Visual Component Library Reference

Delphi for Windows

Introduction

Copyright Agreement

This manual is a reference for the Delphi Visual Component Library (VCL) and the Delphi run-time library. Use it when you want to look up the details of a particular VCL object, component, variable, property, method, event, routine, or type and find out how to use it.

Note See online Help for documentation of the Object Pascal Language Definition and Reference.

Manual conventions

The printed manuals for Delphi use the typefaces and symbols described in Table Intro.1 to indicate special text.

Table Intro.1 Typefaces and symbols in these manuals

Typeface or symbol	Meaning
`Monospace type`	Monospaced text represents text as it appears onscreen or in Object Pascal code. It also represents anything you must type.
Boldface	Boldfaced words in text or code listings represent Object Pascal reserved words or compiler directives.
Italics	Italicized words in text represent Object Pascal identifiers, such as variable or type names. Italics are also used to emphasize certain words, such as new terms.
Keycaps	This typeface indicates a key on your keyboard. For example, "Press Esc to exit a menu."
🔑	This symbol indicates a key, or important property, method, or event.
▷	This symbol indicates a run-time only property, method or event.

Contacting Borland

The Borland Assist program offers a range of technical support plans to fit the different needs of individuals, consultants, large corporations, and developers. To receive help with this product send in the registration card and select the Borland Assist plan that best suits your needs. North American customers can register by phone 24 hours a day by calling 1-800-845-0147. For additional details on these and other Borland services, see the Borland Assist Support and Services Guide included with this product.

Delphi Visual Component Library

The VCL is made up of objects, most of which are also components. Using the objects and components of VCL, you are unlimited in the range of Windows programs you can develop rapidly. Delphi itself was built using VCL.

Delphi objects contain both code and data. The data is stored in the fields and properties of the objects, and the code is made up of methods that act upon the field and property values. All objects descend from the ancestor object *TObject*.

Components are visual objects that you can manipulate at design time. All components descend from the *TComponent* object. To program with a component, this is the model you will use most frequently:

1. Select a component from Delphi's Component palette and add it to a form.
2. Set property values of the component using the Object Inspector.
3. Respond to events that might occur to the component at run time. To respond to an event, you write code within an event handler. Your code modifies property values and calls methods.

For detailed information on how to perform these three steps, see the *Delphi User's Guide*.

You can create your own objects and components by deriving them from the existing Delphi objects and components. For information about writing your own components, see the *Delphi Component Writer's Guide*.

Visual Component Library objects

Objects are the fundamental elements of the VCL. In fact, all components and controls are based on objects.

Objects differ from controls in that you can access them only at run time. Unlike most components, objects do not appear on the Component palette. Instead, a default instance variable is declared in the unit of the object or you have to declare one yourself.

For example, the *Clipboard* variable is declared in the *Clipbrd* unit. To use a *TClipboard* object, add the *Clipbrd* unit to the **uses** clause of the unit, then refer to the *Clipboard* variable. However, to use a *TBitmap* object, add the *Graphics* unit to the **uses** clause of the unit, then execute the following code at run time to declare an instance variable:

```
var
  Bitmap1: TBitmap;
begin
  Bitmap1 := TBitmap.Create;
end;
```

Note The memory allocated for objects that you explicitly declare should be released when you are finished with the object. For example, call the *Free* method of the bitmap:

```
Bitmap1.Free;
```

The properties, methods, and events that all objects have in common are inherited from an abstract object type called *TObject*. You need to understand the internal details of *TObject* only if you are creating a new object based on *TObject*.

The following is a list of all objects in the VCL that directly descend from *TObject*.:

Table Intro.2 VCL objects

TBitmap	*TGraphic*	*TOutlineNode*
TBlobStream	*TGraphicsObject*	*TParam*
TBrush	*TIcon*	*TParams*
TCanvas	*TIndexDef*	*TPen*
TClipboard	*TIndexDefs*	*TPicture*
TControlScrollBar	*TIniFile*	*TPrinter*
TFieldDef	*TList*	*TStringList*
TFieldDefs	*TMetafile*	*TStrings*
TFont	*TOLEDropNotify*	

Note In addition to these objects, all VCL components also descend from *TObject*, although not directly.

The *TObject* object introduces the following methods that all objects and components inherit:

Table Intro.3 Object methods

ClassName	*ClassType*	*Destroy*
ClassParent	*Create*	*Free*

Visual Component Library components

Components are the building blocks of Delphi applications. You build an application by adding components to forms and using the properties, methods, and events of the components.

The properties, methods, and events that all components have in common are inherited from an abstract component type called *TComponent*. You need to understand the internal details of *TComponent* only if you are writing a component based on *TComponent*.

The following is a list of all components in the VCL:

Table Intro.4 VCL components

TApplication	*TDDEClientItem*	*TOutline*
TBatchMove	*TDDEServerConv*	*TPaintBox*
TBCDField	*TDDEServerItem*	*TPanel*
TBevel	*TDirectoryListBox*	*TPopupMenu*
TBitBtn	*TDrawGrid*	*TPrintDialog*
TBlobField	*TDriveComboBox*	*TPrinterSetupDialog*
TBooleanField	*TEdit*	*TQuery*

Table Intro.4 VCL components (continued)

TButton	*TField*	*TRadioButton*
TBytesField	*TFileListBox*	*TRadioGroup*
TCheckBox	*TFilterComboBox*	*TReplaceDialog*
TColorDialog	*TFindDialog*	*TReport*
TComboBox	*TFloatField*	*TSaveDialog*
TCurrencyField	*TFontDialog*	*TScreen*
TDatabase	*TForm*	*TScrollBar*
TDataSource	*TGraphicField*	*TScrollBox*
TDateField	*TGroupBox*	*TSession*
TDateTimeField	*THeader*	*TShape*
TDBCheckBox	*TImage*	*TSmallIntField*
TDBComboBox	*TIntegerField*	*TSpeedButton*
TDBEdit	*TLabel*	*TStoredProc*
TDBGrid	*TListBox*	*TStringField*
TDBImage	*TMainMenu*	*TStringGrid*
TDBListBox	*TMaskEdit*	*TTabbedNotebook*
TDBLookupCombo	*TMediaPlayer*	*TTable*
TDBLookupList	*TMemo*	*TTabSet*
TDBMemo	*TMemoField*	*TTimeField*
TDBNavigator	*TMenuItem*	*TTimer*
TDBRadioGroup	*TNotebook*	*TVarBytesField*
TDBText	*TOLEContainer*	*TWordField*
TDDEClientConv	*TOpenDialog*	

Most components are available from the Component palette. You will not find the following components on the Component palette, however:

Table Intro.5 Components not on the Component palette

TApplication	*TDateTimeField*	*TScreen*
TBCDField	*TField*	*TSession*
TBlobField	*TFloatField*	*TSmallIntField*
TBooleanField	*TGraphicField*	*TStringField*
TBytesField	*TIntegerField*	*TTimeField*
TCurrencyField	*TMemoField*	*TVarBytesField*
TDateField	*TMenuItem*	*TWordField*

The *TComponent* component introduces the following properties that all components inherit:

Table Intro.6 Component properties

▷	*ComponentCount*	▷	*Components*	▷	*Owner*
▷	*ComponentIndex*		*Name*		*Tag*

In addition to the methods components inherit from the *TObject* object, the *TComponent* component introduces the following:

Table Intro.7 Component methods

FindComponent	*InsertComponent*	*RemoveComponent*

Visual Component Library controls

Controls are visual components; that is, they are components you can see when your application is running. All controls have properties in common that specify the visual attributes of controls, such as *Left*, *Top*, *Height*, *Width*, *Cursor*, and *Hint*.

The properties, methods, and events that all controls have in common are inherited from an abstract component type called *TControl*. You need to understand the internal details of *TControl* only if you are writing a component based on *TControl*.

The following is a list of all controls in the VCL.

Table Intro.8 VCL controls

TBevel	*TDBText*	*TNotebook*
TBitBtn	*TDirectoryListBox*	*TOLEContainer*
TButton	*TDrawGrid*	*TOutline*
TCheckBox	*TDriveComboBox*	*TPaintBox*
TComboBox	*TEdit*	*TPanel*
TDBCheckBox	*TFileListBox*	*TRadioButton*
TDBComboBox	*TFilterComboBox*	*TRadioGroup*
TDBEdit	*TForm*	*TScrollBar*
TDBGrid	*TGroupBox*	*TScrollBox*
TDBImage	*THeader*	*TShape*
TDBListBox	*TImage*	*TSpeedButton*
TDBLookupCombo	*TLabel*	*TStringGrid*
TDBLookupList	*TListBox*	*TTabbedNotebook*
TDBMemo	*TMaskEdit*	*TTabSet*
TDBNavigator	*TMediaPlayer*	
TDBRadioGroup	*TMemo*	

In addition to the properties controls inherit from the *TComponent* component, the *TControl* component introduces the following:

Table Intro.9 Control properties

▷	*Align*		*Cursor*	▷	*ShowHint*
▷	*BoundsRect*	▷	*Enabled*		*Top*
▷	*ClientHeight*		*Height*	▷	*Visible*
▷	*ClientOrigin*		*Hint*		*Width*
▷	*ClientRect*		*Left*		
▷	*ClientWidth*	▷	*Parent*		

In addition to the methods controls inherit from the *TComponent* component, the *TControl* component introduces the following methods:

Table Intro.10 Control methods

BeginDrag	*GetTextBuf*	*Repaint*
BringToFront	*GetTextLen*	*ScreenToClient*
ClientToScreen	*Hide*	*SetBounds*
ControlAtPos	*Invalidate*	*SetTextBuf*
Dragging	*Refresh*	*Show*
EndDrag	*SendToBack*	*Update*

Visual Component Library windowed controls

Windowed controls are controls that:

- Can receive focus while your application is running
- Can contain other controls
- Have a window handle

All windowed controls have properties in common that specify their focus attributes, such as *HelpContext*, *TabStop*, and *TabOrder*. Windowed controls also provide the *OnEnter* and *OnExit* events.

The properties, methods, and events that all windowed controls have in common are inherited from an abstract component type called *TWinControl*. You need to understand the internal details of *TWinControl* only if you are writing a component based on *TWinControl*.

The following is a list of all windowed controls in the VCL:

Table Intro.11 VCL windowed controls

TBitBtn	*TDBNavigator*	*TMediaPlayer*
TButton	*TDBRadioGroup*	*TMemo*
TCheckBox	*TDirectoryListBox*	*TNotebook*
TComboBox	*TDrawGrid*	*TOLEContainer*
TDBCheckBox	*TDriveComboBox*	*TOutline*
TDBComboBox	*TEdit*	*TPanel*
TDBEdit	*TFileListBox*	*TRadioButton*
TDBGrid	*TFilterComboBox*	*TRadioGroup*
TDBImage	*TForm*	*TScrollBar*
TDBListBox	*TGroupBox*	*TScrollBox*
TDBLookupCombo	*THeader*	*TStringGrid*
TDBLookupList	*TListBox*	*TTabbedNotebook*
TDBMemo	*TMaskEdit*	*TTabSet*

In addition to the properties windowed controls inherit from the *TControl* component, the *TWinControl* component introduces the following properties:

Table Intro.12 Windowed control properties

▷ *Brush*	▷ *Handle*	▷ *TabOrder*
▷ *Controls*	*HelpContext*	▷ *TabStop*
▷ *ControlCount*	▷ *Showing*	

In addition to the methods windowed controls inherit from the *TControl* component, the *TWinControl* component introduces the following methods:

Table Intro.13 Windowed control methods

CanFocus	*Focused*	*RemoveControl*
ClientOrigin	*HandleAllocated*	*ScaleBy*
Create	*HandleNeeded*	*ScrollBy*
Destroy	*InsertControl*	*SetFocus*

The *TWinControl* component introduces the following events:

Table Intro.14 Windowed control events

OnEnter	*OnExit*

Visual Component Library nonwindowed controls

Nonwindowed controls are controls that:

- Cannot receive focus while your application is running
- Cannot contain other controls
- Do not have a window handle

The properties, methods, and events that all windowed controls have in common are inherited from an abstract component type called *TGraphicControl*. You need to understand the internal details of *TGraphicControl* only if you are writing a component based on *TGraphicControl*.

The following is a list of all nonwindowed controls in the VCL:

Table Intro.15 VCL nonwindowed controls

TBevel	*TLabel*	*TSpeedButton*
TDBText	*TPaintBox*	
TImage	*TShape*	

Visual Component Library procedures and functions

These procedures and functions are part of the VCL, but they aren't methods of any components or objects. They are categorized here by how they are used.

The following routines are used to display messages in dialog boxes:

Table Intro.16 Message dialog box routines

InputBox	*MessageDlg*
InputQuery	*MessageDlgPos*

The following routines are used to define menu command short cuts.

Table Intro.17 Menu shortcut routines

ShortCut	*ShortCutToText*
ShortCutToKey	*TextToShortCut*

The following routines are used to determine the parent form of components:

Table Intro.18 Parent form routines

GetParentForm	*ValidParentForm*

The following routines are used to create graphical points and rectangles:

Table Intro.19 Point and rectangle routines

Bounds	*Point*	*Rect*

The following routines are used to control Object Linking and Embedding (OLE) container applications:

Table Intro.20 OLE routines

BOLEMediumCalc	*LinksDlgEnabled*	*ReleaseOLEInitInfo*
ClearFormOLEDropFormats	*PasteSpecialDlg*	*SetFormOLEDropFormats*
InsertOLEObjectDlg	*PasteSpecialEnabled*	
LinksDlg	*RegisterFormAsOLEDropTarget*	

Library reference

The alphabetical reference following the sample entry in the next section contains a detailed description of the Delphi VCL objects, components, variables, properties, methods, events, routines, and types you use to develop Windows applications. The reference also contains the procedures, functions, types, variables, and constants that make up the Delphi run-time library and are declared in the *System* and *SysUtils* units. These procedures and functions are useful routines that exist outside of the objects of VCL. They are presented here so that you only need to search one reference source for the information you need about programming Delphi applications.

Each alphabetically listed entry contains the declaration format and a description of the entry. If the entry is an object, component, routine, or type, the unit that contains the entry is listed at the beginning of the entry. (The unit that corresponds to a variable, property, method, or event is the unit that contains the object or component to which the entry belongs.) If the entry applies to specific objects or components, they are listed. The cross-referenced entries and examples provide additional information about how to use the specified entry. The following sample illustrates this format.

Sample entry — Unit it occupies (if applicable)

Applies to
Listing of the objects and components the entry applies to, if any.

Declaration

```
{ The declaration of the entry from the unit it occupies }
```

A description containing specific information about the entry.

Note Any special notes that apply to the entry

Example
A description of the example code that follows.

```
   { Example code which illustrates the use of the entry }
```

See also
Related entries that are also listed in the *VCL Reference*.

Delphi Library Reference

Abort method

Applies to

TPrinter object

Declaration

```
procedure Abort;
```

The *Abort* procedure terminates the printing of a print job, dropping all unprinted data. The device is then set for the next print job. Use *Abort* to terminate the print job before it completes; otherwise, use the *EndDoc* method.

To use the *Abort* method, you must add the *Printers* unit to the **uses** clause of your unit.

Example

The following code aborts a print job if the user presses *Esc*. Note that you should set *KeyPreview* to *True* to ensure that the *OnKeyDown* event handler of *Form1* is called.

