      property Title : string read t write t;
    end;

begin
end.
```

To solve this error, make sure that all read & write clauses for properties specify a valid field or method identifier that is a member of the class which owns the property.

175. Constructing instance of '<name>' containing abstract methods

Complete list of compiler error messages

The code you are compiling is constructing instances of classes which contain abstract methods.

```
program Produce;
(*$WARNINGS ON*)
(*$HINTS ON*)

  type
    Base = class
      procedure Abstraction; virtual; abstract;
    end;

  var
    b : Base;

begin
  b := Base.Create;
end.
```

An abstract procedure does not exist, so it becomes dangerous to create instances of a class which contains abstract procedures. In this case, the creation of 'b' is the cause of the warning. Any invocation of 'Abstraction' through the instance of 'b' created here would cause a runtime error. A hint will be issued that the value assigned to 'b' is never used.

```
program Solve;
(*$WARNINGS ON*)
(*$HINTS ON*)

  type
    Base = class
      procedure Abstraction; virtual;
    end;

  var
    b : Base;

  procedure Base.Abstraction;
  begin
  end;

begin
  b := Base.Create;
end.
```

One solution to this problem is to remove the abstract directive from the procedure declaration, as is shown here. Another method of approaching the problem would be to derive a class from Base and then provide a concrete version of Abstraction. A hint will be issued that the value assigned to 'b' is never used.

176. Field definition not allowed after methods or properties

Complete list of compiler error messages

You have attempted to add more fields to a class after the first method or property declaration has been encountered. You must place all field definitions before methods and properties.

```
program Produce;

  type
    Base = class
      procedure FirstMethod;
      a : Integer;
    end;


  procedure Base.FirstMethod;
  begin
  end;

begin
end.
```

The declaration of 'a' after 'FirstMethod' will cause an error.

```
program Solve;

  type
    Base = class
      a : Integer;
      procedure FirstMethod;
    end;


  procedure Base.FirstMethod;
  begin
  end;

begin
end.
```

To solve this error, it is normally sufficient to move all field definitions before the first field or property declaration.

177. Cannot override a static method

Complete list of compiler error messages

You have tried, in a derived class, to override a base method which was not declared as one of the virtual types.

```
program Produce;

  type
    Base = class
      procedure StaticMethod;
    end;

    Derived = class (Base)
      procedure StaticMethod; override;
    end;

    procedure Base.StaticMethod;
    begin
    end;

    procedure Derived.StaticMethod;
    begin
    end;

begin
end.
```

The example above elicits an error because Base.StaticMethod is not declared to be a virtual method, and as such it is not possible to override its declaration.

```
program Solve;

  type
    Base = class
      procedure StaticMethod;
    end;

    Derived = class (Base)
      procedure StaticMethod;
    end;

    procedure Base.StaticMethod;
    begin
    end;

    procedure Derived.StaticMethod;
    begin
    end;

begin
end.
```

The only way to remove this error from your program, when you don't have the source for the base classes, is to remove the 'override' specification from the declaration of the derived method. If you have source to the base classes, you could, with careful consideration, change the base's method to be declared as one of the virtual types - but be aware that this change can have a drastic affect on your programs.

178. Variable '<name>' inaccessible here due to optimization

Complete list of compiler error messages

The evaluator or watch statement is attempting to retrieve the value of <name>, but the compiler was able to determine that the variables actual lifetime ended prior to this inspection point. This error will often occur if the compiler determines a local variable is assigned a value that is not used beyond a specific point in the program's control flow.

```
Create a new application.
Place a button on the form.
Double click the button to be taken to the 'click' method.
Add a global variable, 'c', of type Integer to the implementation
section.

The click method should read as:

  procedure TForm1.Button1Click(Sender: TObject);
    var a, b : integer;
  begin
    a := 10;
    b := 20;
    c := b;
    a := c;
  end;

Set a breakpoint on the assignment to 'c'.
Compile and run the application.
Press the button.
After the breakpoint is reached, open the evaluator (Run|
Evaluate/Watch).
Evaluate 'a'.
```

The compiler realizes that the first assignment to 'a' is dead, since the value is never used. As such, it defers even using 'a' until the second assignment occurs - up until the point where 'c' is assigned to 'a', the variable 'a' is considered to be dead and cannot be used by the evaluator.

The only solution is to only attempt to view variables which are known to have live values.

179. Necessary library helper function was eliminated by linker

Complete list of compiler error messages

The integrated debugger is attempting to use some of the compiler helper functions to perform the requested evaluate. The linker, on the other hand, determined that the helper function was not actually used by the program and it did not link it into the program.

```
Create a new application.
Place a button on the form.
Double click the button to be taken to the 'click' method.
Add a global variable, 'v', of type String to the interface
section.
Add a global variable, 'p', of type PChar to the interface
section.

The click method should read as:

  procedure TForm1.Button1Click(Sender: TObject);
  begin
       v := 'Initialized';
       p := NIL;
       v := 'Abid';
  end;

Set a breakpoint on the second assignment to 'v'.
Compile and run the application.
Press the button.
After the breakpoint is reached, open the evaluator (Run|
Evaluate/Watch).
Evaluate 'v'.
Move the cursor to the 'New Value' box.
Type in 'p'.
Choose Modify.
```

The compiler uses a special function to copy a PChar to a String. In order to reduce the size of the produced executable, if that special function is not used by the program, it is not linked in. In this case, there is no assignment of a PChar to a String, so it is eliminated by the linker.

```
  procedure TForm1.Button1Click(Sender: TObject);
  begin
       v := 'Initialized';
       p := NIL;
       v := 'Abid';
       v := p;
  end;
```

Adding the extra assignment of a PChar to a String will ensure that the linker includes the desired procedure in the program. Encountering this error during a debugging session is an indicator that you are using some language/environment functionality that was not needed in the original program.

180. Missing or invalid conditional symbol in '$<symbol>' directive

Complete list of compiler error messages

The $IFDEF, $IFNDEF, $DEFINE and $UNDEF directives require that a symbol follow them.

```
program Produce;

(*$IFDEF*)
(*$ENDIF*)

begin
end.
```

The $IFDEF conditional directive is incorrectly specified here and will result in an error.

```
program Solve;

(*$IFDEF WIN32*)
(*$ENDIF*)

begin
end.
```

The solution to the problem is to ensure that a symbol to test follows the appropriate directives.

181. '<name>' not previously declared as a PROPERTY

Complete list of compiler error messages

You have attempted to hoist a property to a different visibility level by redeclaration, but <name> in the base class was not declared as a property. -W

```
program Produce;
(*$WARNINGS ON*)

  type
    Base = class
    protected
      Caption : String;
      Title : String;
      property TitleProp : string read Title write Title;
    end;

    Derived = class (Base)
    public
      property Title read Caption write Caption;
    end;

begin
end.
```

The intent of the redeclaration of 'Derived.Title' is to change the field which is used to read and write the property 'Title' as well as hoist it to 'public' visibility. Unfortunately, the programmer really meant to use 'TitleProp', not 'Title'.

```
program Solve;
(*$WARNINGS ON*)

  type
    Base = class
    protected
      Caption : String;
      Title : String;
      property TitleProp : string read Title write Title;
    end;

    Derived = class (Base)
    public
      property TitleProp read Caption write Caption;
      property Title : string read Caption write Caption;
    end;

begin
end.
```

There are a couple ways of approaching this error. The first, and probably the most commonly taken, is to specify the real property which is to be redeclared. The second, which can be seen in the redeclaration of 'Title' addresses the problem by explicitly creating a new property, with the same name as a field in the base class. This new property will hide the base field, which will no longer be accessible without a typecast. (Note: If you have warnings turned on, the redeclaration of 'Title' will issue a warning notifying you that the redeclaration will hide the base class' member.)

182. Field definition not allowed in OLE automation section

Complete list of compiler error messages

You have tried to place a field definition in an OLE automation section of a class declaration. Only properties and methods may be declared in an 'automated' section.

```
program Produce;

  type
    Base = class
    automated
      i : Integer;
    end;

begin
end.
```

The declaration of 'i' in this class will cause the compile error.

```
program Solve;

  type
    Base = class
      i : Integer;
    automated
    end;

begin
end.
```

Moving the declaration of 'i' out of the automated section will vanquish the error.

183. Illegal type in OLE automation section: '<typename>'

Complete list of compiler error messages

<typename> is not an allowed type in an OLE automation section. Only a small subset of all the valid Pascal types are allowed in automation sections.

```
program Produce;

  type
    Base = class
      function GetC : Char;
      procedure SetC(c : Char);
    automated
      property Ch : Char read GetC write SetC dispid 151;
    end;

  procedure Base.SetC(c : Char);
  begin
  end;

  function Base.GetC : Char;
  begin GetC := '!';
  end;

begin
end.
```

Since the character type is not one allowed in the 'automated' section, the declaration of 'Ch' will produce an error when compiled.

```
program Solve;

  type
    Base = class
      function GetC : String;
      procedure SetC(c : String);
    automated
      property Ch : String read GetC write SetC dispid 151;
    end;

  procedure Base.SetC(c : String);
  begin
  end;

  function Base.GetC : String;
  begin GetC := '!';
  end;

begin
end.
```

There are two solutions to this problem. The first is to move the offending declaration out of the 'automated' section. The second is to change the offending type to one that is allowed in 'automated' sections.

184. String constant truncated to fit STRING[<number>]

Complete list of compiler error messages

A string constant is being assigned to a variable which is not large enough to contain the entire string. The compiler is alerting you to the fact that it is truncating the literal to fit into the variable. -W

```
program Produce;
(*$WARNINGS ON*)

  const
    Title = 'Super Galactic Invaders with Turbo Gungla Sticks';
    Subtitle = 'Copyright (c) 1968 by Frank Borland';

  type
    TitleString = String[25];
    SubtitleString = String[18];


  var
    ProgramTitle : TitleString;
    ProgramSubtitle : SubtitleString;

begin
  ProgramTitle := Title;
  ProgramSubtitle := Subtitle;
end.
```

The two string constants are assigned to variables which are too short to contain the entire string. The compiler will truncate the strings and perform the assignment.

```
program Solve;
(*$WARNINGS ON*)

  const
    Title = 'Super Galactic Invaders with Turbo Gungla Sticks';
    Subtitle = 'Copyright (c) 1968';

  type
    TitleString = String[55];
    SubtitleString = String[18];


  var
    ProgramTitle : TitleString;
    ProgramSubtitle : SubtitleString;

begin
  ProgramTitle := Title;
  ProgramSubtitle := Subtitle;
end.
```

There are two solutions to this problem, both of which are demonstrated in this example. The first solution is to increase the size of the variable to hold the string. The second is to reduce the size of the string to fit in the declared size of the variable.

185. Constructors and destructors not allowed in OLE automation section

Complete list of compiler error messages

You have incorrectly tried to put a constructor or destructor into the 'automated' section of a class declaration.