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if (Key=VK_ESCAPE) and Printer.Printing then
  begin
    Printer.Abort;
    MessageDlg('Printing aborted', mtInformation, [mbOK],0);
  end;
end;
```

See also

BeginDoc method, *EndDoc* method, *Printer* variable, *Printing* property

Abort procedure SysUtils

Declaration

```
procedure Abort;
```

The *Abort* procedure raises a special “silent exception” which operates like any other exception, but does not display an error message to the end user.

Use *Abort* to escape from an execution path without reporting an error.

Aborted property

Applies to

TPrinter object

Declaration

```
property Aborted: Boolean;
```

Run-time and read-only. The *Aborted* property determines if the user aborted the print job, thereby calling the *Abort* method. If *Aborted* is *True*, the print job was aborted. If it is *False*, the user did not abort the print job.

Example

The following code displays a dialog box if the print job was aborted:

```
if Printer.Aborted then
  MessageDlg('The print job did not finish printing'), mtInformation, [mbOK], 0);
```

See also

Abort method, *Printer* variable, *Printing* property

AbortOnKeyViol property

Applies to

TBatchMove component

Declaration

```
property AbortOnKeyViol: Boolean;
```

If *AbortOnKeyViol* is *True* (the default) and an integrity (key) violation occurs during the batch move operation, the *Execute* method will immediately terminate the operation. If you prefer to have the operation continue, with all key violations posted to the key violations table, set *AbortOnKeyViol* to *False*.

Note If you set *AbortOnKeyViol* to *False*, you should provide a *KeyViolTableName* to hold the records with errors.

Example

```
BatchMove1.AbortOnKeyViol := False;
```

See also

KeyViolCount property, *KeyViolTableName* property

AbortOnProblem property

Applies to
TBatchMove component

Declaration

```
property AbortOnProblem: Boolean;
```

If *AbortOnProblem* is *True* (the default) and it would be necessary to discard data from a source record to place it into the *Destination*, the *Execute* method will immediately terminate the batch move operation. If you prefer to have the operation continue, with all problems posted to the problems table, set *AbortOnProblem* to *False*.

Note If you set *AbortOnProblem* to *False*, you should provide a *ProblemTableName* to hold the records with problems.

Example

```
BatchMove1.AbortOnProblem := False;
```

See also
ProblemCount property, *ProblemTableName* property

Abs function — System

Declaration

```
function Abs(X);
```

The *Abs* function returns the absolute value of the argument.

X is an integer-type or real-type expression.

Example

```
var
  r: Real;
  i: Integer;
begin
  r := Abs(-2.3);  { 2.3 }
  i := Abs(-157);  { 157 }
end;
```

Abstract procedure — System

Declaration

```
procedure Abstract;
```

A call to this procedure terminates the program with a run-time error.

When implementing an abstract object type, use calls to *Abstract* in virtual methods that must be overridden in descendant types. This ensures that any attempt to use instances of the abstract object type will fail.

Active property

Applies to

TOLEContainer, *TQuery*, *TStoredProc*, *TTable* components

For tables, queries, and stored procedures

Declaration

```
property Active: Boolean;
```

Set the *Active* property to *True* to open a dataset and put it in *Browse* state. Set it to *False* to close the dataset and put it in *Inactive* state. Changing the *Active* property is equivalent to calling the *Open* or *Close* method.

For *TQuery* and *TStoredProc*, if the SQL statement or stored procedure does not return a result set, then setting *Active* to *True* will raise an exception because Delphi expects to get a cursor.

Note *Post* is not called implicitly by setting *Active* to *False*. Use the *BeforeClose* event to post any pending edits explicitly.

Example

```
{ Close the dataset }
Table1.Active := False;
{ Open the dataset }
Table1.Active := True;
```

For OLE containers

Declaration

```
property Active: Boolean;
```

Run-time only. The *Active* property specifies whether the OLE object in an OLE container is active. Set *Active* to *True* to activate the OLE object. Set *Active* to *False* to deactivate the OLE object.

Note Setting *Active* to *False* only deactivates in-place active OLE objects. If the object is activated within its own window, you must deactivate the object by executing a File | Exit command (or its equivalent in the command structure) from the OLE server application.

Example

The following code activates *OLEContainer1* if it contains an OLE object.

```
OLEContainer1.Active := OLEContainer1.OLEObjAllocated;
```

See also

AutoActivate property, *InPlaceActive* property, *OnActivate* event

ActiveControl property

Applies to

TForm, *TScreen* components

Declaration

```
property ActiveControl: TWinControl;
```

For forms, the *ActiveControl* property indicates which control has focus, or has focus initially when the form becomes active. Your application can use the *ActiveControl* property to access methods of the active control. Only one control, the active control, can have focus at a given time in an application.

For the screen, *ActiveControl* is a read-only property. The value of *ActiveControl* is the control that currently has focus on the screen.

Note When focus shifts to another control, the *ActiveControl* property is updated before the *OnExit* event of the original control with focus occurs.

Example

The following event handler responds to timer events by moving the active control one pixel to the right:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  ActiveControl.Left := ActiveControl.Left + 1;
end;
```

See also

ActiveForm property, *OnActiveControlChange* event, *OnEnter* event, *OnExit* event

ActiveForm property

Applies to

TScreen component

Declaration

```
property ActiveForm: TForm;
```

Run-time and read only. The *ActiveForm* property indicates which form currently has focus, or will have focus when the application becomes active again after another Windows application has been active.

Example

This example changes the color of the current form.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Screen.ActiveForm := clBlue;
end;
```

See also

ActiveControl property, *ActiveMDIChild* property, *OnActivate* event, *OnActiveFormChange* event, *Screen* variable

ActiveMDIChild property

Applies to

TForm component

Declaration

```
property ActiveMDIChild: TForm;
```

Run-time and read only. The value of the *ActiveMDIChild* property is the form that currently has focus in an MDI application.

Example

This code uses a button on an MDI application. When the user clicks the button, the active MDI child form turns blue.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  BlueForm: TForm;
begin
  BlueForm := Form1.ActiveMDIChild;
  BlueForm.Color := clBlue;
end;
```

See also

ActiveForm property, *FormStyle* property, *MDIChildCount* property, *MDIChildren* property

ActivePage property

Applies to

TNotebook, *TTabbedNotebook* components

Declaration

```
property ActivePage: string;
```

The *ActivePage* property determines which page displays in the notebook or tabbed notebook control. The value of *ActivePage* must be one of the strings contained in the *Pages* property.

Example

This example uses a notebook control and a button on the form. The notebook has multiple pages, including one called Graphics options. When the user clicks the button, the Graphics options page displays in the notebook control.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Notebook1.ActivePage := 'Graphics options';
end;
```

See also

PageIndex property, *TTabSet* component

Add method

Applies to

TFieldDefs, *TIndexDefs*, *TList*, *TStringList*, *TStrings* objects; *TMenuItem*, *TOutline* components

For field definitions

Declaration

```
procedure Add(const Name: string; DataType: TFieldType; Size: Word; Required: Boolean);
```

The *Add* method creates a new *TFieldDef* object using the *Name*, *DataType*, and *Size* parameters, and adds it to *Items*. Except for special purposes, you do not need to use this method because the *Items* is filled for you when you open the dataset, or because *Update* fills *Items* without opening the dataset.

The value of the *Required* parameter determines whether the newly added field definition is a required field. If the *Required* parameter is *True*, the value of the *Required* property of the *TFieldDef* object is also *True*. If the *Required* parameter is *False*, the value of the *Required* property is also *False*.

For index definitions

Declaration

```
procedure Add(const Name, Fields: string; Options: TIndexOptions);
```

The *Add* method creates a new *TIndexDef* object using the *Name*, *Fields,* and *Options* parameters, and adds it to *Items*. Generally you will never need to use this method since the dataset will have already filled *Items* for you when it is open, or the *Update* method will fill *Items* without opening the dataset.

For list objects

Declaration

```
function Add(Item: Pointer): Integer;
```

The *Add* method adds a new item to the end of a list. *Add* returns the position of the item in the list stored in the *Items* property; the first item in the list has a value of 0. Specify the item you want added to the list as the value of the *Item* parameter.

Example

This example adds a new object to a list in a list object:

```
type
  TMyClass = class
    MyString: string;
    constructor Create(S: string);
  end;

constructor TMyClass.Create(S: string);
begin
  MyString := S;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  MyList: TList;
  MyObject, SameObject: TMyClass;
begin
  MyList := TList.Create;                            { create the list }
  try
    MyObject := TMyClass.Create('Semper Fidelis!');  { create a class instance }
    try
      MyList.Add(MyObject);                          { add instance to list }
      SameObject := TMyClass(MyList.Items[0]);  { get first element in list }
      MessageDlg(SameObject.MyString, mtInformation, [mbOk], 0); { show it }
    finally
      MyObject.Free;
    end;                              { don't forget to clean up! }
  finally
    MyList.Free;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  MyList: TList;
  MyObject, SameObject: TMyClass;
begin
  MyList := TList.Create;                                     { create the list }
  try
    MyObject := TMyClass.Create('Semper Fidelis!');    { create a class instance }
    try
      MyList.Add(MyObject);                                  { add instance to list }
      SameObject := TMyClass(MyList.Items[0]);           { get first element in list }
      MessageDlg(SameObject.MyString, mtInformation, [mbOk], 0);            { show it }
    finally
      MyObject.Free;                                     { don't forget to clean up! }
  finally
    MyList.Free;
end;
```

See also

Capacity property, *Clear* method, *Delete* method, *Expand* method, *First* method, *IndexOf* method, *Insert* method, *Last* method, *Remove* method

For string and string list objects

Declaration

```
function Add(const S: string): Integer;
```

The *Add* method adds a new string to a string list. The S parameter is the new string. *Add* returns the position of the item in the list; the first item in the list has a value of 0.

For *TStrings* objects, such as the *Items* property of a list box, the new string is appended to the end of the list unless the *Sorted* property of the list box or combo box is *True*. In such a case the string is inserted into the list of strings so as to maintain the sort order.

For *TStringList* objects, the value of the *Sorted* property determines how a string is added. If *Sorted* is *False*, the string is appended to the list. If *Sorted* is *True*, the new string is inserted into the list of strings so as to maintain the sort order.

Example

This code uses a button and a list box on a form. When the user clicks the button, the code adds a new string to a list box.

```
  procedure TForm1.Button1Click(Sender: TObject);
  begin
    ListBox1.Items.Add('New string');
  end;
```

This code uses a list box, a button, and a label on a form. When the user clicks the button, the code adds a new string to the list box and reports its position in the list box as the caption of the label.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Position: Integer;
begin
  Position:= ListBox1.Items.Add('New item');
  Label1.Caption := IntToStr(Position);
end;
```

See also

AddObject method, *AddStrings* method, *Clear* method, *Delete* method, *Duplicates* property, *Exchange* method, *Insert* method, *Items* property, *Lines* property, *Move* method, *Sorted* property

For menu items

Declaration

```
procedure Add(Item: TMenuItem);
```

The *Add* method adds a menu item to the end of a menu. Specify the menu item you want added as the value of the *Item* parameter.

Example

This code adds a menu item to a File menu:

```
procedure Form1.Button1Click(Sender: TSender);
var
  NewItem: TMenuItem;
begin
  NewItem := TMenuItem.Create(Self);
  NewItem.Caption := ‘New item’;
  File.Add(NewItem);
end;
```

See also

Delete method, *Insert* method

For outlines

Declaration

```
function Add(Index: LongInt; const Text: string): LongInt;
```

The *Add* method adds an outline item (*TOutlineNode* object) to an outline. The value of the *Index* parameter specifies where to add the new item. The *Text* parameter specifies the *Text* property value of the new item. *Add* returns the *Index* property value of the added item.

The added item is positioned in the outline as the last sibling of the outline item specified by the *Index* parameter. The new item shares the same parent as the item specified by the *Index* parameter. Outline items that appear after the added item are moved down one row and reindexed with valid *Index* values. This is done automatically unless *BeginUpdate* was called.

Note To add items to an empty outline, specify zero (0) as the *Index* parameter.

Example

The following code adds a new item at the top level of the outline. The new item is identified by the text 'New item':

```
Outline1.Add(0, 'New item');
```

See also

AddChild method, *AddChildObject* method, *AddObject* method, *Insert* method, *MoveTo* method

AddChild method

Applies to

TOutline component

Declaration

```
function AddChild(Index: LongInt; const Text: string): LongInt;
```

The *AddChild* method adds an outline item (*TOutlineNode* object) to an outline as a child of an existing item. The value of the *Index* parameter specifies where to add the new item. The *Text* parameter specifies the *Text* property value of the new item. *AddChild* returns the *Index* property value of the added item.

The added item is positioned in the outline as the last child of the outline item specified by the *Index* parameter. Outline items that appear after the added item are moved down one row and reindexed with valid *Index* values. This is done automatically unless *BeginUpdate* was called.

Note To add items to an empty outline, specify zero (0) as the *Index* parameter.

Example

The following code adds a new child to the selected item of the outline. The new item is identified by the text 'New child':

```
Outline1.AddChild(Outline1.SelectedItem, 'New child');
```

See also

Add method, *AddChildObject* method, *AddObject* method, *Insert* method, *MoveTo* method

AddChildObject method

Applies to

TOutline component

Declaration

```
function AddChildObject(Index: LongInt; const Text: string; const Data: Pointer): LongInt;
```

The *AddChildObject* method adds an outline item (*TOutlineNode* object) containing data to an outline as a child of an existing item. The value of the *Index* parameter specifies where to add the new item. The *Text* parameter specifies the *Text* property value of the new item. The *Data* parameter specifies the *Data* property value of the new item. *AddChild* returns the *Index* property value of the added item.

The added item is positioned in the outline as the last child of the outline item specified by the *Index* parameter. Outline items that appear after the added item are moved down one row and reindexed with valid *Index* values. This is done automatically unless *BeginUpdate* was called.

Note To add items to an empty outline, specify zero (0) as the *Index* parameter.

Example

The following code adds a new child to the selected item of the outline. The new item is identified by the text 'New child'. The *TBitmap* object named *Bitmap1* is attached to the new item:

```
Outline1.AddChildObject(Outline1.SelectedItem, 'New child', Bitmap1);
```

See also

Add method, *AddChild* method, *AddObject* method, *Insert* method, *MoveTo* method

AddExitProc procedure **SysUtils**

Declaration

```
procedure AddExitProc(Proc: TProcedure);
```

AddExitProc adds the given procedure to the run-time library's exit procedure list. When an application terminates, its exit procedures are executed in reverse order of definition, i.e. the last procedure passed to *AddExitProc* is the first one to get executed upon termination.

AddFieldDesc method

Applies to

TFieldDefs object

Declaration

```
procedure AddFieldDesc(FieldDesc: FLDDesc; FieldNo: Word);
```

AddFieldDesc creates a new *TFieldDef* object using the information provided by the Borland Database Engine in the *FieldDesc* parameter, and adds it to *Items*. Except for special purposes, you do not need to use this method because the *Items* is filled for you when you open the dataset, or because *Update* fills *Items* without opening the dataset.

AddIndex method

Applies to
TTable component

Declaration

```
procedure AddIndex(const Name, Fields: string; Options: TIndexOptions);
```

The *AddIndex* method creates a new index for the *TTable*. *Name* is the name of the new index. *Fields* is a list of the fields to include in the index. Separate the field names by a semicolon. *Options* is a set of values from the *TIndexOptions* type.

Example

```
  Table1.AddIndex('NewIndex', 'CustNo;CustName', [ixUnique, ixCaseInsensitive]);
```

See also
DeleteIndex method, *IndexDefs* property, *IndexName* property

AddObject method

Applies to
TStringList, *TStrings* objects; *TOutline* component

For string and string list objects

Declaration

```
function AddObject(const S: string; AObject: TObject): Integer;
```

The *AddObject* method adds both a string and an object to a string or string list object. The string and the object are appended to the list of strings. Specify the string to be added as the value of the S parameter, and specify the object to be added as the value of the *AObject* parameter.

Example
This code adds the string 'Orange' and a bitmap of an orange to an owner-draw list box:

```
procedure TForm1.Button1Click(Sender: TSender);
var
  Icon: TIcon;
begin
  Icon := TIcon.Create;
  Icon.LoadFromFile('ORANGE.ICO');
  ListBox1.Items.AddObject('Orange', Icon);
end;
```

See also

Add method, *AddStrings* method, *IndexOf* method, *IndexOfObject* method, *InsertObject* method, *Objects* property, *Strings* property

For outlines

Declaration

```
function AddObject(Index: LongInt; const Text: string; const Data: Pointer): LongInt;
```

The *AddObject* method adds an outline item (*TOutlineNode* object) containing data to an outline. The value of the *Index* parameter specifies where to add the new item. The *Text* parameter specifies the *Text* property value of the new item. The *Data* parameter specifies the *Data* property value of the new item. *Add* returns the *Index* property value of the added item.

The added item is positioned in the outline as the last sibling of the outline item specified by the *Index* parameter. The new item shares the same parent as the item specified by the *Index* parameter. Outline items that appear after the added item are moved down one row and reindexed with valid *Index* values. This is done automatically unless *BeginUpdate* was called.

Note To add items to an empty outline, specify zero (0) as the *Index* parameter.

Example

The following code defines a record type of *TMyRec* and a record pointer type of *PMyRec*.

```
type
  PMyRec = ^TMyRec;
  TMyRec = record
    FName: string;
    LName: string;
  end;
```

Assuming these types are used, the following code adds an outline node to *Outline1*. A *TMyRec* record is associated with the added item. The *FName* and *LName* fields are obtained from edit boxes *Edit1* and *Edit2*. The *Index* parameter is obtained from edit box *Edit3*. The item is added only if the *Index* is a valid value.

```
var
  MyRecPtr: PMyRec;
  OutlineIndex: LongInt;
begin
  New(MyRecPtr);
  MyRecPtr^.FName := Edit1.Text;
  MyRecPtr^.LName := Edit2.Text;
  OutlineIndex := StrToInt(Edit3.Text);
  if (OutlineIndex <= Outline1.ItemCount) and (OutlineIndex >= 0) then
    Outline1.AddObject(OutlineIndex, 'New item', MyRecPtr);
end;
```

After an item containing a *TMyRec* record has been added, the following code retrieves the *FName* and *LName* values associated with the item and displays the values in labels.

```
  Label4.Caption := PMyRec(Outline1.Items[Outline1.SelectedItem].Data)^.FName;
  Label5.Caption := PMyRec(Outline1.Items[Outline1.SelectedItem].Data)^.LName;
```

See also
Add method, *AddChild* method, *AddChildObject* method, *Insert* method, *MoveTo* method

AddParam method

Applies to
TParams object

Declaration

```
procedure AddParam(Value: TParam);
```

AddParam adds *Value* as a new parameter to the *Items* property.

Example

```
  { Move all parameter info from Params2 to Params1 }
  while Params2.Count <> 0 do
    begin
  { Grab the first parameter from Params2 }
    TempParam := Params2[0];
  { Remove it from Params2 }
    Params2.RemoveParam(TempParam);
  { And add it to Params1 }
    Params1.AddParam(TempParam);
    end;
```

See also
RemoveParam method

AddPassword method

Applies to
TSession component

Declaration

```
procedure AddPassword(const Password: string);
```

The *AddPassword* method is used to add a new password to the current *TSession* component for use with Paradox tables. When an application opens a Paradox table that requires a password, the user will be prompted to enter a password unless the Session has a valid password for the table.

Example

```
Session.AddPassword('ASecret');
```

See also

Session variable

Addr function **System**

Declaration

```
function Addr(X): pointer;
```

The *Addr* function returns the address of a specified object.

X is any variable, procedure or function identifier. The result is a pointer to *X*.

The result of *Addr* is of the predefined type *Pointer*, which means that it is assignment-compatible with all pointer types but can't be dereferenced directly without a typecast.

Example

```
var
  P: Pointer;
begin
  P := Addr(P);    { Now points to itself }
end;
```

See also

Ofs function, *Ptr* function, *Seg* function

AddStrings method

Applies to

TStringList, *TStrings* objects

Declaration

```
procedure AddStrings(Strings: TStrings);
```

The *AddStrings* method adds a group of strings to the list of strings in a string or string list object. The new strings are appended to the existing strings. Specify a string object containing the list of strings you want added as the value of the *Strings* parameter.

Example

This code appends the contents of a file to the end of a memo control:

```
procedure TForm1.Button1Click(Sender: TSender);
var
  Contents: TStringList;
begin
  Contents.LoadFromFile('NEWSTUFF.TXT');
  Memo1.Lines.AddStrings(Contents);
finally
  Contents.Free;
end;
```

This code adds the list of strings contained in *ListBox1.Items* to the end of the *ListBox2.Items* list of strings:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox2.Items.AddStrings(ListBox1.Items);
end;
```

See also

Add method, *AddObject* method, *Strings* property

AfterCancel event

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
property AfterCancel: TDataSetNotifyEvent;
```

The *AfterCancel* event is activated when the *dataset* finishes a call to the *Cancel* method. This event is the last action before *Cancel* returns to the caller. If the dataset is not in Edit state or there are no changes pending, then *Cancel* will not activate the *AfterCancel* event.

By assigning a method to this property, you can take any special actions required by the event.

See also

BeforeCancel event

AfterClose event

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
property AfterClose: TDataSetNotifyEvent;
```

The *AfterClose* event is activated after a *dataset* is closed, either by calling the *Close* method or by setting the *Active* property to *False*. This event is the last action before *Close* returns to the caller. Typically, the *AfterClose* event handler closes any private lookup tables opened by the *BeforeOpen* event.

By assigning a method to this property, you can take any special actions required by the event.

See also

BeforeClose event

AfterDelete event

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
property AfterDelete: TDataSetNotifyEvent;
```

The *AfterDelete* event is activated when the *dataset* finishes a call to the *Delete* method. This event is the last action before *Delete* returns to the caller. When *AfterDelete* is called, the deleted record has already been removed from the dataset, and the dataset cursor will be positioned on the following record.

By assigning a method to this property, you can take any special actions required by the event.

See also

BeforeDelete event

AfterEdit event

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
property AfterEdit: TDataSetNotifyEvent;
```

The *AfterEdit* event is activated when a *dataset* finishes a call to the *Edit* method. This event is the last action before *Edit* returns to the caller.

Note The event occurs before any changes have been made to the current record.

By assigning a method to this property, you can take any special actions required by the event.

See also
BeforeEdit event

AfterInsert event

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
property AfterInsert: TDataSetNotifyEvent;
```

The *AfterInsert* event is activated when a *dataset* finishes a call to the *Insert* or *Append* methods. This event is the last action before *Insert* or *Append* returns to the caller.

Note This event occurs before a new record has been added to the component.

By assigning a method to this property, you can take any special actions required by the event.

See also
BeforeInsert event

AfterOpen event

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
property AfterOpen: TDataSetNotifyEvent;
```

The *AfterOpen* event is activated after a *dataset* is opened, either by calling the *Open* method or by setting the *Active* property to *True*. This event is the last action before *Open* returns to the caller.

By assigning a method to this property, you can take any special actions required by the event.

See also
BeforeOpen event

AfterPost event

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
property AfterPost: TDataSetNotifyEvent;
```

The *AfterPost* event is activated after a call to the *Post* method. This event is the last action before *Post* returns to the caller.

If a *TTable* has a range filter (set with *ApplyRange*) in effect, and if the key value of the newly posted record falls outside the range, then in the *AfterPost* event, the cursor will not be positioned on the newly posted record.

By assigning a method to this property, you can take any special actions required by the event.

See also
BeforePost event

AliasName property

Applies to
TDataBase component

Declaration

```
property AliasName: TSymbolStr;
```

AliasName is the name of an existing BDE alias defined with the BDE Configuration Utility. This is where the *TDatabase* component gets its default parameter settings. This property will be cleared if *DriverName* is set. If you try to set *AliasName* of a *TDatabase* for which *Connected* is True, Delphi will raise an exception.

Example

```
   Database1.AliasName := 'DBDEMOS';
```

Align property

Applies to

At design time: *TBevel*, *TDBGrid*, *TDBRadioGroup*, *TDirectoryListBox*, *TDrawGrid*, *TFileListBox*, *THeader*, *TImage*, *TLabel*, *TListBox*, *TMaskEdit*, *TMemo*, *TNotebook*, *TOLEContainer*, *TOutline*, *TPaintBox*, *TPanel*, *TRadioGroup*, *TScrollBox*, *TStringGrid*, *TTabbedNotebook*, *TTabSet* components

At run time: All controls

Declaration

```
property Align: TAlign;
```

The *Align* property determines how the controls align within their container (or parent control). These are the possible values:

Value	Meaning
alNone	The component remains where you place it in the form. This is the default value.
alTop	The component moves to the top of the form and resizes to fill the width of the form. The height of the component is not affected.
alBottom	The component moves to the bottom of the form and resizes to fill the width of the form. The height of the component is not affected.
alLeft	The component moves to the left side of the form and resizes to fill the height of the form. The width of the component is not affected.
alRight	The component moves to the right side of the form and resizes to fill the height of the form. The width of the component is not affected.
alClient	The component resizes to fill the client area of a form. If a component already occupies part of the client area, the component resizes to fit within the remaining client area.

If the form or a component containing other components is resized, the components realign within the form or control.

Using the *Align* property is useful when you want a control to stay in one position on the form, even if the size of the form changes. For example, you could use a panel component with a various controls on it as a tool palette. By changing*Align* to *alLeft*, you guarantee that the tool palette always remains on the left side of the form and always equals the client height of the form.

Example

This example moves a panel control named *Panel1* to the bottom of the form and resizes it to fill the width of the form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Panel1.Align := alBottom;
end;
```

See also

Alignment property

Alignment property

Applies to

TBCDField, *TBooleanField*, *TCheckBox*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TDBCheckBox*, *TDBMemo*, *TDBText*, *TFloatField*, *TIntegerField*, *TLabel*, *TMemo*, *TPanel*, *TPopupMenu*, *TRadioButton*, *TSmallintField*, *TStringField*, *TTimeField*, *TWordField* components

For labels, memos, and panels

Declaration

```
property Alignment: TAlignment;
```

The *Alignment* property specifies how text is aligned within the component.

These are the possible values:

Value	Meaning
taLeftJustify	Align text to the left side of the control
taCenter	Center text horizontally in the control
taRightJustify	Align text to the right side of the control

Example

This code aligns text to the right side of a label named *Label1* in response to a click on a button named *RightAlign*:

```
procedure TForm1.RightAlignClick(Sender: TObject);
begin
  Label1.Alignment := taRightJustify;
end;
```

See also

Caption property, *Text* property

For check boxes and radio buttons

Declaration

```
property Alignment: TLeftRight;
```

For check boxes and radio buttons, the control's caption is always left-aligned within the text area. If the check box is two-dimensional (its *Ctl3D* property is *False*), *Alignment* determines the placement of that caption area relative to the control's check box or radio button. If the check box is three dimensional (its *Ctl3D* property is *True*), the value of the *Alignment* property has no effect on the check box.

These are the possible values:

Value	Meaning
taLeftJustify	The caption appears to the left of the check box or radio button.
taRightJustify	The caption appears to the right of the check box or radio button.

Example

This code makes the check box two-dimensional and puts the check box on the left side of the text:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  CheckBox1.Ctl3D := False;
  CheckBox1.Alignment := taLeftJustify;
end;
```

See also

Caption property

For pop-up menus

Declaration

```
property Alignment: TPopupAlignment;
```

The *Alignment* property determines where the pop-up menu appears when the user clicks the right mouse button. These are the possible values and their meanings:

Value	Meaning
paLeft	The pop-up menu appears with its top left corner under the mouse pointer.
paCenter	The pop-up menu appears with the top center of the menu under the mouse pointer.
paRight	The pop-up menu appears with its top right corner under the mouse pointer.

The default value is *paLeft*.

Example

This example uses a pop-up menu component and a button on a form. The code places the top right corner of a pop-up menu under the mouse pointer when the menu appears:

```
procedure TForm1.AlignPopupMenuClick(Sender: TObject);
begin
  PopupMenu1.Alignment := paRight;
end;
```

See also

AutoPopup property, *OnPopup* event

For field components

Declaration

```
property Alignment: TAlignment;
```

The *Alignment* property is used by some data-aware controls to center, left-, or right-align the data in a field. Data-aware controls that support alignment include *TDBGrid* and *TDBEdit*.

AllocMem function

SysUtils

Declaration

```
function AllocMem(Size: Cardinal): Pointer;
```

AllocMem allocates a block of the given size on the heap. Each byte in the allocated buffer is set to zero. To dispose the buffer, use the *FreeMem* standard procedure.

See also

ReAllocMem function

AllowAllUp property

Applies to

TSpeedButton component

Declaration

```
property AllowAllUp: Boolean;
```

The *AllowAllUp* property determines if all speed buttons in a group this speed button belongs to can be unselected (in their up state) at the same time. *AllowAllUp* should be used only with speed buttons in a group (that is, the value of the button's *GroupIndex* property is not zero). See the *GroupIndex* property for information on how to create a group of speed buttons. If *GroupIndex* is zero, *AllowAllUp* has no effect.

If *AllowAllUp* is *True*, all of the speed buttons in a group can be unselected. All buttons can appear in their up state.

If *AllowAllUp* is *False*, one of the speed buttons belonging to a group must be selected (in its down state) at all times. Clicking a down button won't return the button to its up state. The button only becomes unselected when the user clicks one of the other buttons in the group. In such a group, one button must always be selected. Determine which speed button will be initially down by setting its *Down* property to *True*.

The default value is *False*.

Changing the value of the *AllowAllUp* property for one speed button in a group changes the *AllowAllUp* value for all buttons in the group.

You can use *AllowAllUp* with a single bitmap button in its own group (with a GroupIndex value greater than 0) so that the button can be selected and remain selected until the user clicks the button again—at which time it becomes unselected. In other words, the button can work much like a check box. To make a single speed button behave this way, set its *GroupIndex* property to a value greater than 0 (but different from any other *GroupIndex* value of any other speed buttons you have), and set *AllowAllUp* to *True*.

Example

In this example, there are three speed buttons on a form. All three belong to the same group as all three have a *GroupIndex* value of 1. This line of code changes the *AllowAllUp* property to *True* for all three speed buttons, so it's possible that all the speed buttons in the group can be unselected at the same time:

```
SpeedButton3.AllowAllUp := True;
```

See also

Down property, *Glyph* property, *GroupIndex* property

AllowGrayed property

Applies to

TCheckBox, *TDBCheckBox* components

Declaration

```
property AllowGrayed: Boolean;
```

The value of the *AllowGrayed* property determines if a check box can have two or three possible states. If *AllowGrayed* is *False*, the default value, clicking a check box alternately checks and unchecks it. If *AllowGrayed* is *True*, clicking a check box either checks, grays, or unchecks it.

Example

This example uses a check box on a form. When the application runs, the check box is initially checked. When the user clicks it, the check box is unchecked. Clicking it again grays the check box.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  CheckBox1.AllowGrayed := True;
  CheckBox1.State := cbChecked;
end;
```

See also

Checked property, *State* property

AllowInPlace property

Applies to

TOLEContainer component

Declaration

```
property AllowInPlace: Boolean;
```

The *AllowInPlace* property specifies whether an OLE object can be activated in place. If *AllowInPlace* is *True*, in-place activation is allowed. If *AllowInPlace* is *False*, in-place activation is not allowed and the OLE object is activated in its own window (OLE 1.0-style).

Note To support in-place activation, the OLE container application must include a *TMainMenu* component.

Example

The following code sets *AllowInPlace* to *False*.

```
OLEContainer1.AllowInPlace := False;
```

See also

AutoActivate property

AllowResize property

Applies to

THeader component

Declaration

```
property AllowResize: Boolean;
```

The value of the *AllowResize* property determines if the user can modify the size of the header at run time with the mouse. If *AllowResize* is *False*, the sections within a header can't be resized. If *AllowResize* is *True*, clicking a border of a header section and dragging it left or right changes the width of the section. The default value is *True*.

Example

The following code allows the resizing of the sections of *Header1*.

```
Header1.AllowResize := True;
```

See also

OnSized event, *Sections* property, *SectionWidth* property, *Sizing* event

AnsiCompareStr function — SysUtils

Declaration

```
function AnsiCompareStr(const S1, S2: string): Integer;
```

AnsiCompareStr compares *S1* to *S2*, with case sensitivity. The compare operation is controlled by the currently installed language driver. The return value is the same as for *CompareStr*.

See also

AnsiCompareText function

AnsiCompareText function — SysUtils

Declaration

```
function AnsiCompareText(const S1, S2: string): Integer;
```

AnsiCompareText compares *S1* to *S2*, without case sensitivity. The compare operation is controlled by the currently installed language driver. The return value is the same as for *CompareStr*.

See also

AnsiCompareStr function

AnsiLowerCase function — SysUtils

Declaration

```
function AnsiLowerCase(const S: string): string;
```

AnsiLowerCase converts all characters in the given string to lower case. The conversion uses the currently installed language driver.

See also

AnsiUpperCase function, *LowerCase* function

AnsiToNative function — DB

Declaration

```
function AnsiToNative(Locale: TLocale; const AnsiStr: string; NativeStr: PChar;
  MaxLen: Word): PChar;
```

The *AnsiToNative* function translates the ANSI characters in *AnsiStr* (or the first *MaxLen* characters) to the native character set according to *Locale* by calling *DBIAnsiToNative*.

The translated characters are returned in *NativeStr* with a null terminator. *AnsiToNative* returns *NativeStr.*

AnsiUpperCase function **SysUtils**

Declaration

```
function AnsiUpperCase(const S: string): string;
```

AnsiUpperCase converts all characters in the given string to upper case. The conversion uses the currently installed language driver.

See also

AnsiLowerCase function, *UpperCase* function

Append method

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure Append;
```

The *Append* method moves the cursor to the end of the dataset, puts the *dataset* into Insert state , and opens a new, empty record. When an application calls *Post*, the new record will be inserted in the dataset in a position based on its index, if defined. To discard the new record, use *Cancel*.

This method is valid only for datasets that return a live result set.

Note For indexed tables, the *Append* and *Insert* methods will both put the new record in the correct location in the table, based on the table's index. If no index is defined on the underlying table, then the record will maintain its position—*Append* will add the record to the end of the table, and *Insert* will insert it at the current cursor position. In either case, posting a new record may cause rows displayed in a data grid to change as the dataset follows the new row to its indexed position and then fetches data to fill the data grid around it.

Example

```
with Table1 do
  begin
  Append;
  FieldByName('CustNo').AsString := '9999';
  { Fill in other fields here }
  if { you are sure you want to do this} then Post
  else { if you changed your mind } Cancel;
  end.
```

See also
TField component

Append procedure System

Declaration

```
procedure Append(var f: Text);
```

The *Append* procedure opens an existing file with the name assigned to *F*, so that new text can be added.

F is a text file variable and must be associated with an external file using *AssignFile*.

If no external file of the given name exists, an error occurs.

If *F* is already open, it is closed, then reopened. The current file position is set to the end of the file.

If a *Ctrl+Z* (ASCII 26) is present in the last 128-byte block of the file, the current file position is set so that the next character added to the file overwrites the first *Ctrl+Z* in the block. In this way, text can be appended to a file that terminates with a *Ctrl+Z*.

If *F* was not assigned a name, then, after the call to *Append*, *F* refers to the standard output file (standard handle number 1).

After calling *Append*, *F* is write-only, and the file pointer is at the end of the file.

{**$I+**} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using {**$I–**}, you must use *IOResult* to check for I/O errors.