```
program Produce;

  type
    Base = class
    automated
      constructor HardHatBob;
      destructor  DemolitionBob;
    end;

  constructor Base.HardHatBob;
  begin
  end;

  destructor Base.DemolitionBob;
  begin
  end;

begin
end.
```

It is not possible to declare a class constructor or destruction in an OLE automation section. The constructor and destructor declarations in the above code will both elicit this error.

```
program Solve;

  type
    Base = class
      constructor HardHatBob;
      destructor  DemolitionBob;
    end;

  constructor Base.HardHatBob;
  begin
  end;

  destructor Base.DemolitionBob;
  begin
  end;

begin
end.
```

The only solution to this error is to move your declarations out of the automated section, as has been done in this example.

186. Dynamic methods and message handlers not allowed in OLE automation section

Complete list of compiler error messages

You have incorrectly put a dynamic or message method into an 'automated' section of a class declaration.

```
program Produce;

  type
    Base = class
    automated
      procedure DynaMethod; dynamic;
      procedure MessageMethod(VAR msg : Integer); message 151;
    end;

    procedure Base.DynaMethod;
    begin
    end;

    procedure Base.MessageMethod;
    begin
    end;

begin
end.
```

It is not possible to have a dynamic or message method declaration in an OLE automation section of a class. As such, the two method declarations in the above program both produce errors.

```
program Solve;

  type
    Base = class
      procedure DynaMethod; dynamic;
      procedure MessageMethod(VAR msg : Integer); message 151;
    end;

    procedure Base.DynaMethod;
    begin
    end;

    procedure Base.MessageMethod;
    begin
    end;

begin
end.
```

There are several ways to remove this error from your program. First, you could move any declaration which produces this error out of the automated section, as has been done in this example. Alternatively, you could remove the dynamic or message attributes of the method; of course, removing these attributes will not provide you with the desired behavior, but it will remove the error.

187. Only register calling convention allowed in OLE automation section

Complete list of compiler error messages

You have specified an illegal calling convention on a method appearing in an 'automated' section of a class declaration.

```
program Produce;

  type
    Base = class
    automated
      procedure Method; cdecl;
    end;

  procedure Base.Method; cdecl;
  begin
  end;

begin
end.
```

The language specification disallows all calling conventions except 'register' in an OLE automation section. The offending statement is 'cdecl' in the above code.

```
program Solve;

  type
    Base = class
    automated
      procedure Method; register;
      procedure Method2;
    end;

  procedure Base.Method; register;
  begin
  end;

  procedure Base.Method2;
  begin
  end;

begin
end.
```

There are three solutions to this error. The first is to specify no calling convention on methods declared in an auto section. The second is to specify only the register calling convention. The third is to move the offending declaration out of the automation section.

188. Dispid '<number>' already used by '<name>'

Complete list of compiler error messages

An attempt to use a dispid which is already assigned to another member of this class.

```
program Produce;

  type
    Base = class
      v : Integer;
      procedure setV(x : Integer);
      function getV : Integer;
    automated
      property Value : Integer read getV write setV dispid 151;
      property SecondValue : Integer read getV write setV dispid
151;
    end;

  procedure Base.setV(x : Integer);
  begin v := x;
  end;

  function Base.getV : Integer;
  begin getV := v;
  end;

begin
end.
```

Each automated property's dispid must be unique, thus SecondValue is in error.

```
program Solve;

  type
    Base = class
      v : Integer;
      procedure setV(x : Integer);
      function getV : Integer;
    automated
      property Value : Integer read getV write setV dispid 151;
      property SecondValue : Integer read getV write setV dispid
152;
    end;

  procedure Base.setV(x : Integer);
  begin v := x;
  end;

  function Base.getV : Integer;
  begin getV := v;
  end;

begin
end.
```

Giving a unique dispid to SecondValue will remove the error.

189. Redeclaration of property not allowed in OLE automation section

Complete list of compiler error messages

It is not allowed to move the visibility of a property into an automated section.

```
program Produce;

  type
    Base = class
      v : Integer;
      s : String;
    protected
      property Name : String read s write s;
      property Value : Integer read v write v;
    end;

    Derived = class (Base)
    public
      property Name; (* Move Name to a public visibility by
redeclaration *)
    automated
      property Value;
    end;

begin
end.
```

In the above example, Name is moved from a private visibility in Base to public visibility in Derived by redeclaration. The same idea is attempted on Value, but an error results.

```
program Solve;

  type
    Base = class
      v : Integer;
      s : String;
    protected
      property Name : String read s write s;
      property Value : Integer read v write v;
    end;

    Derived = class (Base)
    public
      property Name; (* Move Name to a public visibility by
redeclaration *)
      property Value;
    automated
    end;

begin
end.
```

It is not possible to change the visibility of a property to an automated section, therefore the solution to this problem is to not redeclare properties of base classes in automated sections.

190. '<clause>' clause not allowed in OLE automation section

Complete list of compiler error messages

INDEX, STORED, DEFAULT and NODEFAULT are not allowed in OLE automation sections.

```
program Produce;

  type
     Base = class
       v : integer;
       procedure setV(x : integer);
       function getV : integer;
       automated
       property Value : integer read getV write setV nodefault;
     end;

  procedure Base.setV(x : integer);
  begin v := x;
  end;

  function Base.getV : integer;
  begin getV := v;
  end;

begin
end.
```

Including a NODEFAULT clause on an automated property is not allowed.

```
program Solve;

  type
     Base = class
       v : integer;
       procedure setV(x : integer);
       function getV : integer;
       automated
       property Value : integer read getV write setV;
     end;

  procedure Base.setV(x : integer);
  begin v := x;
  end;

  function Base.getV : integer;
  begin getV := v;
  end;

begin
end.
```

Removing the offending clause will cause the error to go away. Alternatively, moving the property out of the automated section will also make the error go away.

191. Dispid clause only allowed in OLE automation section

Complete list of compiler error messages

A dispid has been given to a property which is not in an automated section.

```
program Produce;

  type
     Base = class
       v : integer;
       procedure setV(x : integer);
       function getV : integer;
       property Value : integer read getV write setV dispid 151;
     end;

  procedure Base.setV(x : integer);
  begin v := x;
  end;

  function Base.getV : integer;
  begin getV := v;
  end;

begin
end.
```

This program attempts to set the dispid for an OLE automation object, but the property has not been declared in an automated section.

```
program Solve;

  type
     Base = class
       v : integer;
       procedure setV(x : integer);
       function getV : integer;
     automated
       property Value : integer read getV write setV dispid 151;
     end;

  procedure Base.setV(x : integer);
  begin v := x;
  end;

  function Base.getV : integer;
  begin getV := v;
  end;

begin
end.
```

To solve the error, you can either remove the dispid clause from the property declaration, or move the property declaration into an automated section.

192. Type '<name>' must be a class to have OLE automation

Complete list of compiler error messages

Old-style Objects cannot have an automated section.

```
program Produce;

  type
    OldObject = object
    automated
    end;

begin
end.
```

It is not possible to have an automated section in an old-style object, thus an error will result from this example.

```
program Solve;

  type
    NewClass = class
    automated
    end;

begin
end.
```

Changing the type from 'object' to 'class', or removing the automated section will remove the error.

193. Type '<name>' must be a class to have a PUBLISHED section

Complete list of compiler error messages

Old-style Objects cannot have a published section. -$M+

```
(*$TYPEINFO ON*)
program Produce;

  type
    OldObject = object
    published
    end;

begin
end.
```

It is not possible to have a published section in an old-style object, thus an error will result from this example.

```
(*$TYPEINFO ON*)
program Solve;

  type
    NewClass = class
    published
    end;

begin
end.
```

Changing the type from 'object' to 'class', or removing the published section will remove the error.

194. Redeclaration of '<name>' hides a member in the base class

Complete list of compiler error messages

A property has been created in a class with the same name of a variable contained in one of the base classes. One possible, and not altogether apparent, reason for getting this error is that a new version of the base class hierarchy has been installed and it contains new member variables which have names identical to your properties' names. -W

```
(*$WARNINGS ON*)
program Produce;

  type
    Base = class
      v : integer;
    end;

    Derived = class (Base)
      ch : char;
      property v : char read ch write ch;
    end;

begin
end.
```

Derived.v overrides, and thus hides, Base.v; it will not be possible to access Base.v in any variable of type Derived without a typecast.

```
(*$WARNINGS ON*)
program Solve;
  type
    Base = class
      v : integer;
    end;

    Derived = class (Base)
      ch : char;
      property chV : char read ch write ch;
    end;

begin
end.
```

By changing the name of the property in the derived class, the error is alleviated.

195. Overriding automated virtual method '<name>' cannot specify a dispid

Complete list of compiler error messages

The dispid declared for the original virtual automated procedure declaration must be used by all overriding procedures in derived classes.

```
program Produce;

  type
    Base = class
    automated
      procedure Automatic; virtual; dispid 151;
    end;


    Derived = class (Base)
    automated
      procedure Automatic; override; dispid 152;
    end;

  procedure Base.Automatic;
  begin
  end;

  procedure Derived.Automatic;
  begin
  end;

begin
end.
```

The overriding declaration of Base.Automatic, in Derived (Derived.Automatic) erroneously attempts to define another dispid for the procedure.

```
program Solve;

  type
    Base = class
    automated
      procedure Automatic; virtual; dispid 151;
    end;


    Derived = class (Base)
    automated
      procedure Automatic; override;
    end;

  procedure Base.Automatic;
  begin
  end;

  procedure Derived.Automatic;
  begin
  end;

begin
end.
```

By removing the offending dispid clause, the program will now compile.

196. Published Real48 property '<name>' must be Single, Real, Double or Extended

Complete list of compiler error messages

You have attempted to publish a property of type Real, which is not allowed. Published floating point properties must be Single, Double or Extended.

```
program Produce;
  type
    Base = class
      R : Real48;
    published
      property RVal : Real read R write R;
    end;
end.
```

The published Real48 property in the program above must be either removed, moved to an unpublished section or changed into an acceptable type.

```
program Produce;
  type
    Base = class
      R : Single;
    published
      property RVal : Single read R write R;
    end;
end.
```

This solution changed the property into a real type that will actually produce run-time type information.

197. Size of published set '<name>' is >32 bits

Complete list of compiler error messages

The compiler does not allow sets greater than 32 bits to be contained in a published section. The size, in bytes, of a set can be calculated by High(setname) div 8 - Low(setname) div 8 + 1. -$M+

```
(*$TYPEINFO ON*)
program Produce;
  type
    CharSet = set of Char;
    NamePlate = class
      Characters : CharSet;
    published
      property TooBig : CharSet read Characters write
Characters ;
    end;

begin
end.
```

```
(*$TYPEINFO ON*)
program Solve;
  type
    CharSet = set of 'A'..'Z';
    NamePlate = class
      Characters : CharSet;
    published
      property TooBig : CharSet read Characters write
Characters ;
    end;

begin
end.
```

198. Published property '<name>' cannot be of type <type>

Complete list of compiler error messages

Published properties must be an ordinal type, Single, Double, Extended, Comp, a string type, a set type which fits in 32 bits, or a method pointer type. When any other property type is encountered in a published section, the compiler will remove the published attribute -$M+

```
(*$TYPEINFO ON*)
program Produce;

  type
    TitleArr = array [0..24] of char;
    NamePlate = class
    private
      titleStr : TitleArr;
    published
      property Title : TitleArr read titleStr write titleStr;
    end;

begin
end.
```

An error is induced because an array is not one of the data types which can be published.

```
(*$TYPEINFO ON*)
program Solve;

  type
    TitleArr = integer;
    NamePlate = class
      titleStr : TitleArr;
    published
      property Title : TitleArr read titleStr write titleStr;
    end;

begin
end.
```

Moving the property declaration out of the published section will avoid this error. Another alternative, as in this example, is to change the type of the property to be something that can actually be published.

199. Thread local variables cannot be local to a function

Complete list of compiler error messages

Thread local variables must be declared at a global scope.

```
program Produce;

  procedure NoTLS;
    threadvar
      x : Integer;
  begin
  end;

begin
end.
```

A thread variable cannot be declared local to a procedure.

```
program Solve;

  threadvar
    x : Integer;

  procedure YesTLS;
    var
      localX : Integer;
  begin
  end;

begin
end.
```

There are two simple alternatives for avoiding this error. First, the threadvar section can be moved to a local scope. Secondly, the threadvar in the procedure could be changed into a normal var section. Note that if compiler hints are turned on, a hint about localX being declared but not used will be emitted.

200. Thread local variables cannot be ABSOLUTE

Complete list of compiler error messages

A thread local variable cannot refer to another variable, nor can it reference an absolute memory address.

```
program Produce;

  threadvar
     secretNum : integer absolute $151;

begin
end.
```

The absolute directive is not allowed in a threadvar declaration section.

```
program Solve;

  threadvar
    secretNum : integer;

  var
    sNum : integer absolute $151;

begin
end.
```

There are two easy ways to solve a problem of this nature. The first is to remove the absolute directive from the threadvar section. The second would be to move the absolute variable to a normal var declaration section.

201. EXPORTS allowed only at global scope

Complete list of compiler error messages

An EXPORTS clause has been encountered in the program source at a non-global scope.

```
program Produce;

  procedure ExportedProcedure;
  exports ExportedProcedure;
  begin
  end;

begin
end.
```

It is not allowed to have an EXPORTS clause anywhere but a global scope.

```
program Solve;

  procedure ExportedProcedure;
  begin
  end;

exports ExportedProcedure;
begin
end.
```

The solution is to ensure that your EXPORTS clause is at a global scope and textually follows all procedures named in the clause. As a general rule, EXPORTS clauses are best placed right before the source file's initialization code.

202. Constants cannot be used as open array arguments

Complete list of compiler error messages

Open array arguments must be supplied with an actual array variable, a constructed array or a single variable of the argument's element type.

```
program Produce;

  procedure TakesArray(s : array of String);
  begin
  end;


begin TakesArray('Hello Error');
end.
```

The error is caused in this example because a string literal is being supplied when an array is expected. It is not possible to implicitly construct an array from a constant.

```
program Solve;

  procedure TakesArray(s : array of String);
  begin
  end;


begin TakesArray(['Hello Error']);
end.
```

The solution avoids the error because the array is explicitly constructed.

203. Slice standard function only allowed as open array argument

Complete list of compiler error messages

An attempt has been made to pass an array slice to a fixed size array. Array slices can only be sent to open array parameters. none

```
program Produce;

  type
    IntegerArray = array [1..10] OF Integer;

  var
    SliceMe : array [1..200] OF Integer;

  procedure TakesArray(x : IntegerArray);
  begin
  end;

begin TakesArray(SLICE(SliceMe, 5));
end.
```

In the above example, the error is produced because TakesArray expects a fixed size array.

```
program Solve;

  type
    IntegerArray = array [1..10] OF Integer;

  var
    SliceMe : array [1..200] OF Integer;

  procedure TakesArray(x : array of Integer);
  begin
  end;

begin TakesArray(SLICE(SliceMe, 5));
end.
```

In the above example, the error is not produced because TakesArray takes an open array as the parameter.

204. Cannot initialize thread local variables

Complete list of compiler error messages

The compiler does not allow initialization of thread local variables.

```
program Produce;

  threadvar
    tls : Integer = 151;

begin
end.
```

The declaration and initialization of 'tls' above is not allowed.

```
program Solve;

  threadvar
    tls : Integer;

begin tls := 151;
end.
```

You can declare thread local storage as normal, and then initialize it in the initialization section of your source file.

205. Cannot initialize local variables

Complete list of compiler error messages

The compiler disallows the use of initialized local variables.

```
program Produce;

  var
    j : Integer;

  procedure Show;
    var i : Integer = 151;
  begin
  end;

begin
end.
```

The declaration and initialization of 'i' in procedure 'Show' is illegal.

```
program Solve;

  var
    j : Integer;

  procedure Show;
    var i : Integer;
  begin
    i := 151;
  end;

begin
  j := 0;
end.
```

You can use a programmatic style to set all variables to known values.

206. Cannot initialize multiple variables

Complete list of compiler error messages

Variable initialization can only occur when variables are declared individually.

```
program Produce;

  var
    i, j : Integer = 151, 152;

begin
end.
```

The compiler will disallow the declaration and initialization of more than one variable at a time.

```
program Solve;

  var
    i : Integer = 151;
    j : Integer = 152;

begin
end.
```

Simple declare each variable by itself to allow initialization.

207. Constant object cannot be passed as var parameter

Complete list of compiler error messages

As variable parameters are intended to be modified by the called procedure or function, you can not pass a constant object to a variable parameter.

If your intention is just to pass a big datastructure efficiently, and the called function should not modify it, you can use a const parameter instead.

```
program Produce;
(*$APPTYPE CONSOLE*)

function Max(var A: array of Integer): Integer;
var I: Integer;
begin
  Result := Low(Integer);
  for I := 0 to High(A) do
    if Result < A[I] then
       Result := A[I];
end;

begin
  Writeln( Max([1,2,3]) );   (*<-- Error message here*)
end.
```

In the example, function has a variable parameter, but we are passing a constant to it.

```
program Solve;
(*$APPTYPE CONSOLE*)

function Max(const A: array of Integer): Integer;
var I: Integer;
begin
  Result := Low(Integer);
  for I := 0 to High(A) do
    if Result < A[I] then
      Result := A[I];
end;

begin
  Writeln( Max([1,2,3]) );
end.
```

The solution is to declare the parameter as a constant parameter (we do not intend to modify it, after all). Alternatively, you can also modify the call so it does not pass constants.

208. HIGH cannot be applied to a long string

Complete list of compiler error messages

It is not possible to use the standard function HIGH with long strings. The standard function HIGH can, however, be applied to old-style short strings.

Since long strings will dynamically size themselves, there is no analog to the HIGH function which can be used.

This error can be caused if you are porting a 16-bit application to, in which case the only string type available was a short string. If this is the case, then you can turn off the long strings with the $H command line switch or the long-form directive $LONGSTRINGS.

If the HIGH was applied to a string parameter, but you still wish to use long strings, you could change the parameter type to 'openstring'.

```
program Produce;
  var
    i : Integer;
    s : String;

begin
  s := 'Hello, Delphi';
  i := HIGH(s);
end.
```

In the example above, the programmer has attempted to apply the standard function HIGH to a long string variable. This cannot be done.

```
(*$LONGSTRINGS OFF*)
program Solve;
  var
    i : Integer;
    s : String;

begin
  s := 'Hello, Delphi';
  i := HIGH(s);
end.
```

By disabling long string parameters, the application of HIGH to a string variable is now allowed.

209. Unit '<Name>' implicitly imported into package '<Name>'

Complete list of compiler error messages

The unit specified was not named in the contains clause of the package, but a unit which has already been included in the package imports it.

This message will help the programmer avoid violating the rule that a unit may not reside in more than one related package.

Ignoring the warning, will cause the unit to be put into the package. You could also explicitly list the named unit in the contains clause of the package to accomplish the same result and avoid the warning altogether. Or, you could alter the package list to load the named unit from another package.

```
package Produce;
  contains Classes;
end.
```

In the above program, Classes uses (either directly or indirectly) 'consts', 'TypInfo', and 'SysUtils'. We will get a warning message for each of these units.

```
package Solve;
  contains consts, TypInfo, SysUtils, Classes;
end.
```

The best solution for this problem is to explicitly name all the units which will be imported into the package in the contains clause, as has been done here.

210. Packages '<name>' and '<name>' both contain unit '<name>'

Complete list of compiler error messages

The project you are trying to compile is using two packages which both contain the same unit. It is illegal to have two packages which are used in the same project containing the same unit since this would cause an ambiguity for the compiler.

A main cause of this problem is a package set which has been poorly defined.

The only solution to this problem is to redesign your package hierarchy to remove the ambiguity.

211. Package '<name>' already contains unit '<name>'

Complete list of compiler error messages

The package you are compiling requires (either through the requires clause or the package list) another package which already contains the unit specified in the message.

It is an error to have to related packages contain the same unit. The solution to this problem is to remove the unit from one of the packages or to remove the relation between the two packages.

212. File not found: '<name>.dcu'

Complete list of compiler error messages

The compiler needed to load the DCU file specified in the message but was unable to do so. Failure to set the unit/library path for the compiler is a likely cause of this message.

The only solution is to make sure the named unit can be found along the library path.

213. Need imported data reference ($G) to access '<name>' from unit '<name>'

Complete list of compiler error messages

The unit named in the message was not compiled with the $G switch turned on.