Example

```
var F: TextFile;
begin
  if OpenDialog1.Execute then  { Bring up open file dialog }
  begin
    AssignFile(F, OpenDialog1.FileName);
          { Open file selected in dialog }
    Append(F);                                            { Add more text onto end }
    Writeln(F, 'appended text');
    CloseFile(F);                                       { Close file, save changes }
  end;
end;
```

See also
AssignFile procedure, *FileClose* procedure, *Reset* procedure, *Rewrite* procedure

AppendRecord method

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure AppendRecord(const Values: array of const);
```

The *AppendRecord* method appends a new record to the *dataset* using the field values passed in the *Values* parameter. The assignment of the elements of *Values* to fields in the record is sequential; the first element is assigned to the first field, the second to the second, etc. The number of field values passed in *Values* may be fewer than the number of actual fields in the record; any remaining fields are left unassigned and are NULL. The type of each element of *Values* must be compatible with the type of the field in that the field must be able to perform the assignment using *AsString*, *AsInteger*, and so on, according the type of the *Values* element.

This method is valid only for datasets that return a live result set.

Note For indexed tables, the *AppendRecord* and *InsertRecord* methods will both put the new record in the correct location in the table, based on the table's index. If no index is defined on the underlying table, then the record will maintain its position—*AppendRecord* will add the record to the end of the table, and *InsertRecord* will insert it at the current cursor position. In either case, posting a new record in a data grid may cause all the rows before and after the new record to change as the dataset follows the new row to its indexed position and then fetches data to fill the grid around it.

Example

```
  Table1.AppendRecord([9999, 'Doe', 'John']);
```

See also

TField component.

AppendStr procedure **SysUtils**

Declaration

```
procedure AppendStr(var Dest: string; const S: string);
```

AppendStr appends *S* to the end of *Dest*. *AppendStr* corresponds to the statement "Dest := Dest + S", but is more efficient.

Application variable

Forms

Declaration

```
Application: TApplication;
```

The *Application* variable declares an instance of your application for your project. By default, when you create a new project, Delphi constructs an application object and assigns it to *Application*. *Application* has several properties you can use to get information about your application while it runs; see the *TApplication component* for the list of properties.

Example

This code displays the name of your project in an edit box:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.Text := Application.Title;
end;
```

See also

Icon property, *Run* method, *Title* property

ApplyFilePath method

Applies to

TFileListBox component

Declaration

```
procedure ApplyFilePath(const EditText: string);
```

ApplyFileEditText is intended to be used in a dialog box that approximates the utility and behavior of an Open dialog box. Such a dialog box would contain a file list box (*TFileListBox*), a directory list box (*TDirectoryListBox*), a drive combo box (*TDriveComboBox*), a filter combo box *TFilterComboBox*, a label, and an edit box where the user can type a file name including a full directory path. When the user then chooses the OK button, you would like all the controls to update with the information the user entered in the edit box. For example, you would want the directory list box to change to the directory specified in the path the user typed, and you want the drive combo box to change to the correct drive if the path included a different drive letter.

If the file list box, directory list box, drive combo box, filter combo box, label, and edit box are connected using the *FileEdit*, *FileList*, *DirLabel*, and *DirList* properties, your application can call *ApplyFilePath* to update the controls with the text the user entered in the edit box.

The user can enter any of these strings in the edit box: a file name, with or without a path, a drive only, a drive and directory only, relative paths, or a file mask using wildcard characters. In all cases, the *ApplyFilePath* method updates the controls as you would expect. For example, if the user includes a directory name, the directory list box makes that directory the current one.

The *EditText* parameter is the text within the edit box.

Example

This example uses a file list box, a directory list box, a filter combo box, a drive combo box, a label, an edit box, and a button on a form. When the user runs the application and enters a path or file name in the edit box, all the controls update:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FileListBox1.FileEdit := Edit1;
  FilterComboBox1.FileList := FileListBox1;
  DirectoryListBox1.FileList := FileListBox1;
  DirectoryListBox1.DirLabel := Label1;
  DriveComboBox1.DirList := DirectoryListBox1;
  Button1.Default := True;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  FileListBox1.ApplyFilePath(Edit1.Text);
end;
```

See also

Directory property, *Drive* property

ApplyRange method

Applies to

TTable component

Declaration

```
procedure ApplyRange;
```

The *ApplyRange* method is used to apply the start and end ranges established with the *SetRangeStart* and *SetRangeEnd* methods or the *EditRangeStart* and *EditRangeEnd* methods. This will filter the set of records from the database table accessible to the application.

Note When comparing fields for range purposes, a NULL field is always less than any other possible value.

Example

```
{ Limit the range from ‘Goleta’ to ‘Santa Barbara’}
with Table1 do
begin
  EditRangeStart; { Set the beginning key }
```

```
    FieldByName('City').AsString := 'Goleta';
    EditRangeEnd; { Set the ending key }
    FieldByName('City').AsString := 'Santa Barbara';
    ApplyRange; { Tell the dataset to establish the range }
  end;
```

See also

CancelRange method, *KeyExclusive* property, *KeyFieldCount* property, *SetRange* method

Arc method

Applies to

TCanvas object

Declaration

```
procedure Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);
```

The *Arc* method draws an arc on the canvas along the perimeter of the ellipse bounded by the specified rectangle. Coordinates (X1, Y1 and X2, Y2) define the enclosing rectangle for the arc. The arc starts at the intersection of the ellipse edge and the line from the center of the ellipse to the specified starting point (X3, Y3). The arc is drawn counterclockwise until it reaches the position where the ellipse edge intersects the line from the center of the ellipse to the specified ending point (X4, Y4).

Example

The following lines of code draw the top quarter of an arc bounded by the current window:

```
  TForm1.FormPaint(Sender: TObject);
  var
    R: TRect;
  begin
    R := GetClientRect;   {Gets the rectangular coordinates of the current window}
    Canvas.Arc(R.Left, R.Top, R.Right, R.Bottom, R.Right, R.Top, R.Left, R.Top);
  end;
```

See also

Chord method, *Draw* method, *DrawFocusRect* method, *Ellipse* method, *Pie* method

ArcTan function

System

Declaration

```
function ArcTan(X: Real): Real;
```

The *ArcTan* function returns the resulting arctangent of the argument.

You can calculate other trigonometric functions using *Sin*, *Cos*, and *ArcTan* in the following expressions:

```
Tan(x) = Sin(x) / Cos(x)
ArcSin(x) = ArcTan (x/sqrt (1-sqr (x)))
ArcCos(x) = ArcTan (sqrt (1-sqr (x)) /x)
```

Example

```
var
  R: Real;
begin
  R := ArcTan(Pi);
end;
```

See also

Cos function, *Sin* function

ArrangeIcons method

Applies to

TForm component

Declaration

```
procedure ArrangeIcons;
```

The *ArrangeIcons* method arranges the icons of minimized forms so that they are evenly spaced and don’t overlap. The *ArrangeIcons* method applies only to forms that are MDI parent forms (have a *FormStyle* property value of *fsMDIForm*).

Example

This code runs when the user chooses a menu item called Window | Arrange Icons:

```
procedure TForm1.WindowArrangeIconsClick(Sender: TObject);
begin
  Form1.ArrangeIcons;
end;
```

See also

Cascade method, *Next* method, *Previous* method, *Tile* method

AsBCD property

Applies to

TParam object

Declaration

```
property AsBCD: Double;
```

Assigning a value to the *AsBCD* property sets the *DataType* property to *ftBCD* and saves the value as the current data for the parameter.

See also

TFieldType type

AsBoolean property

Applies to

TParam object; *TBooleanField*, *TStringField* components

For TParam objects

Declaration

```
property AsBoolean: Boolean;
```

Assigning a value to the *AsBoolean* property sets the *DataType* property to *ftBoolean* and saves the value as the current data for the parameter. Accessing the *AsBoolean* property attempts to convert the current data to a *Boolean* value and returns that value.

For Boolean and string field components

Declaration

```
property AsBoolean: Boolean;
```

Run-time only. This is a conversion property. For a *TBooleanField*, *AsBoolean* can be used to read or set the value of the field, but *Value* should be used for this purpose instead.

For a *TStringField*, *AsBoolean* returns *True* on reading the value of the field if its text begins with the letters "Y", "y", "T" or "t" (for "Yes" or "True"), and *False* otherwise. Using *AsBoolean* to write a *TStringField*'s value sets the string to 'T' or 'F'.

Example

```
  if Table1.FieldByName('BackOrdered').AsBoolean then ...
```

AsCurrency property

Applies to

TParam object

Declaration

```
property AsCurrency: Double;
```

Assigning a value to the *AsCurrency* property sets the *DataType* property to *ftCurrency* and saves the value as the current data for the parameter. Accessing the *AsCurrency* property attempts to convert the current data to a *Double* value and returns that value.

See also

TFieldType type

AsDate property

Applies to

TParam object

Declaration

```
property AsDate: TDateTime;
```

Assigning a value to the *AsDate* property sets the *DataType* property to *ftDate* and saves the value as the current data for the parameter. Accessing the *AsDate* property attempts to convert the current data to a *TDateTime* value and returns that value.

See also

StrToDateTime function, *TFieldType* type

AsDateTime property

Applies to

TParam object; *TDateField*, *TDateTimeField*, *TStringField*, *TTimeField* components

For TParam objects

Declaration

```
property AsDateTime: TDateTime;
```

Assigning a value to the *AsDateTime* property sets the *DataType* property to *ftDateTime* and saves the value as the current data for the parameter. Accessing the *AsDateTime* property attempts to convert the current data to a *TDateTime* value and returns that value.

See also

StrToDateTime function, *TFieldType* type

For date, date-time, time, and string field components

Declaration

```
property AsDateTime: TDateTime;
```

Run-time only. This is a conversion property. For *TDateField*, *TDateTimeField* or *TTimeField*, *AsDateTime* can be used to read or set the value of the field, but *Value* should be used for this purpose instead.

For a *TStringField*, *AsDateTime* converts a date to a string on assigning a value to the string field, and converts a string to a date when reading from the field.

Example

The following statement converts a string to a date for insertion into a date field:

```
   Table1.FieldByName(TimeStamp).AsDateTime := StrToDateTime(Now);
```

See also

DateToStr function, *StrToDate* function, *StrToDateTime* function, *DateTimeToStr* function, *TimeToStr* function, *StrToTime* function, *Value* property

AsFloat property

Applies to

TParam object; *TBCDField, TCurrencyField*, *TFloatField*, *TStringField* components

For TParam objects

Declaration

```
property AsFloat: Double;
```

Assigning a value to the *AsFloat* property sets the *DataType* property to *ftFloat* and saves the value as the current data for the parameter. Accessing the *AsFloat* property attempts to convert the current data to a Double value and returns that value.

See also

TFieldType type

For field components

Declaration

```
property AsFloat: Double;
```

Run-time only. This is a conversion property. For a *TFloatField*, *TBCDField* or *TCurrencyField*, *AsFloat* can be used to read or set the value of the field as a *Double*, but *Value* should be used for this purpose instead.

For a *TStringField*, *AsFloat* converts a float to a string on assigning a value to the field, and converts a string to a float when reading from the field.

See also

FloatToStr function, *StrToFloat* function

AsInteger property

Applies to

TParam object; *TIntegerField*, *TSmallintField*, *TStringField*, *TWordField* components

For TParam objects

Declaration

```
property AsInteger: LongInt;
```

Assigning a value to the *AsInteger* property sets the *DataType* property to *ftInteger* and saves the value as the current data for the parameter. Accessing the *AsInteger* property attempts to convert the current data to a *Longint* value and returns that value.

See also

TFieldType type

For field components

Declaration

```
property AsInteger: Longint;
```

Run-time only. This is a conversion property. For a *TIntegerField*, *TSmallintField* or *TWordField*, *AsInteger* can be used to read or set the value of the field as a *Longint*, but *Value* should be used for this purpose instead.

For a *TStringField, AsInteger* converts an integer to a string on assigning a value to the field, and converts a string to an integer when reading from the field.

See also

Data Access Components Hierarchy, *IntToStr* function, *StrToInt* function, *Value* property

Assign method

Applies to

TBitmap, *TBrush*, *TClipboard*, *TControlScrollBar*, *TFieldDef*, *TFieldDefs*, *TFont*, *TIcon*, *TIndexDef*, *TIndexDefs*, *TMetafile*, *TParam*, *TParams*, *TPen*, *TPicture*, *TStringList*, *TStrings* objects

TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

For the Clipboard

Declaration:

```
procedure Assign(Source: TPersistent);
```

The *Assign* method assigns the object specified by the *Source* parameter to the Clipboard. If the object is a *TGraphic*, *TBitmap*, *TPicture* or *TMetafile*, the image will be copied to the Clipboard in the corresponding format (either CF_BITMAP or CF_METAFILE). For example, the following code copies the bitmap from a bitmap object named *Bitmap1* to the Clipboard:

```
  Clipboard.Assign(Bitmap1);
```

To retrieve an object from the Clipboard, simply use the *Assign* method of an appropriate object. For example, if a bitmap is on the Clipboard, the following code copies it to a bitmap object named *Bitmap1*:

```
  Bitmap1.Assign(Clipboard);
```

Example

The following code copies the bitmap of a speed button named *SpeedButton1* to the Clipboard:

```
  Clipboard.Assign(SpeedButton1.Glyph);
```

See also

AsText property, *Clipboard* variable, *HasFormat* property

For field definitions

Declaration

```
procedure Assign(FieldDefs: TFieldDefs);
```

Assign creates a new set of *TFieldDef* objects in *Items* from the *FieldDefs* parameter. Any previously entries in *Items* are freed.

For index definitions

Declaration

```
procedure Assign(IndexDefs: TIndexDefs);
```

Assign creates a new set of *TIndexDef* objects in *Items* from the *IndexDefs* parameter. Any previously entries in *Items* are freed.

For field components

Declaration

```
procedure Assign(Source: TPersistent);
```

Assign copies data from one field to another. Both fields must be valid and have the same *DataType* and *Size*, and the *DataSize* of *Source* must be 255 bytes or less.

The restrictions on type compatibility and size do not apply to *TBlobField*, *TBytesField*, *TGraphicField*, *TMemoField,* and *TVarBytesField*. For a *TBlobField*, *TBytesField* or *TVarBytesField*, the source can be a *TBlobField*, *TBytesField*, *TVarBytesField*, *TMemoField* component, *TGraphicField* component, *TMemoField* component, *TStrings* object, *TPicture* or *TGraphicField*.

Examples

```
{ Copy one date-time field to another }
DateTimeField1.Assign(DateTimeField2);

{ Copy a graphic field to a blob field }
BlobField1.Assign(GraphicField1);

{Copy strings in a TMemo to a TMemoField}
MemoField1.Assign(Memo1.Lines);
```

See also

DataType property, *Size* property

For TParam objects

Declaration

```
procedure Assign(Param: TParam);
```

The *Assign* method transfers all of the data contained in the *Param* parameter to the *TParam* object that calls it. If, however, you have specified a value for the *ParamType* property of the *TParam* object that calls *Assign*, the data in the *Param* parameter will *not* be assigned to the *TParam* object.

Example

```
{ Copy the CustNo parameter from Query1 to Query2 }
```

```
Query2.ParamByName('CustNo').Assign(Query1.ParamByName('CustNo'));
```

See also

ParamType property, *DataType* property, *AssignField* method

For TParams objects

Declaration

```
procedure Assign(Source: TPersistent);
```

If *Source* is another *TParams* object, *Assign* discards any current parameter information and replaces it with the information from *Source*. If *Source* is any other type of object, *Assign* calls its inherited method. Use this method to save and restore a set of parameter information or copy another object's information.

Example

```
var SavedParams: TParams;
...
{ Initialize SavedParams }
SavedParams := TParams.Create;
{ Save the parameters for Query1 }
SavedParams.Assign(Query1.Parameters);
{ Do something with Query1 }
...
{ Restore the parameters to Query1 }
Query1.Parameters.Assign(SavedParams);
SavedParams.Free;
```

See also

AssignValues method

For other objects

Declaration

```
procedure Assign(Source: TPersistent);
```

The *Assign* method assigns one object to another. The general form of a call to *Assign* is

```
Destination.Assign(Source);
```

which tells the *Destination* object to assign the contents of the *Source* object to itself.

In general, the statement "Destination := Source" is *not* the same as the statement "Destination.Assign(Source)". The statement "Destination := Source" makes *Destination* reference the same object as *Source*, whereas "Destination.Assign(Source)" copies the contents of the object references by *Source* into the object referenced by *Destination*.

If *Destination* is a property of some object, however, and that property is not a reference to another object (such as the *ActiveControl* property of a form, or the *DataSource* property of a data-aware control), then the statement "Destination := Source" is the same as "Destination.Assign(Source)". Consider these statements:

```
Button1.Font := Button2.Font;
ListBox1.Items := Memo1.Lines;
Table1.Fields[0] := Query1.Fields[2];
```

They correspond to these statements:

```
Button1.Font.Assign(Button2.Font);
ListBox1.Items.Assign(Memo1.Lines);
Table1.Fields[0].Assign(Query1.Fields[2]);
```

The actions performed by *Assign* depend on the actual types of *Destination* and *Source*. For example, if *Destination* and *Source* are string objects (*TStrings*), the strings contained in *Source* are copied into *Destination*. Likewise, if *Destination* and *Source* are bitmaps (*TBitmap*), the bitmap contained in *Source* is copied into *Destination*.

Although the compiler allows any two *TPersistent* objects to be used in a call to *Assign*, the call succeeds at run time only if the objects involved "know" how to perform an assignment. For example, if *Destination* is a button (*TButton*) and *Source* is an edit box (*TEdit*), the call to *Assign* raises an *EConvertError* exception at run time.

An object of one type can always be assigned to another object of the same type. In addition, *Assign* supports the following special cases:

- If *Destination* is of type *TPicture* then *Source* can be of type *TBitmap*, *TIcon*, or *TMetafile*.
- If *Destination* is of type *TBitmap*, *TIcon*, or *TMetafile* then *Source* can be of type *TPicture* if the *Graphic* property of the picture is of the same type as *Destination*.
- If *Destination* is of type *TBlobField* then *Source* can be of type *TBitmap*, *TPicture*, or *TStrings*.

Example

The following code changes the properties of a label’s font so that they match the properties of the button’s font when the user clicks the button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Font.Assign(Button1.Font);
end;
```

AssignCrt procedure — WinCrt

Declaration

```
procedure AssignCrt(var f: Text);
```

The *AssignCrt* procedure associates a text file with the CRT window.