```
(*$IMPORTEDDATA OFF*)
unit u0;
interface
implementation
begin
  WriteLn(System.RandSeed);
end.

program u1;
  uses u0;
end.
```

In the above example, u0 should be compiled alone. Then, u1 should be compiled with the VCLxx (where xx represents the version). The problem occurs because u0 is compiled under the premise that it will never use data which resides in a package. in a package

```
(*$IMPORTEDDATA ON*)
unit u0;
interface
implementation
begin
  WriteLn(System.RandSeed);
end.

program u1;
  uses u0;
end.
```

To alleviate the problem, it is generally easiest just to turn on the $IMPORTEDDATA switch and recompile the unit which produces the error.

214. Required package '<name>' not found

Complete list of compiler error messages

The package which is referenced in the message appears on the package list, either explicitly or through a requires clause of another unit appearing on the package list, bit can not be found by the compiler.

The solution to this problem is to ensure that the DCP file for the named package is in one of the units named in the library path.

215. $WEAKPACKAGEUNIT '<name>' contains global data

Complete list of compiler error messages

A unit which was marked with $WEAKPACKAGEUNIT is being placed into a package, but it contains global data. It is not legal for such a unit to contain global data or initialization or finalization code.

The only solutions to this problem are to remove the $WEAKPACKAGEUNIT mark, or remove the global data from the unit before it is put into the package.

216. Improper GUID syntax

Complete list of compiler error messages

The GUID encountered in the program source is malformed. A GUID must be of the form: 00000000-0000-0000-0000-000000000000.

```
program Produce;

begin
end.
```

```
program Solve;

begin
end.
```

217. Interface type required

Complete list of compiler error messages

A type which is an interface was expected but was not found. A common cause of this error is the specification of a user-defined type which has not been declared as an interface type.

```
program Produce;
  type
    Name = string;

    MyObject = class
    end;

    MyInterface = interface(MyObject)
    end;

    Base = class(TObject, Name)
    end;

begin
end.
```

In this example, the type 'Base' is erroneously declared since 'Name' is not declared as an interface type. Likewise, 'MyInterface' is incorrectly declared because its ancestor interface was not declared as such.

```
program Solve;
  type
    BaseInterface = interface
    end;

    MyInterface = interface(BaseInterface)
    end;

    Base = class(TObject, MyInterface)
    end;

begin
end.
```

The best solution when encountering this error is to reexamine the source code to determine what was really intended. If a class is to implement an interface, it must first be explicitly derived from a base type such as TObject. When extended, interfaces can only have a single interface as its ancestor.

In the example above, the interface is properly derived from another interface and the object definition correctly specifies a base so that interfaces can be specified.

218. Property overrides not allowed in interface type

Complete list of compiler error messages

A property which was declared in a base interface has been overridden in an interface extension.

```
program Produce;
  type
    Base = interface
      function Reader : Integer;
      function Writer(a : Integer);
      property Value : Integer read Reader write Writer;
    end;

    Extension = interface (Base)
      function Reader2 : Integer;
      property Value Integer read Reader2;
    end;

begin
end.
```

The error in the example is that Extension attempts to override the Value property.

```
program Solve;
  type
    Base = interface
      function Reader : Integer;
      function Writer(a : Integer);
      property Value : Integer read Reader write Writer;
    end;

    Extension = interface (Base)
      function Reader2 : Integer;
      property Value2 Integer read Reader2;
    end;

begin
end.
```

A solution to this error is to rename the offending property. Another, more robust, approach is to determine the original intent and restructure the system design to solve the problem.

219. '<name>' clause not allowed in interface type

Complete list of compiler error messages

The clause noted in the message is not allowed in an interface type. Typically this error indicates that an illegal directive has been specified for a property field in the interface.

```
program Produce;
  type
    Base = interface
      function Reader : Integer;
      procedure Writer(a : Integer);
      property Value : Integer read Reader write Writer stored
false;
    end;
begin
end.
```

The problem in the above program is that the stored directive is not allowed in interface types.

```
program Solve;
  type
    Base = interface
      function Reader : Integer;
      procedure Writer(a : Integer);
      property Value : Integer read Reader write Writer;
    end;

begin
end.
```

The solution to problems of this nature are to remove the offending directive. Of course, it is best to understand the desired behavior and to implement it in some other fashion.

220. Interface '<name1>' already implemented by '<name2>'

Complete list of compiler error messages

The class specified by name2 has specified the interface name1 more than once in the inheritance section of the class definition.

```
program Produce;
  type
    IBaseIntf = interface
    end;

    TBaseClass = class (TInterfacedObject, IBaseIntf, IBaseIntf)
    end;

begin
end.
```

In this example, the IBaseIntf interface is specified multiple times in the inheritance section of the definition of TBaseClass. As a class can not implement the same interface more than once, this cause the compiler to emit the error message.

```
program Solve;

  type
    IBaseIntf = interface
    end;

    TBaseClass = class (TInterfacedObject, IBaseIntf)
    end;

begin
end.
```

The only solution to this error message is to ensure that a particular interface appears no more than once in the inheritance section of a class definition.

221. Field declarations not allowed in interface type

Complete list of compiler error messages

An interface has been encountered which contains definitions of fields; this is not permitted.

```
program Produce;
  type
    IBaseIntf = interface
      FVar : Integer;
      property Value : Integer read FVar write FVar;
    end;

begin
end.
```

The desire above is to have a property which has a value associated with it. However, as interfaces can have no fields, this idea will not work.

```
program Solve;
    IBaseIntf = interface
      function Reader : Integer;
      procedure Writer(a : Integer);
      property Value : Integer read Reader write Writer;
    end;

begin
end.
```

An elegant solution to the problem described above is to declare getter and setter procedures for the property. In this situation, any class implementing the interface must provide a method which will be used to access the data of the class.

222. '<name>' directive not allowed in interface type

Complete list of compiler error messages

A directive was encountered during the parsing of an interface which is not allowed.

```
program Produce;
  type
    IBaseIntf = interface
    private
      procedure fnord(x, y, z : Integer);
    end;

begin
end.
```

In this example, the compiler gives an error when it encounters the private directive, as it is not allowed in interface types.

```
program Solve;
  type
    IBaseIntf = interface
      procedure fnord(x, y, z : Integer);
    end;

    TBaseClass = class (TInterfacedObject, IBaseIntf)
    private
      procedure fnord(x, y, z : Integer);
    end;

  procedure TBaseClass.fnord(x, y, z : Integer);
  begin
  end;
begin
end.
```

The only solution to this problem is to remove the offending directive from the interface definition. While interfaces do not actually support these directives, you can place the implementing method into the desired visibility section. In this example, placing the TBaseClass.fnord procedure into a private section should have the desired results.

223. Declaration of '<name1>' differs from declaration in interface '<name2>'

Complete list of compiler error messages

A method declared in a class which implements an interface is different from the definition which appears in the interface. Probable causes are that a parameter type or return value is declared differently, the method appearing in the class is a message method, the identifier in the class is a field or the identifier in the class is a property, which does not match with the definition in the interface.

```
program Produce;

  type
    IBaseIntf = interface
      procedure p0(var x : Shortint);
      procedure p1(var x : Integer);
      procedure p2(var x : Integer);
    end;

    TBaseClass = class (TInterfacedObject)
      procedure p1(var x : Integer); message 151;
    end;

    TExtClass = class (TBaseClass, IBaseIntf)
      p2 : Integer;
      procedure p0(var x : Integer);
      procedure p1(var x : Integer); override;
    end;

  procedure TBaseClass.p1(var x : Integer);
  begin
  end;

  procedure TExtClass.p0(var x : Integer);
  begin
  end;

  procedure TExtClass.p1(var x : Integer);
  begin
  end;

begin
end.
```

Generally, as in this example, errors of this type are plain enough to be easily visible. However, as can be seen with p1, things can be more subtle. Since p1 is overriding a procedure from the inherited class, p1 also inherits the virtuality of the procedure defined in the base class.

```
program Solve;

  type
    IBaseIntf = interface
      procedure p0(var x : Shortint);
      procedure p1(var x : Integer);
      procedure p2(var x : Integer);
    end;

    TBaseClass = class (TInterfacedObject)
      procedure p1(var x : Integer); message 151;
    end;

    TExtClass = class (TBaseClass, IBaseIntf)
      p2 : Integer;

      procedure IBaseIntf.p1 = p3;
      procedure IBaseIntf.p2 = p4;

      procedure p0(var x : Shortint);
      procedure p1(var x : Integer); override;
      procedure p3(var x : Integer);
      procedure p4(var x : Integer);
    end;

  procedure TBaseClass.p1(var x : Integer);
  begin
  end;

  procedure TExtClass.p0(var x : Shortint);
  begin
  end;

  procedure TExtClass.p1(var x : Integer);
  begin
  end;

  procedure TExtClass.p3(var x : Integer);
  begin
  end;

  procedure TExtClass.p4(var x : Integer);
  begin
  end;

begin
end.
```

One approach to solving this problem is to use a message resolution clause for each problematic identifier, as is done in the example shown here. Another viable approach, which requires more thoughtful design, would be to ensure that the class identifiers are compatible to the interface identifiers before compilation.

224. Package unit '<name>' cannot appear in contains or uses clauses

Complete list of compiler error messages

The unit named in the error is a package unit and as such cannot be included in your project. A possible cause of this error is that somehow a Pascal unit and a package unit have been given the same name. The compiler is finding the package unit on its search path before it can locate a same-named Pascal file. Packages cannot be included in a project by inclusion of the package unit in the uses clause.

225. Bad packaged unit format: <name>.<name>

Complete list of compiler error messages

When the compiler attempted to load the specified unit from the package, it was found to be corrupt. This problem could be caused by an abnormal termination of the compiler when writing the package file (for example, a power loss). The first recommended action is to delete the offending DCP file and recompile the package. If this fails, contact Inprise Developer Support.

226. Package '<name>' is recursively required

Complete list of compiler error messages

When compiling a package, the compiler determined that the package requires itself. the

```
package Produce;
  requires Produce;

end.
```

The error is caused because it is not legal for a package to require itself.

The only solution to this problem is to remove the recursive use of the package.

227. 16-Bit segment encountered in object file '<name>'

Complete list of compiler error messages

A 16-bit segment has been found in an object file which was loaded using the $L directive.

```
end.
```

The only solution to this error is to obtain an object file which does not have a 16-bit segment definition. You should consult the documentation for the product which produced the object file for instructions on turning 16-bit segment definitions into 32-bit segment definitions.

228. Published field '<name>' not a class nor interface type

Complete list of compiler error messages

An attempt has been made to publish a field in a class which is not a class nor interface type.

```
program Produce;

  type
    TBaseClass = class
    published
      x : Integer;
    end;
begin
end.
```

The program above generates an error because x is included in a published section, despite the fact that it is not of a type which can be published.

```
program Solve;
  type
    TBaseClass = class
      Fx : Integer;
    published
      property X : Integer read Fx write Fx;
    end;

begin
end.
```

To solve this problem, all fields which are not class nor interface types must be removed from the published section of a class. If it is a requirement that the field actually be published, then it can be accomplished by changing the field into a property, as was done in this example.

229. Private symbol '<name>' declared but never used

Complete list of compiler error messages

The symbol referenced appears in a private section of a class, but is never used by the class. It would be more memory efficient if you removed the unused private field from your class definition.

```
program Produce;
  type
    Base = class
    private
      FVar : Integer;
      procedure Init;
    end;

procedure Base.Init;
begin
end;

begin
end.
```

Here we have declared a private variable which is never used. The message will be emitted for this case.

```
program Solve;
program Produce;
  type
    Base = class
    private
      FVar : Integer;
      procedure Init;
    end;

procedure Base.Init;
begin
  FVar := 0;
end;

begin
end.
```

There are various solutions to this problem, and since this message is not an error message, all are correct. If you have included the private field for some future use, it would be valid to ignore the message. Or, if the variable is truly superfluous, it can be safely removed. Finally, it might have been a programming oversight not to use the variable at all; in this case, simply add the code you forgot to implement.

230. Could not compile package '<name>'

Complete list of compiler error messages

An error occurred while trying to compile the package named in the message. The only solution to the problem is to correct the error and recompile the package.

231. Never-build package '<name>' requires always-build package '<name>'

Complete list of compiler error messages

You are attempting to create a no-build package which requires an always-build package. Since the interface of an always-build package can change at anytime, and since giving the no-build flag instructs the compiler to assume that a package is up-to-date, each no-build package can only require other packages that are also marked no-build.

```
package Base;
end.

(*$IMPLICITBUILD OFF*)
package NoBuild;
  requires Base;
end.
```

In this example, the NoBuild package requires a package which was compiled in the always-build compiler state.

```
(*$IMPLICITBUILD OFF*)
package Base;
end.

(*$IMPLICITBUILD OFF*)
package NoBuild;
  requires Base;
end.
```

The solution used in this example was to turn Base into a never-build package. Another viable option would have been to remove the (*$IMPLICITBUILD OFF*) from the NoBuild package, thereby turning it into an always-build package.

232. $WEAKPACKAGEUNIT '<name>' cannot have initialization or finalization code

Complete list of compiler error messages

A unit which has been flagged with the $weakpackageunit directive cannot contain initialization or finalization code, nor can it contain global data. The reason for this is that multiple copies of the same weakly packaged units can appear in an application, and then referring to the data for that unit becomes and ambiguous proposition. This ambiguity is furthered when dynamically loaded packages are used in your applications.

```
(*$WEAKPACKAGEUNIT*)
unit yamadama;
interface
implementation
  var
    Title : String;

initialization
  Title := 'Tiny Calc';
finalization
end.
```

In the above example, there are two problems: Title is a global variable, and Title is initialized in the initialization section of the unit.

There are only two alternatives: either remove the $weakpakcageunit directive from the unit, or remove all global data, initialization and finalization code.

233. $WEAKPACKAGEUNIT & $DENYPACKAGEUNIT both specified

Complete list of compiler error messages

It is not legal to specify both $WEAKPACKAGEUNIT & $DENYPACKAGEUNIT. Correct the source code and recompile.

234. $DENYPACKAGEUNIT '<name>' cannot be put into a package

Complete list of compiler error messages

You are attempting to put a unit which was compiled with $DENYPACKAGEUNIT into a package. It is not possible to put a unit compiled with the $DENYPACKAGEUNIT direction into a package.

235. $DESIGNONLY and $RUNONLY only allowed in package unit

Complete list of compiler error messages

The compiler has encountered either $designonly or $runonly in a source file which is not a package. These directives affect the way that the IDE will treat a package DLL, and therefore can only be contained in package source files.

236. Never-build package '<name>' must be recompiled

Complete list of compiler error messages

The package referenced in the message was compiled as a never-build package, but it requires another package to which interface changes have been made. The named package cannot be used without recompiling because it was linked with a different interface of the required package.

The only solution to this error is to manually recompile the offending package. Be sure to specify the never-build switch, if it is still desired.

237. Compilation terminated; too many errors

Complete list of compiler error messages

The compiler has surpassed the maximum number of errors which can occur in a single compilation.

The only solution is to address some of the errors and recompile the project.

238. Imagebase is too high - program exceeds 2 GB limit

Complete list of compiler error messages

There are three ways to cause this error: 1. Specify a large enough imagebase that, when compiled, the application code passes the 2GB boundary. 2. Specify an imagebase via the command line which is above 2GB. 3. Specify an imagebase via $imagebase which is above 2GB.

The only solution to this problem is to lower the imagebase address sufficiently so that the entire application will fit below the 2GB limit.

239. A dispinterface type cannot have an ancestor interface

Complete list of compiler error messages

An interface type specified with dispinterface cannot specify an ancestor interface.

```
program Produce;

  type
    IBase = interface
    end;

    IExtend = dispinterface (IBase)
    ['{00000000-0000-0000-0000-000000000000}']

    end;

begin
end.
```

In the example above, the error is caused because IExtend attempts to specify an ancestor interface type.

```
program Solve;

  type
    IBase = interface
    end;

    IExtend = dispinterface
    ['{00000000-0000-0000-0000-000000000000}']

    end;

begin
end.
```

Generally there are two solutions when this error occurs: remove the ancestor interface declaration, or change the dispinterface into a regular interface type. In the example above, the former approach was taken.

240. A dispinterface type requires an interface identification

Complete list of compiler error messages

When using dispinterface types, you must always be sure to include a GUID specification for them.

```
program Produce;

  type
    IBase = dispinterface
    end;

begin
end.
```

In the example shown here, the dispinterface type does not include a GUID specification, and thus causes the compiler to emit an error.

```
program Solve;

  type
    IBase = dispinterface
    ['{00000000-0000-0000-0000-000000000000}']

    end;

begin
end.
```

Ensuring that each dispinterface has a GUID associated with it will cause this error to go away.

241. Methods of dispinterface types cannot specify directives

Complete list of compiler error messages

Methods declared in a dispinterface type cannot specify any calling convention directives.

```
program Produce;

  type
    IBase = dispinterface
    ['{00000000-0000-0000-0000-000000000000}']
      procedure yamadama; register;
    end;

begin
end.
```

The error in the example shown here is that the method 'yamadama' attempts to specify the register calling convention.

```
program Solve;

  type
    IBase = dispinterface
    ['{00000000-0000-0000-0000-000000000000}']
      procedure yamadama;
    end;

begin
end.
```

Since no dispinterface method can specify calling convention directives, the only solution to this problem is to remove the offending directive, as shown in this example.

242. '<text>' directive not allowed in dispinterface type

Complete list of compiler error messages

You have specified a clause in a dispinterface type which is not allowed.

```
program Produce;

  type
    IBase = dispinterface
    ['{00000000-0000-0000-0000-000000000000}']
      function Get : Integer;

      property BaseValue : Integer read Get;
    end;

    IExt = interface (IBase)
    end;


begin
end.
```

```
program Solve;

  type
    IBase = dispinterface
    ['{00000000-0000-0000-0000-000000000000}']
      function Get : Integer;

      property BaseValue : Integer;
    end;

begin
end.
```

243. Interface '<name>' has no interface identification

Complete list of compiler error messages

You have attempted to assign an interface to a GUID type, but the interface was not defined with a GUID.

```
program Produce;

  type
    IBase = interface
    end;

  var
    g : TGUID;

  procedure p(x : TGUID);
  begin
  end;

begin
  g := IBase;
  p(IBase);
end.
```

In this example, the IBase type is defined but it is not given an interface, and is thus cannot be assigned to a GUID type.

```
program Solve;

  type
    IBase = interface
    ['{00000000-0000-0000-0000-000000000000}']
    end;

  var
    g : TGUID;

  procedure p(x : TGUID);
  begin
  end;

begin
  g := IBase;
  p(IBase);
end.
```

To solve the problem, you must either not attempt to assign an interface type without a GUID to a GUID type, or you must assign a GUID to the interface when it is defined. In this solution, a GUID has been assigned to the interface type when it is defined.

244. Property '<name>' inaccessible here

Complete list of compiler error messages

An attempt has been made to access a property through a class reference type. It is not possible to access fields nor properties of a class through a class reference.

```
program Produce;

  type
    TBase = class
    public
      FX : Integer;
      property X : Integer read FX write FX;
    end;

    TBaseClass = class of TBase;

  var
    BaseRef : TBaseClass;
    x : Integer;

begin
  BaseRef := TBase;
  x := BaseRef.X;
end.
```

Attempting to access the property X in the example above causes the compiler to issue an error.

```
program Solve;

  type
    TBase = class
    public
      FX : Integer;
      property X : Integer read FX write FX;
    end;

    TBaseClass = class of TBase;

  var
    BaseRef : TBaseClass;
    x : Integer;

begin
  BaseRef := TBase;
end.
```

There is no other solution to this problem than to remove the offending property access from your source code. If you wish to access properties or fields of a class, then you need to create an instance variable of that class type and gain access through that variable.

245. Unsupported language feature: '<text>'

Complete list of compiler error messages

You are attempting to translate a Pascal unit to a C++ header file which contains unsupported language features.

You must remove the offending construct from the interface section before the unit can be translated.

246. Getter or setter for property '<name>' cannot be found

Complete list of compiler error messages

During translation of a unit to a C++ header file, the compiler is unable to locate a named symbol which is to be used as a getter or setter for a property. This is usually caused by having nested records in the class and the accessor is a field in the nested record.

247. Package '<name>' does not use or export '<unit>.<name>'

Complete list of compiler error messages

You have compiled a unit into a package which contains a symbol which does not appear in the interface section of the unit, nor is it referenced by any code in the unit. In effect, this code is dead code and could be removed from the unit without changing the semantics of your program.

248. Constructors and destructors must have register calling convention

Complete list of compiler error messages

An attempt has been made to change the calling convention of a constructor or destructor from the default register calling convention.

```
program Produce;

  type
    TBase = class
      constructor Create; pascal;
    end;

  constructor TBase.Create;
  begin
  end;

begin
end.
```

```
program Solve;

  type
    TBase = class
      constructor Create;
    end;

  constructor TBase.Create;
  begin
  end;

begin
end.
```

The only viable approach when this error has been issued by the compiler is to remove the offending calling convention directive from the constructor or destructor definition, as has been done in this example.

249. Parameter '<name>' not allowed here due to default value

Complete list of compiler error messages

When using default parameters a list of parameters followed by a type is not allowed; you must specify each variable and its default value individually.

```
program Produce;

  procedure p0(a, b : Integer = 151);
  begin
  end;

begin
end.
```

The procedure definitions shown above will cause this error since it declares two parameters with a default value.

```
program Solve;

  procedure p0(a : Integer; b : Integer = 151);
  begin
  end;

  procedure p1(a : Integer = 151; b : Integer = 151);
  begin
  end;

begin
end.
```

Depending on the desired result, there are different ways of approaching this problem. If only the last parameter is supposed to have the default value, then take the approach shown in the first example. If both parameters are supposed to have default values, then take the approach shown in the second example.

250. Default value required for '<name>'

Complete list of compiler error messages

```
program Produce;

  procedure p0(a : Integer = 151; b : Char);
  begin
  end;

  procedure p1(a : Integer = 151; b : Char);
  begin
  end;

begin
end.
```

The two procedures definitions shown above both will cause this error since they both declare a non-default parameter following a default value.