AssignCrt works exactly like the *Assign* standard procedure except that no file name is specified. Instead, the text file associates with the CRT window, which emulates a text-based CRT in the Windows environment. Subsequent *Write* and *Writeln* operations on the file write to the CRT window, and *Read* and *Readln* operations read from the CRT window.

This allows faster output (and input) than would normally be possible using standard output (or input).

See also

AssignFile procedure, *Read* procedure, *Readln* procedure, *Write* procedure, *Writeln* procedure

Assigned function — System

Declaration

```
function Assigned(var P): Boolean;
```

The *Assigned* function tests if a pointer or procedural variable is **nil** (unassigned).

P must be a variable reference of a pointer or procedural type. *Assigned*(*P*) corresponds to the test *P*<> **nil** for a pointer variable, and @*P* <> **nil** for a procedural variable.

Assigned returns *True* if *P* is **nil**, *False* otherwise.

Note *Assigned* can't detect a "stale" pointer—that is, one that isn't **nil** but no longer points to valid data. For example, in the following code, *Assigned* won't detect the fact that *P* isn't valid.

Example

```
var P: Pointer;
begin
  P := nil;
  if Assigned (P) then Writeln ('You won''t see this');
  P := @P;
  if Assigned (P) then Writeln ('You''ll see this');
end;
```

AssignField method

Applies to

TParam object

Declaration

```
procedure AssignField(Field: TField);
```

The *AssignField* method transfers the *DataType* value and *Name* from *Field*. Use *AssignField* to set a parameter from a *TField* component.

Example

```
{ Copy the CustNo field value from Query1 to the CustNo parameter of Query2 }
Query2.ParamByName('CustNo').AssignField(Query1.FieldNyName('CustNo'));
```

AssignFile procedure System

Declaration

```
procedure AssignFile(var F, String);
```

To avoid scope conflicts, *AssignFile* replaces *Assign* in Delphi. However, for backward compatibility you can still use *Assign*.

The *AssignFile* procedure associates the name of an external file with a file variable.

F is a file variable of any file type, and **string** is a string-type expression or an expression of type *PChar* if extended syntax is enabled. All further operations on *F* operate on the external file name.

After calling *AssignFile*, *F* is associated with the external file until *F* is closed.

When the *String* parameter is empty, *F* associates with the standard input or standard output file.

If assigned an empty name, after a call to *Reset* (*F*), *F* refers to the standard input file, and after a call to *Rewrite* (*F*), *F* refers to the standard output file.

Do not use *AssignFile* on a file variable that is already open.

A file name consists of a path of zero or more directory names separated by backslashes, followed by the actual file name:

```
Drive:\DirName\...\DirName\FileName
```

If the path begins with a backslash, it starts in the root directory; otherwise, it starts in the current directory.

Drive is a disk drive identifier (A–Z). If *Drive* and the colon are omitted, the default drive is used. \DirName\...\DirName is the root directory and subdirectory path to the file name. *FileName* consists of a name of up to eight characters, optionally followed by a period and an extension of up to three characters. The maximum length of the entire file name is 79 characters.

Example

```
var
  F: TextFile;
  S: string;
begin
  if OpenDialog1.Execute then                              { Display Open dialog box }
```

```
    begin
      AssignFile(F, OpenDialog1.FileName);               { File selected in dialog box }
      Reset(F);
      Readln(F, S);                               { Read the first line out of the file }
      Edit1.Text := S;                                  { Put string in a TEdit control }
      CloseFile(F);
    end;
  end;
```

See also

Append procedure, *FileClose* procedure, *Reset* procedure, *Rewrite* procedure

AssignPrn procedure Printers

Declaration

```
procedure AssignPrn(var F: Text);
```

The *AssignPrn* procedure assigns a text-file variable to the printer. After the variable is assigned, your application must call the *Rewrite* procedure. Then any time an application writes data to *F*, the text-file variable, the data is sent to the printer using the pen and font of the *Canvas* property.

Example

This code prints a line of text on the printer when the user clicks the button on the form:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyFile: TextFile;
begin
  AssignPrn(MyFile);
  Rewrite(MyFile);
  Writeln(MyFile, 'Print this text');
  System.CloseFile(MyFile);
end;
```

AssignStr procedure SysUtils

Declaration

```
procedure AssignStr(var P: PString; const S: string);
```

AssignStr assigns a new dynamically allocated string to the given string pointer. *AssignStr* corresponds to the statement DisposeStr(P) followed by the statement P := NewStr(S). Note that *P* must be NIL or contain a valid string pointer before calling *AssignStr*. In other words, *AssignStr* cannot be used to initialize a string pointer variable.

Example

```
var
  P: PString;
begin
  P := NewStr('First string');   { Allocate and point to 'First string' }
  AssignStr(P, 'Second string'); { Dispose of 'First string', allocate and point to }
                                 { 'Second string' }
  DisposeStr(P);    { Dispose of 'Second string' }
end;
```

See also

DisposeStr procedure

AssignValue method

Applies to

TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
procedure AssignValue(const Value: TVarRec);
```

The *AssignValue* method sets the field to *Value* using one of the *AsInteger*, *AsBoolean*, *AsString* or *AsFloat* properties, depending on the type of *Value*. If *Value* is of type *TObject* or a *TObject* descendant, *AssignValue* uses the *Assign* method to transfer the information.

Example

```
Field1.AssignValue(‘new string’);
```

AssignValues method

Applies to

TParams object

Declaration

```
procedure AssignValues(Value: TParams);
```

For each entry in *Items*, the *AssignValues* method attempts to find a parameter with the same *Name* property in *Value*. If successful, the parameter information (type and current data) from the *Value* parameter is assigned to the *Items* entry. Entries in *Items* for which no match is found are left unchanged.

Example

```
var SavedParams: TParams;
...
{ Initialize SavedParams }
SavedParams := TParams.Create;
{ Save the parameters for Query1 }
SavedParams.Assign(Query1.Parameters);
{ Do something with Query1 }
...
{ Restore the parameters to Query1 }
Query1.Parameters.AssignValues(SavedParams);
SavedParams.Free;
```

AsSmallInt property

Applies to
TParam object

Declaration

```
procedure SetAsSmallInt(Value: Longint);
```

Assigning a value to the *AsSmallInt* property sets the *DataType* property to *fsSmallInt* and save the value as the current data for the parameter. Accessing the *AsSmallInt* property attempts to convert the current data to a *SmallInt* value and returns that value.

AsString property

Applies to
TParam object; *TBCDField*, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

For TParam objects

Declaration

```
property AsString: string;
```

Assigning a value to the *AsString* property sets the *DataType* property to *ftString* and saves the value as the current data for the parameter. Accessing the *AsString* property attempts to convert the current data to a string value and returns that value.

See also
DateToStr function, *DateTimeToStr* function, *FloatToStr* function, *IntToStr* function, *TFieldType* type, *TimeToStr* function

For field components

Declaration

```
property AsString: string;
```

Run-time only. This a conversion property. For a *TStringField*, *AsString* can be used to read or set the value of the field as a string, but *Value* should be used for this purpose instead.

For *TBCDField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TIntegerField*, *TSmallintField*, *TTimeField*, and *TWordField*, *AsString* converts a string to the appropriate type on inserting to or updating the field, and converts the type to a string when reading from the field.

For *TBooleanField*, on insert or update *AsString* sets the value to *True* if the text begins with the letter "Y", "y", "T" or "t" and to *False* otherwise. When reading from a Boolean field, AsString returns 'T' or 'F'.

For a *TMemoField*, *AsString* should only be used to read from the field. It sets the string value to '(Memo)'. An exception is raised if *AsString* is used to write to a *TMemoField*.

For a *TGraphicField*, *AsString* should only be used to read from the field. It sets the string value to '(Graphic)'. An exception is raised if *AsString* is used to write to a *TGraphicField*.

For a *TBlobField*, *AsString* should only be used to read from the field. It sets the string value to '(Blob)'. An exception is raised if *AsString* is used to write to a *TBlobField*.

For a *TBytesField*, *AsString* should only be used to read from the field. It sets the string value to '(Bytes)'. An exception is raised if *AsString* is used to write to a *TBytesField*.

For a *TVarBytesField*, *AsString* should only be used to read from the field. It sets the string value to '(Var Bytes)'. An exception is raised if *AsString* is used to write to a *TVarBytesField*.

Note When working with *TMemoField*, *TGraphicField*, or *TBlobField*, use the *Assign*, *LoadFromFile*, or *LoadFromStream* methods to write to a field, and *Assign*, *SaveToFile*, or *SaveToStream* methods to read from a field.

AsText property

Applies to

TClipboard object

Declaration

```
property AsText: String;
```

Run-time only. The *AsText* property returns the current contents of the Clipboard as a string. The Clipboard must contain a string or an exception occurs.

You can also use the *AsText* property to place a copy of a string on the Clipboard. Assign a string as the value of *AsText*.

The string value of the *AsText* property is limited to 255 characters. If you need to set and retrieve more than 255 characters, use the *SetTextBuf* and *GetTextBuf* Clipboard methods.

If the Clipboard contains a string, this expression is *True*:

```
Clipboard.HasFormat(CF_TEXT)
```

Example

The following code retrieves the contents of the Clipboard as a string and displays the value in a label:

```
begin
  Label1.Caption := Clipboard.AsText;
end;
```

See also

Clipboard variable, *HasFormat* method

AsTime property

Applies to

TParam object

Declaration

```
property AsTime: TDateTime;
```

Assigning a value to the *AsTime* property sets the *DataType* property to *ftTime* and saves the value as the current data for the parameter. Accessing the *AsTime* property attempts to convert the current data to a *TDateTime* value and returns that value.

See also

StrToDateTime function, *TDateTime* type, *TFieldType* type

AsWord property

Applies to

TParam object

Declaration

```
property AsWord: Longint;
```

Assigning a value to the *AsWord* property sets the *DataType* property to *ftWord* and saves the value as the current data for the parameter. Accessing the *AsWord* property attempts to convert the current data to a Longint value and returns that value.

See also
TFieldType type

AutoActivate property

Applies to
TOLEContainer component

Declaration

```
property AutoActivate: TAutoActivate;
```

AutoActivate determines how an object in an OLE container can be activated. These are the possible values:

Value	Meaning
aaManual	The OLE object must be manually activated. To activate the OLE object manually, set the *Active* property to *True*.
aaGetFocus	The user activates the OLE object by clicking the OLE container or pressing Tab until focus shifts to the OLE container. If the OLE container has a *TabOrder* of 0, the OLE container initially receives focus but the OLE object won't be activated.
aaDoubleClick	The user activates the OLE object by double-clicking the OLE container, or pressing Enter when the container has focus. An *OnDblClick* event is generated immediately after the OLE server application is activated.

Example
The following code sets the activation method of *OLEContainer1* to *aaManual*, then activates *OLEContainer1*:

```
OLEContainer1.AutoActivate := aaManual;
OLEContainer1.Active := True;
```

See also
OnActivate event

AutoCalcFields property

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
property AutoCalcFields: Boolean;
```

The *AutoCalcFields* property determines when *OnCalcFields* is called. *OnCalcFields* is always called whenever an application retrieves a record from the database. If *AutoCalcFields* is *True*, then *OnCalcFields* is called also whenever a field in a dataset is edited.

If *AutoCalcFields* is *True*, *OnCalcFields* should not perform any actions that modify the dataset (or the linked dataset if it is part of a master-detail relationship), because this can lead to recursion. For example, if *OnCalcFields* performs a *Post*, and *AutoCalcFields* is *True*, then *OnCalcFields* will be called again, leading to another *Post*, and so on.

AutoDisplay property

Applies to

TDBImage, *TDBMemo* component

Declaration

```
property AutoDisplay: Boolean;
```

The value of the *AutoDisplay* property determines whether to automatically display the contents of a memo or graphic BLOB in a database memo (*TDBMemo*) or database image (*TDBImage*) control.

If *AutoDisplay* is *True* (the default value), the control automatically displays new data when the underlying BLOB field changes (such as when moving to a new record).

If *AutoDisplay* is *False*, the control clears whenever the underlying BLOB field changes. To display the data, the user can double-click on the control or select it and press *Enter*. In addition, by calling the *LoadMemo* method of a database memo or the *LoadPicture* method of a database image you can ensure that the control is showing data.

You might want to change the value of *AutoDisplay* to *False* if the automatic loading of BLOB fields seems to take too long.

Example

The following code displays the text BLOB in *DBMemo1*.

```
DBMemo1.AutoDisplay := True;
```

See also

LoadMemo method, *LoadPicture* method

AutoEdit property

Applies to

TDataSource component

Declaration

```
property AutoEdit: Boolean;
```

AutoEdit determines if data-aware controls connected to *TDataSource* automatically place the current record into edit mode by calling the table's *Edit* method when the user begins typing within one of them. A*utoEdit* is *True* by default; set it to *False* to protect the data from being unintentionally modified. When *AutoEdit* is *False*, you can still call the *Edit* method to modify a field.

AutoEnable property

Applies to

TMediaPlayer component

Declaration

```
property AutoEnable: Boolean;
```

The *AutoEnable* property determines whether the media player automatically enables and disables individual buttons in the component.

If *AutoEnable* is *True*, the media player automatically enables or disables its control buttons. The media player determines which buttons to enable or disable by the current mode specified in the *Mode* property, and the current multimedia device type specified in the *DeviceType* property.

AutoEnable overrides the *EnabledButtons* property. The buttons enabled or disabled automatically by the media player supersede any buttons enabled or disabled with *EnabledButtons*.

If *AutoEnable* is *False*, the media player does not enable or disable buttons. You must enable or disable buttons with the *EnabledButtons* property.

The following table shows whether buttons are automatically enabled or disabled for each device mode:

Button	Play	Record	Pause	Stop	Not Open
Back	Enabled	Enabled	Enabled	Enabled	Disabled
Eject	Enabled	Enabled	Enabled	Enabled	Disabled
Next	Enabled	Enabled	Enabled	Enabled	Disabled
Pause	Enabled	Enabled	Enabled	Disabled	Disabled
Play	Disabled	Disabled	Enabled	Enabled	Disabled
Prev	Enabled	Enabled	Enabled	Enabled	Disabled
Record	Disabled	Disabled	Enabled	Enabled	Disabled
Step	Enabled	Enabled	Enabled	Enabled	Disabled
Stop	Enabled	Enabled	Disabled	Disabled	Disabled

Example

The following code causes all of the buttons of *MediaPlayer1* to become disabled when a bitmap button is clicked:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  with MediaPlayer1 do begin
    AutoEnable := False;
```

```
      EnabledButtons := [];
    end;
  end;
```

See also
AutoOpen property

AutoMerge property

Applies to
TMainMenu component

Declaration

```
property AutoMerge: Boolean;
```

The *AutoMerge* property determines if the main menus (*TMainMenu*) of forms other than the main form merge with the main menu of the main form in non-MDI applications at run time. The default value is *False*. To merge the form's menus with the main menu in the main form, set the *AutoMerge* property of each main menu you want merged to *True*. Make sure that the *AutoMerge* property of the main menu you are merging with other menus remains *False*. How menus merge depends on the value of the *GroupIndex* property for each menu item.

If the application is an MDI application (the *FormStyle* properties are set so the main form is a parent form and subsequent forms are child forms), menu merging occurs automatically and you don't need to use the *AutoMerge* property. In an MDI application, you should be sure that the *AutoMerge* value for the main menu of the parent form is *False*, or else the menu bar of the parent form disappears when a child form appears.

Example
This example uses two forms with a main menu and a button on each form. Using the Object Inspector, set the *GroupIndex* value for each menu item on the menu bar in the second form to a number greater than 0. When the application runs and the user clicks the button on the first form, the main menu on the second form merges with the main menu of the first form. When the user clicks the button on the second form, the form closes.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.MainMenu1.AutoMerge := True;
  Form2.Show;
end;
```

This is the code for the button-click event handler on the second form:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  Close;
end;
```

To run this example, you must add *Unit2* to the **uses** clause of *Unit1*.