```
program Solve;

  procedure p0(a : Integer = 151; b : Char = 'A');
  begin
  end;

  procedure p1(b : Char; a : Integer = 151);
  begin
  end;

begin
end.
```

There are two ways of approaching this problem: add default values or rearrange the order of the parameters. As can be seen in the two example procedures above, both are simple to do.

251. Default parameter '<name>' must be by-value or const

Complete list of compiler error messages

Parameters which are given default values cannot be passed by reference.

```
program Produce;

  procedure p0(var x : Integer = 151);
  begin
  end;

begin
end.
```

Since the parameter x is passed by reference in this example, it cannot be given a default value.

```
program Solve;

  procedure p0(const x : Integer = 151);
  begin
  end;

begin
end.
```

In this solution, the by-reference parameter has been changed into a const parameter. Alternatively it could have been changed into a by-value parameter or the default value could have been removed.

252. Constant 0 converted to NIL

Complete list of compiler error messages

The Pascal compiler now allows the constant 0 to be used in pointer expressions in place of NIL. This change was made to allow older code to still compile with changes which were made in the low-level RTL.

```
program Produce;

  procedure p0(p : Pointer);
  begin
  end;

begin
  p0(0);
end.
```

In this example, the procedure p0 is declared to take a Pointer parameter yet the constant 0 is passed. The compiler will perform the necessary conversions internally, changing 0 into NIL, so that the code will function properly.

```
program Solve;

  procedure p0(p : Pointer);
  begin
  end;

begin
  p0(NIL);
end.
```

There are two approaches to solving this problem. In the case above the constant 0 has been replaced with NIL. Alternatively the procedure definition could be changed so that the parameter type is of Integer type.

253. $EXTERNALSYM and $NODEFINE not allowed for '<name>'; only global symbols

Complete list of compiler error messages

The $EXTERNALSYM and $NODEFINE directives can only be applied to global symbols.

254. $HPPEMIT '<text>' ignored

Complete list of compiler error messages

The $HPPEMIT directive can only appear after the unit header.

255. Integer and HRESULT interchanged

Complete list of compiler error messages

In Pascal Integer, Longint and HRESULT are compatible types, but in C++ the types are not compatible and will produce differently mangled C++ parameter names. To ensure that there will not be problems linking object files created with the Pascal compiler this message alerts you to possible problems. If you are compiling your source to an object file, this is an error otherwise it will be presented as a warning.

```
program Produce;
  uses Windows;

  type
    I0 = interface (IUnknown)
      procedure p0(var x : Integer);
    end;

    C0 = class (TInterfacedObject, I0)
      procedure p0(var x : HRESULT);
    end;

  procedure C0.p0(var x : HRESULT);
  begin
  end;

begin
end.
```

The example shown here declares the interface and class methods differently. While they are equivalent in Pascal they are not so in C++.

```
program Solve;

  uses Windows;

  type
    I0 = interface (IUnknown)
      procedure p0(var x : Integer);
    end;

    C0 = class (TInterfacedObject, I0)
      procedure p0(var x : Integer);
    end;

  procedure C0.p0(var x : Integer);
  begin
  end;

begin
end.
```

The easiest solution to this problem is to match the class-declared methods to be identical to the interface-declared methods.

256. C++ obj files must be generated (-jp)

Complete list of compiler error messages

Because of the language features used, standard C object files cannot be generated for this unit. You must generate C++ object files.

257. '<name>' is not the name of a unit

Complete list of compiler error messages

The $NOINCLUDE directive must be given a known Pascal unit name.

258. Expression needs no Initialize/Finalize

Complete list of compiler error messages

You have attempted to use the standard procedure Finalize on a Pascal type which requires no finalization.

```
program Produce;

  var
    ch : Char;

begin
  Finalize(ch);
end.
```

In this example, the Pascal type Char needs no finalization.

The usual solution to this problem is to remove the offending use of Finalize.

259. Pointer expression needs no Initialize/Finalize - need ^ operator?

Complete list of compiler error messages

You have attempted to finalize a Pointer type.

```
program Produce;

  var
    str : String;
    pstr : PString;

begin
  str := 'Sharene';
  pstr := @str;
  Finalize(pstr);  (*note: do not attempt to use 'str' after
this*)
end.
```

In this example the pointer, pstr, is passed to the Finalize procedure. This causes an hint since pointers do not require finalization.

```
program Solve;


  var
    str : String;
    pstr : PString;

begin
  str := 'Sharene';
  pstr := @str;
  Finalize(pstr^);  (*note: do not attempt to use 'str' after
this*)
end.
```

The solution to this problem is to apply the ^ operator to the pointer which is passed to the Finalization procedure.

260. Recursive include file <name>

Complete list of compiler error messages

The $I directive has been used to recursively include another file. You must check to make sure that all include files terminate without having cycles in them.

261. Need to specify at least one dimension for SetLength of dynamic array

Complete list of compiler error messages

The standard procedure `SetLength` has been called to alter the length of a dynamic array, but no array dimensions have been specified.

```
program Produce;

  var
    arr : array of integer;

begin
  SetLength(arr);
end.
```

The `SetLength` in the above example causes an error since no array dimensions have been specified.

```
program solve;

  var
    arr : array of integer;

begin
  SetLength(arr, 151);
end.
```

To remove this error from your program, specify the number of elements you wish the array to contain.

262. Cannot take the address when compiling to byte code

Complete list of compiler error messages

The *address-of* operator, `@`, cannot be used when compiling to byte code.

263. Cannot use old style object types when compiling to byte code

Complete list of compiler error messages

Old-style `Object` types are illegal when compiling to byte code.

264. Cannot use absolute variables when compiling to byte code

Complete list of compiler error messages

The use of absolute variables is prohibited when compiling to byte code.

265. There is no overloaded version of '<name>' that can be called with these arguments

Complete list of compiler error messages

An attempt has been made to call an overloaded function which cannot be resolved with the current set of overloads.

```
program Produce;

procedure f0(a : integer); overload;
begin
end;

procedure f0(a : char); overload;
begin
end;

begin
  f0(1.2);
end.
```

The overloaded procedure `f0` has two versions: one which takes a `char` and one which takes an `integer`. However, the call to `f0` uses a floating point type, which the compiler cannot resolve into neither a `char` nor an `integer`.

```
program Solve;

procedure f0(a : integer); overload;
begin
end;

procedure f0(a : char); overload;
begin
end;

begin
  f0(1);
end.
```

There are two basic ways to solve this problem: either supply a parameter type which can be resolved into a match of an overloaded procedure, or create a new version of the overloaded procedure which matches the parameter type.

In the example above, the parameter type has been modified to match one of the existing overloaded versions of `f0`.

266. Ambiguous overloaded call to '<name>'

Complete list of compiler error messages

Based on the current overload list for the specified function, and the programmed invocation, the compiler is unable to determine which version of the procedure should be invoked.

```
program Produce;

procedure f0(a : integer); overload;
begin
end;

procedure f0(a : integer; b : char = 'A'); overload;
begin
end;

begin
  f0(1);
end.
```

In this example, the default parameter that exists in one of the versions of `f0` makes it impossible for the compiler to determine which procedure should actually be called.

```
program Solve;

procedure f0(a : integer); overload;
begin
end;

procedure f0(a : integer; b : char); overload;
begin
end;

begin
  f0(1);
end.
```

The approach taken in this example was to remove the default parameter value. The result here is that the procedure taking only one `integer` parameter will be called. It should be noted that this approach is the only way that the single-parameter function can be called.

267. Method '<name>' with identical parameters exists already

Complete list of compiler error messages

A method with an identical signature already exists in the data type.

```
program Produce;

  type
    t0 = class
      procedure f0(a : integer); overload;
      procedure f0(a : integer); overload;
    end;

procedure T0.f0(a : integer);
begin
end;

begin
end.
```

The error is produced here because there are two overloaded declarations for the same procedure.

```
program Solve;

  type
    t0 = class
      procedure f0(a : integer); overload;
      procedure f0(a : char); overload;
    end;

procedure T0.f0(a : integer);
begin
end;

procedure T0.f0(a : char);
begin
end;

begin
end.
```

There are different approaches to curing this error. One approach is to remove the redundant declaration of the procedure. Another approach, taken here, is to change the parameter type of the duplicate declarations so that it creates a unique version of the overloaded procedure.

268. Ancestor type '<name>' does not have default constructor

Complete list of compiler error messages

The ancestor of the class being compiled does not have a default constructor. This error only occurs with the byte code version of the compiler.

269. Overloaded procedure '<name>' must be marked with the 'overload' directive

Complete list of compiler error messages

The compiler has encountered a procedure, which is not marked `overload`, with the same name as a procedure already marked `overload`. All overloaded procedures must be marked as such.

```
program Produce;

procedure f0(a : integer); overload;
begin
end;

procedure f0(a : integer; ch : char);
begin
end;

begin
end.
```

The procedure `f0(a : integer; ch : char)` causes the error since it is not marked with the `overload` keyword.

```
program solve;

procedure f0(a : integer); overload;
begin
end;

procedure f0(a : integer; ch : char); overload;
begin
end;

begin
end.
```

If the procedure is intended to be an overloaded version, then mark it as `overload`. If it is not intended to be an overloaded version, then change its name.

270. Class methods not allowed as property getters or setters

Complete list of compiler error messages

The compiler has encountered a property declaration which specified a "class methods" as its getter or setter method. These special method types have different semantics, such as not being able to access instance data, and cannot be used for property accessors.

```
program Produce;

  type
    T0 = class
      ch : char;
      class procedure access(a : char);
      property CharValue : Char read ch write access;
    end;

class procedure T0.access(a : char);
begin
end;

begin
end.
```

The solution to this problem is to change your program so that it does not use class methods as property accessors.

271. New not supported for dynamic arrays - use SetLength

Complete list of compiler error messages

The program has attempted to use the standard procedure `NEW` on a dynamic array. The proper method for allocating dynamic arrays is to use the standard procedure `SetLength`.

```
program Produce;
  var
    arr : array of integer;

begin
  new(arr, 10);
end.
```

The standard procedure `NEW` cannot be used on dynamic arrays.

```
program Solve;
  var
    arr : array of integer;

begin
  SetLength(arr, 10);
end.
```

Use the standard procedure `SetLength` to allocate dynamic arrays.

272. Dispose not supported (nor necessary) for dynamic arrays

Complete list of compiler error messages

The compiler has encountered a use of the standard procedure `DISPOSE` on a dynamic array. Dynamic arrays are reference counted and will automatically free themselves when there are no longer any references to them.

```
program Produce;
  var
    arr : array of integer;

begin
  SetLength(arr, 10);
  Dispose(arr);
end.
```

The use of `DISPOSE` on the dynamic array `arr` causes the error in this example.

```
program Produce;
  var
    arr : array of integer;

begin
  SetLength(arr, 10);
end.
```

The only solution here is to remove the offending use of `DISPOSE`

273. Duplicate implements clause for interface <name>

Complete list of compiler error messages

The compiler has encountered two different `property` declarations which claim to `implement` the same `interface`. An `interface` may be implemented by only one `property`.

```
program Produce;
type
  IMyInterface = interface
  end;

  TMyClass = class(TInterfacedObject, IMyInterface)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface read FMyInterface
implements IMyInterface;
    property OtherInterface: IMyInterface read FMyInterface
implements IMyInterface;
  end;
end.
```

Both `MyInterface` and `OtherInterface` attempt to implement `IMyInterface`. Only one property may implement the chosen interface.

The only solution in this case is to remove one of the offending `implements` clauses.

274. Implements clause only allowed within class types

Complete list of compiler error messages

```
program Produce;
type
  IMyInterface = interface
    function getter : IMyInterface;
    property MyInterface: IMyInterface read getter implements
IMyInterface;
  end;
end.
```

The `interface` definition in this example attempt to use an implements clause which causes the error.

```
program Solve;
type
  IMyInterface = interface
    function getter : IMyInterface;
    property MyInterface: IMyInterface read getter;
  end;
end.
```

The only viable solution to this problem is to remove the offending `implements` clause.

275. Implements clause only allowed for properties of class or interface type

Complete list of compiler error messages

An attempt has been made to use the `implements` clause with an improper type. Only `class` or `interface` types may be used.

```
program Produce;
type
  TMyClass = class(TInterfacedObject)
    FInteger : Integer;
    property MyInterface: Integer read FInteger implements
Integer;
  end;
end.
```

In this example the error is caused because an `Integer` type is used with an `implements` clause.

The only solution for this error is to correct the `implements` clause so that it refers to a `class` or `interface` type, or to remove the offending clause altogether.

276. Implements clause not allowed together with index clause

Complete list of compiler error messages

277. Implements clause only allowed for readable property

Complete list of compiler error messages

The compiler has encountered a "write only" property that claims to `implement` an `interface`. A `property` must be read/write to use the `implements` clause.

```
program Produce;
type
  IMyInterface = interface
  end;

  TMyClass = class(TInterfacedObject, IMyInterface)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface implements IMyInterface;
  end;
end.
```

The property in this example is *write only* and cannot be used to implement an interface.

```
program Solve;
type
  IMyInterface = interface
  end;

  TMyClass = class(TInterfacedObject, IMyInterface)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface read FMyInterface
implements IMyInterface;
  end;
end.
```

by adding a `read` clause, the property can use the `implements` clause.

278. Implements getter must be register calling convention

Complete list of compiler error messages

The compiler has encountered a getter or setter which does not have `register` calling convention.

```
program Produce;
type
  I0 = interface
  end;

  T0 = class(TInterfacedObject, I0)
    function getter : I0; cdecl;
    property p0 : I0 read getter implements I0;
  end;

function T0.getter : I0;
begin
end;
end.
```

As can be seen in this example, the `cdecl` on the function `getter` causes this error to be produced.

```
program Solve;
type
  I0 = interface
  end;

  T0 = class(TInterfacedObject, I0)
    function getter : I0;
    property p0 : I0 read getter implements I0;
  end;

function T0.getter : I0;
begin
end;
end.
```

The only solution to this problem is to remove the offending *calling convention* from the property getter declaration.

279. Implements getter cannot be dynamic or message method

Complete list of compiler error messages

An attempt has been made to use a `dynamic` or `message` method as a property accessor of a property which has an `implements` clause.

```
program Produce;
type
  I0 = interface
  end;

  T0 = class(TInterfacedObject, I0)
    function getter : I0; dynamic;
    property p0 : I0 read getter implements I0;
  end;

function T0.getter : I0;
begin
end;

end.
```

As shown in the example here, it is an error to use the `dynamic` modifier on a getter for a property which has an `implements` clause.

```
program Produce;
type
  I0 = interface
  end;

  T0 = class(TInterfacedObject, I0)
    function getter : I0;
    property p0 : I0 read getter implements I0;
  end;

function T0.getter : I0;
begin
end;

end.
```

To remove this error from your programs, remove the offending `dynamic` or `method` declaration.

280. Cannot have method resolutions for interface '<name>'

Complete list of compiler error messages

An attempt has been made to use a method resolution clause for an interface named in an `implements` clause.

```
program Produce;
type
  I0 = interface
    procedure i0p0(a : char);
  end;

  T0 = class(TInterfacedObject, I0)
    procedure I0.i0p0 = proc0;
    function getter : I0;
    procedure proc0(a : char);
    property p0 : I0 read getter implements I0;
  end;

procedure T0.proc0(a : char);
begin
end;

function T0.getter : I0;
begin
end;
end.
```

In this example the method `proc0` is mapped onto the interface procedure `i0p0`, but because the interface is mentioned in a `implements` clause, this renaming is not allowed.

```
program Solve;
type
  I0 = interface
    procedure i0p0(a : char);
  end;

  T0 = class(TInterfacedObject, I0)
    function getter : I0;
    procedure i0p0(a : char);
    property p0 : I0 read getter implements I0;
  end;

procedure T0.i0p0(a : char);
begin
end;

function T0.getter : I0;
begin
end;
end.
```

The solution for this error is to remove the offending "name resolution clause". One easy way to accomplish this is to name the procedure in the class to the same name as the interface method.

281. Interface '<name>' not mentioned in interface list

Complete list of compiler error messages

An `implements` clause references an interface which is not mentioned in the interface list of the class.

```
program Produce;
type
  IMyInterface = interface
  end;

  TMyClass = class(TInterfacedObject, IUnknown)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface read FMyInterface
implements IMyInterface;
  end;
end.
```

The example shown here uses `implements` with the `IMyInterface` interface, but it is not mentioned in the interface list.

```
program Solve;
type
  IMyInterface = interface
  end;

  TMyClass = class(TInterfacedObject, IUnknown, IMyInterface)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface read FMyInterface
implements IMyInterface;
  end;
end.
```

A quick solution, shown here, is to add the required interface to the interface list of the class definition. Of course, adding it to the interface list might require the implementation of the methods of the interface.

282. Exported package threadvar '<name>.<name>' cannot be used outside of this package

Complete list of compiler error messages

Windows does not support the exporting of `threadvar` variables from a DLL, but since using Delphi packages is meant to be semantically equivalent to compiling a project without them, the Pascal compiler must somehow attempt to support this construct.

This warning is to notify you that you have included a unit which contains a `threadvar` in an interface into a package. While this is not illegal, you will not be able to access the variable from a unit outside the package.

Attempting to access this variable may appear to succeed, but it actually did not.

A solution to this warning is to move the `threadvar` to the `implementation` section and provide function which will retrieve the variables value.

283. Only one of a set of overloaded methods can be published

Complete list of compiler error messages

Only one member of a set of overloaded functions may be published because the RTTI generated for procedures only contains the name.

```
(*$M+*)
(*$APPTYPE CONSOLE*)
program Produce;
type
  Base = class
  published
    procedure p1(a : integer); overload;
    procedure p1(a : boolean); overload;
  end;

  Extended = class (Base)
    procedure e1(a : integer); overload;
    procedure e1(a : boolean); overload;
  end;

  procedure Base.p1(a : integer);
  begin
  end;

  procedure Base.p1(a : boolean);
  begin
  end;

  procedure Extended.e1(a : integer);
  begin
  end;

  procedure Extended.e1(a : boolean);
  begin
  end;

end.
```

In the example shown here, both overloaded `p1` functions are contained in a `published` section, which is not allowed.

Further, since the `$M+` state is used, the `Extended` class starts with `published` visibility, thus the error will also appear for this class also.

```
(*$M+*)
(*$APPTYPE CONSOLE*)
program Solve;
type
  Base = class
  public
    procedure p1(a : integer); overload;
  published
    procedure p1(a : boolean); overload;
  end;

  Extended = class (Base)
  public
    procedure e1(a : integer); overload;
    procedure e1(a : boolean); overload;
  end;

  procedure Base.p1(a : integer);
  begin
  end;

  procedure Base.p1(a : boolean);
  begin
  end;

  procedure Extended.e1(a : integer);
  begin
  end;

  procedure Extended.e1(a : boolean);
  begin
  end;

end.
```

The solution here is to ensure that no more than one member of a set of overloaded function appears in a `published` section. The easiest way to achieve this is to change the visibility to `public`, `protected` or `private`; whichever is most appropriate.

284. Previous declaration of '<name>' was not marked with the 'overload' directive

Complete list of compiler error messages

```
program Produce;
type
  Base = class
    procedure func(a : integer);
    procedure func(a : char); overload;
  end;

  procedure Base.func(a : integer);
  begin
  end;

  procedure Base.func(a : char);
  begin
  end;

end.
```

This example attempts to overload the `char` version of `func` without marking the first version of `func` as overloadable.

You must mark all functions to be overloaded with the `overload` directive. If `overload` were not required on all versions it would be possible to introduce a new method which overloads an existing method and then a simple recompilation of the source could produce different behavior.

```
program Solve;
type
  Base = class
    procedure func(a : integer); overload;
    procedure func(a : char); overload;
  end;

  procedure Base.func(a : integer);
  begin
  end;

  procedure Base.func(a : char);
  begin
  end;

end.
```

There are two solutions to this problem. You can either remove the attempt at overloading or you can mark the original declaration with the `overload` directive. The example shown above marks the original declaration.

285. Parameters of this type cannot have default values

Complete list of compiler error messages

The default parameter mechanism incorporated into the Delphi Pascal compiler allows only simple types to be initialized in this manner. You have attempted to use a type that is not supported.

```
program Produce;
type
  ArrayType = array [0..1] of integer;

  procedure p1(proc : ArrayType = [1, 2]);
  begin
  end;
end.
```

Default parameters of this type are not supported in Delphi Pascal.

```
program solve;
type
  ArrayType = array [0..1] of integer;

  procedure p1(proc : ArrayType);
  begin
  end;

end.
```

The only way to get rid of this error is to remove the offending parameter assignment or to change the type of the parameter to one that can be initialized with a default value.

286. Overriding virtual method '<class>.<method>' has a lower visibility than base class

Complete list of compiler error messages

The method named in the error message has been declared as an override of a virtual method in a base class, but the visibility in the current class is lower than that used in the base class for the same method.

While the visibility rules of Pascal would seem to indicate that the function cannot be seen, the rules of invoking virtual functions will cause the function to be properly invoked through a virtual call.

Generally this means that the method of the derived class was declared in a `private` or `protected` section while the method of the base class was declared in a `protected` or `pubic` (including `published`) section respectively.