See also
GroupIndex property, *Merge* method, *Unmerge* method

AutoOpen property

Applies to
TMediaPlayer component

Declaration

```
property AutoOpen: Boolean;
```

The *AutoOpen* property determines if the media player is opened automatically when the application is run. If *AutoOpen* is *True*, the media player attempts to open the multimedia device specified by the *DeviceType* property (or *FileName* if *DeviceType* is *dtAutoSelect*) when the form containing the media player component is created at run time. If *AutoOpen* is *False*, the device must be opened with a call to the *Open* method. *AutoOpen* defaults to *True*.

If an error occurs when opening the device, an exception of type *EMCIDeviceError* is raised which contains the error message. Upon completion, a numerical error code is stored in the *Error* property, and the corresponding error message is stored in the *ErrorMessage* property.

The *Wait* property determines whether control is returned to the application before opening the multimedia device. The *Notify* property determines whether opening the device generates an *OnNotify* event.

Example
The following code opens *MediaPlayer1* if *AutoOpen* was not set to *True*. This code assumes that an appropriate value was specified for *FileName* at design time.

```
with MediaPlayer1 do
  if not AutoOpen then
    Open;
```

See also
Close method

AutoPopup property

Applies to
TPopupMenu component

Declaration

```
property AutoPopup: Boolean;
```

The *AutoPopup* property determines if the pop-up menu appears when the user clicks the right mouse button on the component that has this menu specified as the value of its *PopupMenu* property. If *AutoPopup* is *True*, a right click displays the pop-up menu. If *AutoPopup* is *False*, the menu won't appear when the user clicks the right mouse button. The default value is *True*.

To display a pop-up menu when *AutoPopup* is *False*, you must use the *Popup* method.

Example

The following prevents the pop-up menu from appearing when the user clicks the right mouse button:

```
   PopupMenu1.AutoPopup := False;
```

See also

OnPopup event, *Popup* method

AutoRewind property

Applies to

TMediaPlayer component

Declaration

```
property AutoRewind: Boolean;
```

The *AutoRewind* property determines if the media player control rewinds before playing or recording.

If *AutoRewind* is *True* and the current position is at the end of the medium, *Play* or *StartRecording* moves the current position to the beginning of the medium before playing or recording. If *AutoRewind* is *False*, the user must click the Prev button or your code must call *Previous* to move to the beginning.

Note If values have been assigned to *StartPos* or *EndPos* or if the multimedia device uses tracks, *AutoRewind* has no effect on playing or recording. When you call *Play* or *StartRecording*, the current position remains at the end of the medium.

Example

The following code plays *MediaPlayer*. If *AutoRewind* is *False*, *Previous* is called to rewind after *Play* is finished.

```
   MediaPlayer.Wait := True;
   MediaPlayer.Play;
   if not MediaPlayer.AutoRewind then MediaPlayer.Previous;
```

See also
Rewind method

AutoScroll property

Applies to
TForm, *TScrollBox*, *TTabSet* components

For tab set controls

Declaration

```
property AutoScroll: Boolean;
```

The *AutoScroll* property determines if scroll buttons automatically appear in a tab set control if there isn't room in the control to display all the tabs.

If *AutoScroll* is *False*, your application can still access tabs that aren't visible by using the *FirstIndex* or *TabIndex* properties at design time or run time, but the user can't click on the tabs with the mouse at run time.

Example
This code displays scroll buttons in the tab set control if all the tabs aren't visible:

```
TabSet11.AutoScroll := True;
```

See also
FirstIndex property, *TabIndex* property

For forms and scroll boxes

Declaration

```
property AutoScroll: Boolean;
```

The *AutoScroll* property determines if scroll bars appear on the form when the form is not large enough to display all the controls it contains. If *AutoScroll* is *True*, the scroll bars appear automatically when necessary. For example, if the user resizes the form so that it is smaller and some controls are partially obscured, scroll bars appear. If *AutoScroll* is *False*, no scroll bars appear.

Example
This example uses a label on a form. When the form becomes active, the label displays a message informing the user whether scroll bars will be available if the form is resized so that not all controls are fully visible.

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

```
    if AutoScroll then
      Label1.Caption := 'Scroll bars might appear!'
    else
      Label1.Caption := 'No scroll bars will appear';
  end;
```

See also
HorzScrollBar property, *ScrollInView* method, *VertScrollBar* property

AutoSelect property

Applies to
TDBEdit, *TDBLookupCombo*, *TEdit*, *TMaskEdit* components

Declaration

```
property AutoSelect: Boolean;
```

The value of the *AutoSelect* property determines if the text in the edit box or combo box is automatically selected when the user tabs to the control. If*AutoSelect* is *True*, the text is selected. If *AutoSelect* is *False*, the text is not selected.

The default value is *True*.

Example
This example uses an edit box and a check box on a form. Set the caption of the check box to ‘AutoSelect text’. When the user checks the check box, text is automatically selected each time the user tabs to the edit box. If the user unchecks the check box, text is no longer selected automatically when the user tabs to the edit box.

```
  procedure TForm1.CheckBox1Click(Sender: TObject);
  begin
    if CheckBox1.Checked then
      Edit1.AutoSelect := True
    else
      Edit1.AutoSelect := False;
    end;
  end;
```

See also
AutoSize property, *SelLength* property, *SelStart* property, *SelText* property, *Text* property

AutoSize property

Applies to
TDBEdit, *TDBText*, *TEdit*, *TImage*, *TLabel*, *TMaskEdit*, *TOLEContainer* components

The *AutoSize* property determines if the component automatically resizes to match the size of its contents.

For images

Declaration

```
property AutoSize: Boolean;
```

When the *AutoSize* property is *True*, the image control resizes to accommodate the image it contains (specified by the *Picture* property). When *AutoSize* is *False*, the image control remains the same size, regardless of the size of the image. If the image control is smaller than the image, only the portion of the picture that fits inside the image component will be visible.

The default value is *False*.

Note You must remember to set the *AutoSize* property to *True* before loading the picture, or *AutoSize* has no effect.

To resize the image to fill an image control completely when the control is larger than the native size of the image, use the *Stretch* property.

Example

This example uses an image control and a button. Resize the image control so that it is too small to display the entire bitmap. When the user clicks the button, the bitmap is loaded into the image control, and the image control resizes to display the bitmap in its entirety.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Image1.AutoSize := True;
  Image1.Picture.LoadFromFile('c:\windows\arches.bmp');
end;
```

See also

LoadFromFile method, *Stretch* property

For edit boxes and database lookup combo boxes

Declaration

```
property AutoSize: Boolean;
```

When the *AutoSize* property is *True*, the height of the edit box changes to accommodate font size changes to the text. When *AutoSize* is *False*, the edit box remains the same size, regardless of any font changes. The default value is *True*.

If an edit box has no border, changing the value of *AutoSize* has no effect. In other words, the *BorderStyle* property must have a value of *bsSingle*.

Example
This example uses an edit box, a label, and a button on a form. When the user clicks the button, the font in the edit box enlarges, and the edit box enlarges also to accommodate the larger font size.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.AutoSize := True;
  Edit1.Font.Size := 20;
  Label1.Caption := 'The edit box is bigger now';
end;
```

See also
Font property

For label and database text components

Declaration

```
property AutoSize: Boolean;
```

When the *AutoSize* property is *True*, the label component resizes to the width and length of the current string in the label's *Caption* property. If you type text for a label while *AutoSize* is *True*, the label grows for each character you type. If you change the font size of the text, the label resizes to the new font size. When *AutoSize* is *False*, the size of the label is not affected by the length of the string in its *Caption* property.

The default value of *AutoSize* is *True*.

Example
The following code keeps the size of the label control constant, even though the length of the label's caption changes. As a result, the caption of the label is probably too long to display in the label when the user clicks the button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.AutoSize := False;
  Label1.Caption := 'This string is too long as the caption of this label';
end;
```

See also
WordWrap property

For OLE containers

Declaration

```
property AutoSize: Boolean;
```

Autosize determines whether the OLE container automatically resizes to the size of the OLE object it contains.

If *Autosize* is *True*, the OLE container adopts the shape of the OLE object at run time. If the user activates the object and changes its size, the OLE container resizes to the new size. Setting *Autosize* to *True* may unintentionally cause the OLE container to resize to a shape that exists outside the client area of the form or over other controls.

If *Autosize* is *False*, the shape of the OLE container remains constant. The picture of the OLE object is clipped to fit in the shape of the OLE container when deactivated. This clipping does not affect the OLE object itself, however. The user can still access the entire OLE object when it is activated.

Example

The following code resizes *OLEContainer1* automatically when activated:

```
  OLEContainer1.AutoSize := True;
```

AutoTracking typed constant — WinCrt

Declaration

```
const AutoTracking: Boolean = True;
```

The *AutoTracking* typed constant controls automatic cursor tracking in the CRT window.

When *AutoTracking* is *True*, the CRT window automatically scrolls to ensure that the cursor is visible after each *Write* and *Writeln*.

If *AutoTracking* is *False*, the CRT window will not scroll automatically, and text written to the window might not be visible to the user.

AutoUnload property

Applies to

TReport component

Declaration

```
property AutoUnload: Boolean;
```

The *AutoUnload* property determines whether ReportSmith Runtime unloads from memory when you have finished running a report.

If *AutoUnload* is *True*, ReportSmith Runtime unloads as soon as the report is finished running.

If *AutoUnload* is *False*, ReportSmith Runtime remains in memory. For example, you can create an application that includes a menu item that runs a report. After the report runs, you want ReportSmith Runtime to stay in memory so the report can be quickly rerun again. To remove ReportSmith Runtime from memory when *AutoUnload* is *False*, you must then call the *CloseApplication* method.

Example

The following code sets *AutoUnload* to *False*, so that *Report1* can be run twice using two different variables. After the second run, ReportSmith is unloaded by a call to *CloseApplication*.

```
Report1.AutoUnload := False;
if Report1.SetVariable('FName', 'Linda') then
  Report1.Run;
if Report1.SetVariable('LName', 'King') then
  Report1.Run;
Report1.CloseApplication(False);
```

Back method

Applies to

TMediaPlayer component

Declaration

```
procedure Back;
```

The *Back* method steps backward a number of frames (determined by the value of the *Frames* property) in the currently loaded medium. *Back* is called when the Back button on the media player control is clicked at run time.

Upon completion, *Back* stores a numerical error code in the *Error* property and the corresponding error message in the *ErrorMessage* property.

The *Wait* property determines whether control is returned to the application before the *Back* method has been completed. The *Notify* property determines whether *Back* generates an *OnNotify* event.

Example

The following example lets the user pick an .AVI video file using *OpenDialog1* and opens that file in *MediaPlayer1*. Then, the Back button can be used to step backward through the .AVI clip. You could use this to hide *MediaPlayer1* and design your own user interface for the media player.

```
procedure TForm1.OpenClick(Sender: TObject);
begin
  OpenDialog1.Filename := '*.*';
  if OpenDialog1.Execute then
  begin
    MediaPlayer1.Filename := OpenDialog1.Filename;
    MediaPlayer1.Open;
  end;
end;
```

```
procedure TForm1.BackClick(Sender: TObject);
begin
  MediaPlayer1.Back;
end;
```

See also
Capabilities property, *OnClick* event, *Rewind* method, *Step* method

BackgroundColor property

Applies to
TTabSet component

Declaration

```
property BackgroundColor: TColor;
```

The *BackgroundColor* property determines the background color of the tab set control. The background area of the tab set control is the area between the tabs and the border of the control. For a list of possible color values, see the *Color* property.

Example
This code changes the background color of the tab set control:

```
TabSet1.BackgroundColor := clBackground;
```

See also
DitherBackground property

BatchMove method

Applies to
TTable component

Declaration

```
function BatchMove(ASource: TDataSet; AMode: TBatchMode): Longint;
```

The *BatchMove* method copies, appends, updates, or deletes records in the *TTable*. *ASource* is a *TTable* linked to a database table containing the source records. *AMode* is the copy mode; it can be any of the elements of *TBatchMode*: *batAppend*, *batUpdate*, *batAppendUpdate*, *batDelete*, or *batCopy*.

BatchMove returns the number of records operated on.

Example

```
Table1.BatchMove(Table2, batAppend);
```

See also
TBatchMove component

BeforeCancel event

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
property BeforeCancel: TDataSetNotifyEvent;
```

The *BeforeCancel* event is activated at the beginning of a call to the *Cancel* method. This event is the first action taken by *Cancel*. If the dataset is not in Edit state or there are no changes pending, then *Cancel* will not activate the *BeforeCancel* event.

By assigning a method to this property, you can take any special actions required by the event. By raising an exception in this event handler, you can prevent the *Cancel* operation from occurring.

See also
AfterCancel event

BeforeClose event

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
property BeforeClose: TDataSetNotifyEvent;
```

The *BeforeClose* event is activated before the *dataset* is closed, either by calling the *Close* method or by setting the *Active* property to *False*. This event is the first action taken by *Close*.

By assigning a method to this property, you can take any special actions required by the event. By raising an exception in this event handler, you can prevent the *Close* operation from occurring.

See also
AfterClose event

BeforeDelete event

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
property BeforeDelete: TDataSetNotifyEvent;
```

The *BeforeDelete* event is activated when the *dataset* begins a call to *Delete*. This event is the first action taken by the *Delete* method.

By assigning a method to this property, you can take any special actions required by the event. By raising an exception in this event handler, you can prevent the *Delete* operation from occurring.

See also
AfterDelete event

BeforeEdit event

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
property BeforeEdit: TDataSetNotifyEvent;
```

The *BeforeEdit* event is activated when the *dataset* begins a call to the *Edit* method. This event is the first action taken by *Edit*.

By assigning a method to this property, you can take any special actions required by the event. By raising an exception in this event handler, you can prevent the *Edit* operation from occurring.

See also
AfterEdit event

BeforeInsert event

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
property BeforeInsert: TDataSetNotifyEvent;
```

The *BeforeInsert* event is activated when the *dataset* begins a call to the *Insert* or *Append* methods. This event is the first action taken by *Insert* or *Append*.

By assigning a method to this property, you can take any special actions required by the event. By raising an exception in this event handler, you can prevent the *Insert* operation from occurring.

See also

AfterInsert event

BeforeOpen event

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
property BeforeOpen: TDataSetNotifyEvent;
```

The *BeforeOpen* event is activated before the *dataset* is opened, either by calling the *Open* method or by setting the *Active* property to *True*. This event is the first action taken by the *Open* method. Typically, the *BeforeOpen* event handler opens any private lookup tables used by other event handlers in the dataset.

By assigning a method to this property, you can take any special actions required by the event. By raising an exception in this event handler, you can prevent the *Open* operation from occurring.

See also

AfterOpen event

BeforePost event

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
property BeforePost: TDataSetNotifyEvent;
```

The *BeforePost* event is activated at the beginning of a call to the *Post* method. This event is the first action taken by the *Post* method, after it calls the *UpdateRecord* method to reflect any changes made to the record by data controls. The *BeforePost* event can be used to validate a record before it is posted. By raising an exception, a *BeforePost* event handler can prevent the posting of an invalid record.

By assigning a method to this property, you can take any special actions required by the event. By raising an exception in this event handler, you can prevent the *Post* operation from occurring.

See also
AfterPost event

BeginDoc method

Applies to
TPrinter object

Declaration

```
procedure BeginDoc;
```

The *BeginDoc* method sends a print job to the printer. If the print job is sent successfully, the application should call *EndDoc* to end the print job. The print job won't actually start printing until *EndDoc* is called.

To use the *BeginDoc* method, you must add the *Printers* unit to the **uses** clause of your unit.

Example
This code prints a rectangle on the default printer:

```
begin
  Printer.BeginDoc;                               { begin to send print job to printer }
  Printer.Canvas.Rectangle(20,20,1000,1000);     { draw rectangle on printer's canvas }
  Printer.EndDoc;                          { EndDoc ends and starts printing print job }
end;
```

To use the *BeginDoc* method, you must add the *Printers* unit to the **uses** clause of your unit.

See also
Abort method, *Printer* variable

BeginDrag method

Applies to
All controls

Declaration

```
procedure BeginDrag(Immediate: Boolean);
```

The *BeginDrag* method starts the dragging of a control. If the *Immediate* parameter is *True*, the mouse pointer changes to the value of the *DragCursor* property and dragging begins immediately. If *Immediate* is *False*, the mouse pointer doesn't change to the value of the *DragCursor* property and dragging doesn't begin until the user moves the mouse pointer a short distance (5 pixels). This allows the control to accept mouse clicks without beginning a drag operation.

Your application needs to call the *BeginDrag* method to begin dragging only when the *DragMode* property value for the control is *dmManual*.