```
unit Produce;
interface

  type
    Base = class(TObject)
    public
      procedure VirtualProcedure(X: Integer); virtual;
    end;

    Extended = class(Base)
    protected
      procedure VirtualProcedure(X: Integer); override;
    end;

implementation

  procedure Base.VirtualProcedure(X: Integer);
  begin
  end;

  procedure Extended.VirtualProcedure(X: Integer);
  begin
  end;
end.
```

The example above aptly shows how this error is produced by putting `Extended.VirtualProcedure` into the `protected` section.

In practice this is never harmful, but it can be confusing to someone reading documentation and observing the visibility attributes of the document.

This hint will only be produced for classes appearing in the interface section of a unit.

```
unit Solve;
interface

  type
    Base = class(TObject)
    public
      procedure VirtualProcedure(X: Integer); virtual;
    end;

    Extended = class(Base)
    public
      procedure VirtualProcedure(X: Integer); override;
    end;

implementation

  procedure Base.VirtualProcedure(X: Integer);
  begin
  end;

  procedure Extended.VirtualProcedure(X: Integer);
  begin
  end;
end.
```

There are three basic solutions to this problem.

1. Ignore the hint
2. Change the visibility to match the base class
3. Move class definition to the implementation section.

The example program above has taken the approach of changing the visibility to match the base class.

287. Published property getters and setters must have register calling convention

Complete list of compiler error messages

A property appearing in a `published` section has a *getter* or *setter* procedure that does not have the `register` calling convention.

```
unit Produce;
interface
  type
    Base = class
    public
      function getter : Integer; cdecl;
    published
      property Value : Integer read getter;
    end;

implementation
function Base.getter : Integer;
begin getter := 0;
end;

end.
```

This example declares the getter function `getter` for the `published` property `Value` to be of `cdecl` calling convention, which produces the error.

```
unit Solve;
interface
  type
    Base = class
    public
      function getter : Integer;
    published
      property Value : Integer read getter;
    end;

implementation
function Base.getter : Integer;
begin getter := 0;
end;

end.
```

The only solution to this problem is to declare the getter function to be of `register` calling convention, which is the default. As can be seen in this example, no calling convention is specified.

288. Property getters and setters cannot be overloaded

Complete list of compiler error messages

A property has specified an overloaded procedure as either its *getter* or *setter*.

```
unit Produce;
interface
  type
    Base = class
    public
      function getter : Integer; overload;
      function getter(a : char) : Integer; overload;
      property Value : Integer read getter;
    end;

implementation
function Base.getter : Integer;
begin getter := 0;
end;

function Base.getter(a : char) : Integer;
begin
end;

end.
```

The overloaded function `getter` in the above example will cause this error to be produced.

```
unit Solve;
interface
  type
    Base = class
    public
      function getter : Integer;
      property Value : Integer read getter;
    end;

implementation
function Base.getter : Integer;
begin getter := 0;
end;

end.
```

The only solution when this problem occurs is to remove the offending `overload` specifications, as is shown in the above example.

289. Comparing signed and unsigned types - widened both operands

Complete list of compiler error messages

To compare signed and unsigned types correctly the compiler must promote both operands to the next larger size data type.

To see why this is necessary, consider two operands, a `Shortint` with the value -128 and a `Byte` with the value 130. The `Byte` type has one more digit of precision than the `Shortint` type, and thus comparing the two values cannot accurately be performed in only 8 bits. The proper solution for the compiler is to promote both these types to a larger, common, size and then to perform the comparison.

```
program Produce;
  var
    s : shortint;
    b : byte;

begin
  s := -128;
  b := 130;

  assert(b < s);
end.
```

290. Combining signed and unsigned types - widened both operands

Complete list of compiler error messages

To mathematically combine signed and unsigned types correctly the compiler must promote both operands to the next larger size data type and then perform the combination.

To see why this is necessary, consider two operands, an `Integer` with the value -128 and a `Cardinal` with the value 130. The `Cardinal` type has one more digit of precision than the `Integer` type, and thus comparing the two values cannot accurately be performed in only 32 bits. The proper solution for the compiler is to promote both these types to a larger, common, size and then to perform the comparison.

The compiler will only produce this warning when the size is extended beyond what would normally be used for calculating the result.

```
{$APPTYPE CONSOLE}
program Produce;
  var
    i : Integer;
    c : Cardinal;

begin
  i := -128;
  c := 130;
  WriteLn(i + c);
end.
```

In the example above, the compiler warns that the expression will be calculated at 64 bits rather than the supposed 32 bits.

291. Duplicate <text> <name> with identical parameters will be inaccessible from C++

Complete list of compiler error messages

An object file is being generated and Two, differently named, constructors or destructors with identical parameter lists have been created; they will be inaccessible if the code is translated to an `HPP` file because constructor and destructor names are converted to the `class` name. In C++ these duplicate declarations will appear to be the same function.

```
unit Produce;
interface
  type
    Base = class
      constructor ctor0(a, b, c : integer);
      constructor ctor1(a, b, c : integer);
    end;

implementation
constructor Base.ctor0(a, b, c : integer);
begin
end;

constructor Base.ctor1(a, b, c : integer);
begin
end;

begin
end.
```

As can be seen in this example, the two constructors have the same signature and thus, when the file is compiled with one of the `-j` options, will produce this warning.

```
unit Solve;
interface
  type
    Base = class
      constructor ctor0(a, b, c : integer);
      constructor ctor1(a, b, c : integer; dummy : integer = 0);
    end;

implementation
constructor Base.ctor0(a, b, c : integer);
begin
end;

constructor Base.ctor1(a, b, c : integer; dummy : integer);
begin
end;

begin
end.
```

A simple method to solve this problem is to change the signature of one of constructors, for example, to add an extra parameter. In the example above, a default parameter has been added to `ctor1`. This method of approaching this error has the benefit that Pascal code using `ctor1`

does not need to be changed. C++ code, on the other hand, will have to specify the extra parameter to allow the compiler to determine which constructor is desired.

292. Comparison always evaluates to False

Complete list of compiler error messages

The compiler has determined that the expression will always evaluate to `False`. This most often can be the result of a boundary test against a specific variable type, for example, a `Integer` against `$80000000`. In versions of the Pascal compiler prior to 12.0, the hexadecimal constant `$80000000` would have been a negative `Integer` value, but with the introduction of the `int64` type, this same constant now becomes a positive `int64` type. As a result, comparisons of this constant against `Integer` variables will no longer behave as they once did.

As this is a warning rather than an error, there is no standard method of addressing the problems: sometimes the warning can be ignored, sometimes the code must be rewritten.

```
program Produce;

  var
    i : Integer;
    c : Cardinal;

begin
  c := 0;
  i := 0;
  if c < 0 then
    WriteLn('false');

  if i >= $80000000 then
    WriteLn('false');
end.
```

Here the compiler determines that the two expressions will always be `False`. In the first case, a `Cardinal`, which is unsigned, can never be less than 0. In the second case, a 32-bit `Integer` value can never be larger than, or even equal to, an `int64` value of `$80000000`.

293. Comparison always evaluates to True

Complete list of compiler error messages

The compiler has determined that the expression will always evaluate to `True`. This most often can be the result of a boundary test against a specific variable type, for example, a `Integer` against `$80000000`.

In versions of the Pascal compiler prior to 12.0, the hexadecimal constant `$80000000` would have been a negative `Integer` value, but with the introduction of the `int64` type, this same constant now becomes a positive `int64` type. As a result, comparisons of this constant against `Integer` variables will no longer behave as they once did.

As this is a warning rather than an error, there is no standard method of addressing the problems: sometimes the warning can be ignored, sometimes the code must be rewritten.

```
program Produce;

  var
    i : Integer;
    c : Cardinal;

begin
  c := 0;
  i := 0;
  if c >= 0 then
    WriteLn('true');

  if i < $80000000 then
    WriteLn('true');
end.
```

Here the compiler determines that the two expressions will always be `True`. In the first case, a `Cardinal`, which is unsigned, will always be greater or equal to 0. In the second case, a 32-bit `Integer` value will always be smaller than an `int64` value of `$80000000`.

294. Cannot use reserved unit name <name>

Complete list of compiler error messages

An attempt has been made to use one of the reserved unit names, such as `System`, as the name of a user-created unit.

The names in the following list are currently reserved by the compiler.

- System
- SysInit

```
unit System;
interface
implementation
begin
end.
```

The name of the unit in this example is illegal because it is reserved for use by the compiler.

```
unit MySystem;
interface
implementation
begin
end.
```

The only solution to this problem is to use a different name for the unit.

295. No overloaded version of '<name>' with this parameter list exists

Complete list of compiler error messages

An attempt has been made to call an overloaded procedure but no suitable match could be found.

```
program overload;
  procedure f(x : Char); overload;
  begin
  end;

  procedure f(x : Integer); overload;
  begin
  end;

begin
  f(1.0);

end.
```

In the use of `f` presented here, the compiler is unable to find a suitable match (using the type compatibility & *overloading* rules) given the actual parameter `1.0`.

```
program overload;
  procedure f(x : char); overload;
  begin
  end;

  procedure f(x : integer); overload;
  begin
  end;

begin
  f(1);
end.
```

Here, the call to `f` has been changed to pass an integer as the actual parameter which will allow the compiler to find a suitable match. Another approach to solving this problem would be to introduce a new procedure which takes a floating point parameter.

296. property attribute 'label' cannot be used in dispinterface

Complete list of compiler error messages

You have added a label to a property defined in a dispinterface, but this is disallowed by the language definition.

```
program Problem;

  type
    T0 = dispinterface
      ['{15101510-1510-1510-1510-151015101510}']
      function R : Integer;
      property value : Integer label 'Key';
    end;

begin
end.
```

Here an attempt is made to use a `label` attribute on a `dispinterface` property.

```
program Solve;

  type
    T0 = dispinterface
      ['{15101510-1510-1510-1510-151015101510}']
      function R : Integer;
      property value : Integer;
    end;

begin
end.
```

the only solution to this problem is to remove `label` attribute from the property definition.

297. property attribute 'label' cannot be an empty string

Complete list of compiler error messages

```
unit Problem;
interface
  type
    T0 = class
      f : integer;
      property g : integer read f write f label '';
    end;

implementation
begin
end.
```

The error is output because the `label` attribute for `g` is an empty string.

```
unit Solve;
interface
  type
    T0 = class
      f : integer;
      property g : integer read f write f label 'LabelText';
    end;

implementation
begin
end.
```

In this solution, the `label` attribute has been replaced by a non-zero length string.