Example

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Button1.DragMode = dmManual then
    Button1.BeginDrag(True);
end;
```

See also

DragMode property, *EndDrag* method, *OnDragDrop* event, *OnDragOver* event, *OnEndDrag* event

BeginUpdate method

Applies to

TStringList, *TStrings* objects; *TOutline* component

Declaration

```
procedure BeginUpdate;
```

The *BeginUpdate* method prevents the updating of the outline or string object until the *EndUpdate* method is called. For string objects, *BeginUpdate* prevents the screen from being repainted when new strings are added. For outlines, *BeginUpdate* prevents the screen from being repainted and prevents outline items from being reindexed when new items are added, deleted, or inserted. Outline items affected by the changes will have invalid *Index* values until *EndUpdate* is called.

For example, the *Lines* property of a memo component is of type *TStrings*. If your application calls the *AddStrings* method to add several strings at once to the *Lines* property, *AddStrings* calls *BeginUpdate* before the strings are added. After the strings are added, *AddStrings* calls *EndUpdate* and the screen repaints, displaying the new list of strings.

Use *BeginUpdate* to prevent screen repaints and to speed processing time while you are rebuilding your list.

Example

BeginUpdate and *EndUpdate* should always be used in conjunction with a **try...finally** statement to ensure that *EndUpdate* is called if an exception occurs. A block that uses *BeginUpdate* and *EndUpdate* typically looks like this:

```
ListBox1.Items.BeginUpdate;
try
  ListBox1.Items.Clear;
  ListBox1.Items.Add(...);
  ...
  ListBox1.Items.Add(...);
finally
  ListBox1.Items.EndUpdate;                      { Executed even in case of an exception }
end;
```

See also

EndUpdate method

BevelInner property

Applies to

TPanel component

Declaration

```
property BevelInner: TPanelBevel;
```

A panel component has two bevels, an outer bevel drawn next to the border of the control, and an inner bevel drawn inside the outer bevel the number of pixels specified in the *BorderWidth* property.

The *BevelInner* property determines the style of the inner bevel of a panel component. These are the possible values:

Value	Meaning
bvNone	No inner bevel exists.
bvLowered	The inner bevel is lowered.
bvRaised	The inner bevel is raised.

Example

This example uses a panel component and a button named *CreateStatusLine* on a form. The code moves the panel to the bottom of the form when the user clicks the button and gives the panel the appearance of a status line by changing the value of the *BevelInner*, *BevelOuter*, *BevelWidth*, and *BorderWidth* properties.

```
procedure TForm1.CreateStatusLineClick(Sender: TObject);
begin
with Panel1 do
  begin
```

```
      Align := alBottom;
      BevelInner := bvLowered;
      BevelOuter := bvRaised;
      BorderWidth := 1;
      BevelWidth := 1;
    end;
  end;
```

See also
BevelOuter property, *BevelWidth* property, *BorderWidth* property, *TPanelBevel* type

BevelOuter property

Applies to
TPanel component

Declaration

```
property BevelOuter: TPanelBevel;
```

A panel component has two bevels, an outer bevel drawn next to the border of the control, and an inner bevel drawn inside the outer bevel. The width of the inner bevel is specified in the *BorderWidth* property in pixels.

The *BevelOuter* property determines the style of the outer bevel of a panel component. These are the possible values:

Value	Meaning
bvNone	No outer bevel exists.
bvLowered	The outer bevel is lowered.
bvRaised	The outer bevel is raised.

Example
This code creates a lowered frame 10 pixels wide around a panel component named *Panel1*:

```
  Panel1.BorderWidth := 10;
  Panel1.BevelInner := bvRaised;
  Panel1.BevelOuter := bvLowered;
```

See also
BevelInner property, *BevelWidth* property, *BorderWidth* property

BevelWidth property

Applies to
TPanel component

Declaration

```
property BevelWidth: TBevelWidth;
```

The *BevelWidth* property determines the width in pixels between the inner and the outer bevels of a panel. The *BevelInner* property determines how the inner bevel appears, and the *BevelOuter* property determines how the outer bevel appears. By changing these properties, you change the appearance of the panel.

Example

This code alternately displays and hides the bevels of a panel when the user clicks the *Button1* button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with Panel1 do
  begin
    BevelInner := bvLowered;
    BevelOuter := bvRaised;
    if BevelWidth = 0 then
      BevelWidth := 2
    else
      BevelWidth := 0;
  end;
end;
```

See also

BorderWidth property

Bitmap property

Applies to

TBrush, *TPicture* objects

For brushes

Declaration

```
property Bitmap: TBitmap;
```

Run-time only. The *Bitmap* property enables a brush to use a bitmap image for painting with the ability to produce special painting effects such as patterns. The bitmap must be 8 pixels high and 8 pixels wide.

Example

The following code loads a bitmap from a file and assigns it to the *Brush* of the *Canvas* of *Form1*:

```
begin
  Form1.Canvas.Brush.Bitmap.LoadFromFile('MYBITMAP.BMP');
end;
```

For pictures

Declaration

```
property Bitmap: TBitmap;
```

The *Bitmap* property specifies the contents of the *TPicture* object as a bitmap graphic (.BMP file format). If *Bitmap* is referenced when the *TPicture* contains a *Metafile* or *Icon* graphic, the graphic won't be converted. Instead, the original contents of the *TPicture* are discarded and *Bitmap* returns a new, blank bitmap.

Example

The following code copies the bitmap in *Picture1* to the *Glyph* of *BitBtn1*.

```
BitBtn1.Glyph := Picture1.Bitmap;
```

See also

Graphic property

BlockRead procedure **System**

Declaration

```
procedure BlockRead(var F: File; var Buf; Count: Word [; var Result: Word]);
```

The *BlockRead* procedure reads one or more records from an open file into a variable.

F is an untyped file variable, *Buf* is any variable, *Count* is an expression of type *Word*, and *Result* is an optional variable of type *Word*.

BlockRead reads *Count* or fewer records from the file *F* into memory, starting at the first byte occupied by *Buf*. The actual number of complete records read (less than or equal to *Count*) is returned in *Result*.

The entire transferred block occupies at most *Count * RecSize* bytes. *RecSize* is the record size specified when the file was opened (or 128 if the record size was not specified). An error occurs if *Count * RecSize* is greater than 65,535 (64K). You can handle this error using exceptions.

If the entire block was transferred, *Result* is equal to *Count*.

If *Result* is less than *Count*, *ReadBlock* reached the end of the file before the transfer was complete. If the file's record size is greater than 1, *Result* returns the number of complete records read.

If *Result* isn't specified, an I/O error occurs if the number of records read isn't equal to *Count*. You can use the *EInOutError* exception to handle this error.

{**$I+**} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using {**$I–**}, you must use *IOResult* to check for I/O errors.

Example

```
var
  FromF, ToF: file;
  NumRead, NumWritten: Word;
  Buf: array[1..2048] of Char;
begin
  if OpenDialog1.Execute then                               { Display Open dialog box }
  begin
    AssignFile(FromF, OpenDialog1.FileName);
    Reset(FromF, 1);                                              { Record size = 1 }
    if SaveDialog1.Execute then                              { Display Save dialog box}
    begin
      AssignFile(ToF, SaveDialog1.FileName);                     { Open output file }
      Rewrite(ToF, 1);                                            { Record size = 1 }
      Canvas.TextOut(10, 10, 'Copying ' + IntToStr(FileSize(FromF))
        + ' bytes...');
      repeat
        BlockRead(FromF, Buf, SizeOf(Buf), NumRead);
        BlockWrite(ToF, Buf, NumRead, NumWritten);
      until (NumRead = 0) or (NumWritten <> NumRead);
        CloseFile(FromF);
        CloseFile(ToF);
    end;
  end;
end;
```

See also

BlockWrite procedure

BlockWrite procedure — System

Declaration

```
procedure BlockWrite(var f: File; var Buf; Count: Word [; var Result: Word]);
```

The *BlockWrite* procedure writes one or more records from a variable to an open file.

F is an untyped file variable, *Buf* is any variable, *Count* is an expression of type *Word*, and *Result* is an optional variable of type *Word*.

BlockWrite writes *Count* or fewer records to the file *F* from memory, starting at the first byte occupied by *Buf*. The actual number of complete records written (less than or equal to *Count*) is returned in *Result*.

The entire block transferred occupies at most *Count* * *RecSize* bytes. *RecSize* is the record size specified when the file was opened (or 128 if the record size was unspecified). An error occurs if *Count* * *RecSize* is greater than 65,535 (64K). You can use the exception handler *EInOutError* to deal with this error.

If the entire block is transferred, *Result* is equal to *Count* on return.

If *Result* is less than *Count*, the disk became full before the transfer was complete. In this case, if the file's record size is greater than 1, *Result* returns the number of complete records written.

The current file position is advanced by *Result* records as an effect of the *BlockWrite*.

If *Result* isn't specified, an I/O error occurs if the number written isn't equal to *Count*. You can use exception handler *EInOutError* to deal this error.

{**$I+**} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using {**$I–**}, you must use *IOResult* to check for I/O errors.

Example

```
var
  FromF, ToF: file;
  NumRead, NumWritten: Word;
  Buf: array[1..2048] of Char;
begin
  if OpenDialog1.Execute then                              { Display Open dialog box }
  begin
    AssignFile(FromF, OpenDialog1.FileName);
    Reset(FromF, 1);                                                { Record size = 1 }
    if SaveDialog1.Execute then                            { Display Save dialog box }
    begin
      AssignFile(ToF, SaveDialog1.FileName);                      { Open output file }
      Rewrite(ToF, 1);                                              { Record size = 1 }
      Canvas.TextOut(10, 10, 'Copying ' + IntToStr(FileSize(FromF))
        + ' bytes...');
      repeat
        BlockRead(FromF, Buf, SizeOf(Buf), NumRead);
        BlockWrite(ToF, Buf, NumRead, NumWritten);
      until (NumRead = 0) or (NumWritten <> NumRead);
        CloseFile(FromF);
        CloseFile(ToF);
    end;
  end;
end;
```

See also

BlockRead procedure

BOF property

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
property BOF: Boolean;
```

Run-time and read only. *BOF* is a Boolean property that indicates whether a dataset is known to be at its first row. The *BOF* property returns a value of *True* only after:

- An application first opens a table
- A call to a table's *First* method
- A call to a table's *Prior* method fails

Example

```
Table1.Last;
while not Table1.BOF do
begin
  {DoSomething}
  Table1.Prior;
end;
```

See also

MoveBy method

BOLEFormat type **BOLEDefs**

Declaration

```
BOleFormat = Record
  fmtId: Word;
  fmtName: array [0..31] of char;
  fmtResultName: array [0..31] of char;
  fmtMedium: BOleMedium;
  fmtIsLinkable: Bool;
end;
```

BOLEFormat registers a format that allows drag-and-drop of OLE objects and other types onto a form. Pass an array of *BOLEFormat* as a parameter to the *ClearFormOLEDropFormats, RegisterFormAsOLEDropTarget,* and *SetFormOLEDropFormats* procedures.

An array of *BOLEFormat* records is also used when pasting objects from the Clipboard with the *PasteSpecialDlg* function. Each object type you want to be able to paste should be registered as an element of the *Fmts* parameter of *PasteSpecialDlg*. To see if any objects of a given type are on the Clipboard so that the Paste Special dialog box is enabled, pass an array of *BOLEFormats* in the *Fmts* parameter of *PasteSpecialEnabled*.

These are the fields of *BOLEFormat*:

Field	Description
fmtId	Windows Clipboard format ID. For non-OLE data, *fmtId* should be a standard Clipboard format such as *CF_TEXT* for text or *CF_BITMAP* for bitmap graphics. For OLE objects, you should register new Clipboard formats with the Windows API function *RegisterClipboardFormat*.
fmtName	Name to appear in the list box of Paste Special dialog box.
fmtResultName	Name to appear in the Results box of the Paste Special dialog box.
fmtMedium	Based on the Clipboard format ID specified in *fmtId*. For linked OLE objects, *fmtMedium* should be *BOLE_MED_STREAM*. For embedded OLE objects, *fmtMedium* should be *BOLE_MED_STORAGE*.
fmtIsLinkable	*True* if the object is linkable, *False* if not. For linked OLE objects, *fmtIsLinkable* should be set to *True*. For embedded OLE objects, *fmtIsLinkable* should be *False*.

BOLEMedium type — BOLEDefs

Declaration

```
type BoleMedium = Integer;
const
  BOLE_MED_NULL     = 0;
  BOLE_MED_HGLOBAL = 1;  { used for most non-ole2 formats }
  BOLE_MED_FILE     = 2;
  BOLE_MED_STREAM   = 4;  { used for ole2 linked objects }
  BOLE_MED_STORAGE = 8;  { used for ole2 embedded objects }
  BOLE_MED_GDI      = 16; { used for bitmaps and other gdi formats }
  BOLE_MED_MFPICT   = 32; { used for metafile format }
```

BOLEMedium is the type of the *fmtMedium* field of the *BOLEFormat* type. This is based on the *fmtId* field in the same record. For linked OLE objects, the *BOLEMedium* should be *BOLE_MED_STREAM*. For embedded OLE objects, the *BOLEMedium* should be *BOLE_MED_STORAGE*. For other objects, the *BOLEMedium* should be one of the other values, according to the comments in the declaration above. Use *BOLEMediumCalc* to calculate the *BOLEMedium* for a given Clipboard format.

BOLEMediumCalc function — Toctrl

Declaration

```
function BOleMediumCalc(fmtId: Word): BOleMedium;
```

The *BOLEMediumCalc* function returns the *BOLEMedium* value that should be used with the Clipboard format ID passed in the *fmtId* parameter. *BOLEMedium* is the type of the *fmtMedium* field of the *BOLEFormat* record.

Example

The following code calculates the *BOLEMedium* associated with *CF_BITMAP* and stores it in the *fmtMedium* field of the first element of a *BOLEFormat* record array.

```
var
  Fmts: array[0..2] of BOLEFormat;
begin
  Fmts[0].fmtId := CF_BITMAP;
  Fmts[0].fmtMedium := BOLEMediumCalc(CF_BITMAP);
  Fmts[0].fmtIsLinkable := False;
  StrPCopy (Fmts[0].fmtName, '%s');
  StrPCopy (Fmts[0].fmtResultName, '%s');
end;
```

BorderColor property

Applies to

TShape component

Declaration

```
property BorderColor: TColor;
```

The *BorderColor* property is used to color the border of a shape component. For a complete list of the values the *BorderColor* property can have, see the *Color* property.

Example

This example changes the border color of a shape component at run time:

```
Shape1.BorderColor := clBlack;
```

BorderIcons property

Applies to

TForm component

Declaration

```
property BorderIcons: TBorderIcons;
```

The *BorderIcons* property is a set whose values determine which icons appear on the title bar of a form. These are the possible values that the *BorderIcons* set can contain:

Value	Meaning
biSystemMenu	The form has a Control menu (also known as a System menu)
biMinimize	The form has a Minimize button
biMaximize	The form has a Maximize button

Example

The following code removes a form's Maximize button when the user clicks a button:

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  BorderIcons := BorderIcons - [biMaximize];
end;
```

See also

BorderStyle property

BorderStyle property

Applies to

TDBEdit, *TDBGrid*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBLookupList*, *TDBMemo*, *TDrawGrid*, *TEdit*, *TForm*, *THeader*, *TListBox*, *TMaskEdit*, *TMemo*, *TOLEContainer*, *TOutline*, *TPanel*, *TScrollBox*, *TStringGrid* components

For forms

Declaration

```
property BorderStyle: TFormBorderStyle;
```

The *BorderStyle* property for forms specifies both the appearance and the behavior of the form border. You normally set *BorderStyle* at design time, but you can also change it at run time.

BorderStyle can have any of the following values:

Value	Meaning
bsDialog	Not resizeable; standard dialog box border
bsSingle	Not resizeable; single-line border
bsNone	Not resizeable; no visible border line, Minimize or Maximize buttons, or Control menu
bsSizeable	Standard resizeable border

Changing the border style of an MDI child form to *bsDialog* or *bsNone* has no effect.

Example

This example creates a form with a single-line border that the user can't resize:

```
Form1.BorderStyle := bsSingle;
```

See also

BorderIcons property

For controls

Declaration

```
property BorderStyle: TBorderStyle;
```

The *BorderStyle* property of edit boxes, list boxes, memo controls, grid controls, outlines, and scroll boxes determines whether these components have a border. These are the possible values:

Value	Meaning
bsNone	No visible border
bsSingle	Single-line border

If you set the *AutoSize* property of an edit box to *True*, the edit box resizes automatically when the font size of the text changes. You must set the value of the *BorderStyle* property to *fsSingle*, or else *AutoSize* has no effect.

Example

The following example puts a single-line border around the edit box, *Edit1*.

```
Edit1.BorderStyle := bsSingle;
```

See also

Ctl3D property

BorderWidth property

Applies to

TPanel component

Declaration

```
property BorderWidth: TBorderWidth;
```

The *BorderWidth* property determines the width in pixels of the border around a panel. The default value is 0, which means no border.

Example

This example uses a panel component and a button named *CreateStatusLine* on a form. The code moves the panel to the bottom of the form when the user clicks the button, and gives the panel the appearance of a status line by changing the value of the *BevelInner*, *BevelOuter*, *BevelWidth*, and *BorderWidth* properties:

```
procedure TForm1.CreateStatusLineClick(Sender: TObject);
begin
  with Panel1 do
    Align := alBottom;
```

```
      BevelInner := bvLowered;
      BevelOuter := bvRaised;
      BorderWidth := 1;
      BevelWidth := 1;
    end;
  end;
```

See also

BevelInner property, *BevelOuter* property, *BevelWidth* property

Bounds function — Classes

Declaration

```
function Bounds(ALeft, ATop, AWidth, AHeight: Integer): TRect;
```

The *Bounds* function returns a rectangle with the given dimensions. The statement

```
  R := Bounds(X, Y, W, H);
```

corresponds to

```
  R := Rect(X, Y, X + W, Y + H);
```

Example

This example returns a *TRect* record that defines a rectangle that is 100 pixels long on each side with the top left corner at coordinate 10, 10.

```
  var
    R: TRect;
  begin
    R := Bounds(10, 10, 100, 100);
  end;
```

See also

BoundsRect property

BoundsRect property

Applies to

All controls

Declaration

```
property BoundsRect: TRect;
```

The *BoundsRect* property returns the bounding rectangle of the control, expressed in the coordinate system of the parent control. The statement

```
  R := Control.BoundsRect;
```

corresponds to

```
R.Left := Control.Left;
R.Top := Control.Top;
R.Right := Control.Left + Control.Width;
R.Bottom := Control.Top + Control.Height;
```

Example

This code resizes a button control to twice as wide and half as high:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyRect: TRect;
begin
  MyRect := Button2.BoundsRect;
  MyRect.Right := MyRect.Left + 2 * (MyRect.Right - MyRect.Left);
  MyRect.Bottom := MyRect.Top + (MyRect.Bottom - MyRect.Top) div 2;
  Button2.BoundsRect := MyRect;
end;
```

See also

Bounds function

Break procedure — System

Declaration

```
procedure Break;
```

The *Break* procedure causes the flow of control to exit a **for**, **while**, or **repeat** statement and continue at the next statement following the loop statement.

The compiler reports an error if a call to *Break* isn't in a **for**, **while**, or **repeat** statement.

Example

```
uses WinCRT;

var
  S: string;
begin
  while True do
  begin
    ReadLn(S);
    if S = '' then Break;
    WriteLn(S);
  end;
end;
```

See also

Continue procedure, *Exit* procedure, *Halt* procedure

Break property

Applies to

TMenuItem component

Declaration

```
property Break: TMenuBreak;
```

The *Break* property lets you break a long menu into columns. These are the possible values:

Value	Meaning
mbNone	No menu breaking occurs.
mbBarBreak	The menu breaks into another column with the menu item appearing at the top of the new column. A bar separates the new and the old columns.
mbBreak	The menu breaks into another column with the menu item appearing at the top of the new column. Only space separates the new and the old columns.

The default value is *mbNone*.

Example

This example uses a button and a main menu component with several subitems on it, including one labeled Save As, so that Delphi automatically names that menu item *SaveAs1*. When the user clicks the button on the form, the menu breaks so Save As appears in a second column with a bar between the two columns. The change to the menu is visible when the menu displays.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  SaveAs1.Break := mbBarBreak;
end;
```

See also

Checked property, *Enabled* property

BringToFront method

Applies to

All controls; *TForm* component

Declaration

```
procedure BringToFront;
```

The *BringToFront* method puts the component or form in front of all other components or forms within its parent component or form. *BringToFront* is especially useful for making certain that a form is visible. You can also use it to reorder overlapping components within a form.

The order in which controls stack on top of each order (also called the *Z order*) depends on whether the controls are windowed or non-windowed controls. For example, if you put a label and an image on a form so that one is on top of the other, the one you placed first on the form is the one on the bottom. Because both the label and the image are non-windowed controls, they "stack" as you would expect them to. Suppose that the label is on the bottom. If you call the *BringToFront* method for the label, the label then appears on top of the image.

The stacking order of windowed controls is the same. For example, if you put a memo on a form, then put a check box on top of it, the check box remains on top. If you call *BringToFront* for the memo, the memo appears on top.

The stacking order of windowed and non-windowed controls cannot be mingled. For example, if you put a memo, a windowed control, on a form, and then put a label, a non-windowed control, on top of it, the label disappears behind the memo. Windowed controls always stack on top of non-windowed controls. In this example, if you call the *BringToFront* method of the label, it remains behind the memo.

Example

The following code uses two forms. *Form1* has a button on it. The second form is used as a tool palette. This code makes the palette form visible, and ensures it is the top form by bringing it to the front.

To run this example, you must put *Unit2* in the **uses** clause of your unit.

```
procedure TForm1.ShowPaletteButtonClick(Sender: TObject);
begin
  if Form2.Visible = False then Form2.Visible := True;
  Form2.BringToFront;
end;
```

See also

SendToBack method

Brush property

Applies to

All controls; *TCanvas* object; *TForm*, *TShape* components

Declaration

```
property Brush: TBrush;
```

A canvas or shape object's *Brush* property determines what kind of color and pattern the canvas uses for filling graphical shapes and backgrounds. Controls also specify an additional brush in their *Brush* properties, which they use for painting their backgrounds.

For controls, *Brush* is a read only and run-time only property.

Example

The following code sets the color of the brush used by *Form1* to fill shapes drawn on it with red:

```
procedure TForm1.MakeRedButtonClick(Sender: TObject);
begin
  Canvas.Brush.Color := clRed;
end;
```

This code changes the shape, color, and pattern of a shape component:

```
procedure TForm1.ChangeShapeClick(Sender: TObject);
begin
  Shape1.Shape := stEllipse;
  Shape1.Brush.Color := clMaroon;
  Shape1.Brush.Style := bsFDiagonal;
end;
```

See also

BrushCopy method, *Canvas* property, *Font* property, *Pen* property, *TCanvas* object

BrushCopy method

Applies to

TCanvas object

Declaration

```
procedure BrushCopy(const Dest: TRect; Bitmap: TBitmap; const Source: TRect; Color: TColor);
```

The *BrushCopy* method copies a portion of a bitmap onto a portion of a canvas, replacing one of the colors of the bitmap with the brush of the destination canvas. *Dest* specifies the rectangular portion of the destination canvas to copy to. *Bitmap* specifies the graphic to copy from. Source specifies the rectangular area of the bitmap to copy. *Color* specifies the color in *Bitmap* to replace with the brush of the canvas (specified in the *Brush* property).

You could use *BrushCopy* to make the copied image partially transparent, for example. To do this, you would specify the color of the surface being copied to (*clBackground* for example) as the *Color* of the *Brush* property of the destination canvas, then call *BrushCopy*.

Example

The following code illustrates the differences between *CopyRect* and *BrushCopy*. The bitmap graphic 'TARTAN.BMP' is loaded into *Bitmap* and displayed on the *Canvas* of Form1. *BrushCopy* replaces the color black in the graphic with the brush of the canvas, while *CopyRect* leaves the colors intact.

```
var
  Bitmap: TBitmap;
  MyRect, MyOther: TRect;
begin
  MyRect.Top := 10;
  MyRect.Left := 10;
  MyRect.Bottom := 100;
  MyRect.Right := 100;
  MyOther.Top := 111; {110}
  MyOther.Left := 10;
  MyOther.Bottom := 201; {210}
  MyOther.Right := 100;
  Bitmap := TBitmap.Create;
  Bitmap.LoadFromFile('c:\windows\tartan.bmp');
  Form1.Canvas.BrushCopy(MyRect,Bitmap, MyRect, clBlack);
  Form1.Canvas.CopyRect(MyOther,Bitmap.Canvas,MyRect);
  Bitmap.Free;
end;
```

See also
Brush property, *CopyRect* method

BtnClick method

Applies to
TDBNavigator component

Declaration

```
procedure BtnClick(Index: TNavigateBtn);
```

The *BtnClick* method simulates a button click on the database navigator, invoking the action of the button. Specify which button *BtnClick* should operate on as the value of the *Index* parameter.

Example
This line of code simulates the clicking of the Next button on a database navigator control, which makes the next record in the dataset the current record:

```
DBNavigator1.BtnClick(nbNext);
```

Buttons property

Applies to
TDBRadioGroup, *TRadioGroup* components

Declaration

```
property Buttons: TList;
```

Run-time and read only. The *Buttons* property lets your application access the list of radio buttons in the database radio button group box. Use the properties and methods of a list object (*TList*) to manipulate the list of buttons.

Example

The following code disables the first button in *DBRadioGroup1*.

```
   TRadioButton(DBRadioGroup1.Buttons.First).Enabled := False;
```

Calculated property

Applies to

TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
property Calculated: Boolean;
```

Calculated is *True* if the value of the field is calculated by the *OnCalcFields* event handler. Calculated fields can be created with the Fields Editor, but are not stored in or retrieved from the physical tables underlying a dataset. Instead they are calculated for each record in the table by the dataset's *OnCalcFields* event handler, which typically uses expressions involving values from other fields in the record to generate a value for each calculated field. For example, a table might have non-calculated fields for Quantity and UnitPrice, and a calculated field for ExtendedPrice, which would be calculated by multiplying the values of the Quantity and UnitPrice fields. Calculated fields are also useful for performing lookups in other tables. For example, a part number can be used to retrieve a part description for display in an invoice line item.

Cancel method

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure Cancel;
```

The *Cancel* method returns the dataset to Browse state and discards any changes to the current record.

See also

Append method, *Insert* method, *Post* method

Cancel property

Applies to

TBitBtn, *TButton* components

Declaration

```
property Cancel: Boolean;
```

The *Cancel* property indicates whether a button or a bitmap button is a Cancel button. If *Cancel* is *True*, any time the user presses Esc, the *OnClick* event handler for the button executes. Although your application can have more than one button designated as a Cancel button, the form calls the *OnClick* event handler only for the first button in the tab order that is visible.

Example

The following code designates a button called *Button1* as a Cancel button:

```
Button1.Cancel := True;
```

See also

Default property, *OnClick* event

CancelRange method

Applies to

TTable component

Declaration

```
procedure CancelRange;
```

The *CancelRange* method removes any range limitations for the *TTable* which were previously established by calling the *ApplyRange* or *SetRange* methods.

Example

```
Table1.CancelRange;
```

CanFocus method

Applies to

All controls

Declaration

```
function CanFocus: Boolean;
```

The *CanFocus* method determines whether a control can receive focus. *CanFocus* returns *True* if both the control and its parent(s) have their *Visible* and *Enabled* properties set to *True*. If all the *Visible* and *Enabled* properties of the control and the components from which the control descends are **not** *True*, then *CanFocus* returns *False*.

Example

This example uses a group box, a label, and a button on a form. The group box contains a check box. When the application runs, the group box is disabled (*Enabled* is set to *False*). Because the group box is the parent of the check box, the user can never tab to the check box. When the user clicks the button, the caption of the label reports that the check box can not receive the input focus:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  GroupBox1.Enabled := True;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if CheckBox1.CanFocus then
    Label1.Caption := 'The check box can focus'
  else
    Label1.Caption := 'The check box cannot focus';
end;
```

See also

Parent property

CanModify property

Applies to

TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStoredProc*, *TStringField*, *TQuery*, *TTable*, *TTimeField*, *TVarBytesField*, *TWordField* components

For tables, queries, and stored procedures

Declaration

```
property CanModify: Boolean;
```

Run-time and read only. *CanModify* specifies whether an application can modify the data in a dataset. When *CanModify* is *False*, then the dataset is read-only, and cannot be put into *Edit* or *Insert* state. When *CanModify* is *True*, the dataset can enter *Edit* or *Insert* state.

Even if *CanModify* is True, it is not a guarantee that a user will be able to insert or update records in a table. Other factors may come in to play, for example, SQL access privileges.

TTable has a *ReadOnly* property that requests write privileges when set to *False*. When *ReadOnly* is *True*, *CanModify* will automatically be set to *False*. When *ReadOnly* is *False*, *CanModify* will be *True* if the database allows read and write privileges for the dataset and the underlying table.

Example

```
if Table1.CanModify then
{ Do this only if the dataset can be modified }
  Table1.CustNo := 1234;
```

See also
Active property

For field components

Declaration

```
property CanModify: Boolean;
```

Run-time and read only. Specifies if a field can be modified for any reason, such as during a *SetKey* operation. *CanModify* is *True* if the value of the field can be modified. If the *ReadOnly* property of the field is *True*, or the *ReadOnly* property of the dataset is *True*, then *CanModify* is *False*.

See also
DataSet *property*

Canvas property

Applies to
TBitmap, *TComboBox*, *TDBComboBox*, *TDBGrid*, *TDBListBox*, *TDirectoryListBox*, *TDrawGrid*, *TFileListBox*, *TForm*, *TImage*, *TListBox*, *TOutline*, *TPaintBox*, *TPrinter*, *TStringGrid* components

For forms, images, and paint boxes

Declaration

```
property Canvas: TCanvas;
```

Run-time only. The *Canvas* property gives you access to a drawing surface that you can use when implementing a handler for the *OnPaint* event of a form, an image, or a paint box.

The *Canvas* property of an image or a form is read only.

Example

The following code sets the *Color* of the *Pen* of the *Canvas* of *Bitmap1* to *clBlue*.

```
Bitmap1.Canvas.Pen.Color := clBlue;
```

See also

Search for Graphics in online Help and choose the topic Drawing Graphics at Run Time

For list boxes, combo boxes, and outlines

Declaration

```
property Canvas: TCanvas;
```

Run-time and read only. The *Canvas* property gives you access to a drawing surface that you can use when implementing a handler for the *OnDrawItem* event of an owner-draw list box, combo box, or outline control.

Example

The following code draws a graphic stored in the *Objects* property of the *Items* list of *ListBox1*. This code should be attached to the *OnDrawItem* event handler of *ListBox1*, and the *Style* property of *ListBox1* should be *lbOwnerDrawFixed*.

```
procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
var
  SourceRect: TRect;
begin
  SourceRect.Top := 0;
  SourceRect.Left := 0;
  SourceRect.Bottom := TBitmap(ListBox1.Items.Objects[Index]).Height;
  SourceRect.Right := TBitmap(ListBox1.Items.Objects[Index]).Width;
  ListBox1.Canvas.CopyRect(Rect, TBitmap(ListBox1.Items.Objects[Index]).Canvas,
    SourceRect);
end;
```

The following code draws a graphic stored in the *Data* property of the *Items* list of *Outline1*. This code should be attached to the *OnDrawItem* event handler of *Outline1*, and the *Style* property of *Outline1* should be *otOwnerDraw*.

```
procedure TForm1.Outline1DrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
var
  SourceRect: TRect;
begin
  SourceRect.Top := 0;
```

```
    SourceRect.Left := 0;
    SourceRect.Bottom := TBitmap(Outline1.Items[Index].Data).Height;
    SourceRect.Right := TBitmap(Outline1.Items[Index].Data).Width;
    Outline1.Canvas.CopyRect(Rect, TBitmap(Outline1.Items[Index].Data).Canvas,
      SourceRect);
  end;
```

See also

ItemHeight property, *OnDrawCell* event, *OnDrawDataCell* event, *OnDrawItem* event, *OnPaint* event

For grids

Declaration

```
property Canvas: TCanvas;
```

Run-time and read only. The *Canvas* property gives you access to a drawing surface that you can use when implementing a handler for the *OnDrawCell* or *OnDrawDataCell* event of a grid control.

For printer objects

Declaration

```
property Canvas: TCanvas;
```

Run-time only and read only. The *Canvas* property for a printer object represents the surface of the currently printing page.

Note Some printers do not support graphics. Therefore, the *Draw*, *StretchDraw*, or *CopyRect* methods might fail on these printers.

Example

The following code prints the text ‘Hello, world!’:

```
Printer.BeginDoc;
Printer.Canvas.TextOut(0, 0, 'Hello, world');
Pritner.EndDoc;
```

See also

Brush property, *Font* property, *Pen* property, *TextOut* method

For bitmap objects

Applies to

TBitmap object

Declaration

```
property Canvas: TCanvas;
```

Run-time and read only. The *Canvas* property gives you access to a drawing surface that represents the bitmap. When you draw on the canvas you are in effect modifying the underlying bitmap.

See also
Draw method

Capabilities property

Applies to
TMediaPlayer component

Declaration

```
property Capabilities: TMPDevCapsSet;
```

Run-time and read only. The *Capabilities* property determines the capabilities of the open multimedia device.

The various capabilities specified in *Capabilities* are determined when the device is opened with the *Open* method. The following table lists the capabilities a device can have:

Value	Capability
mpCanEject	Can eject media
mpCanPlay	Can play media
mpCanRecord	Can record media
mpCanStep	Can step forward or backward within media
mpUsesWindows	Uses a window for displaying output

Note Currently, there is no way to check whether a device can step forward or backward. *Capabilities* includes *mpCanStep* only if the device type (specified in the *DeviceType* property) is *Animation*, *AVI Video*, *Digital Video*, *Overlay*, or *VCR*.

Example
The following code determines whether the device opened by the media player control *MediaPlayer1* uses a window to display output. If so, the output displays in a form named *Form2*:

```
if mpUsesWindows in MediaPlayer1.Capabilities then
  MediaPlayer1.Display := Form2;
```

See also

Back method, *Display* property, *Eject* method, *Play* method, *StartRecording* method, *Step* method

Capacity property

Applies to

TList object

Declaration

```
property Capacity: Integer;
```

Run time only. The *Capacity* property contains the allocated size of the array of pointers maintained by a *TList* object. This is different from the *Count* property, which contains the number of entries that are actually in use. The value of the *Capacity* property is always greater than or equal to the value of the *Count* property.

When setting the *Capacity* property, an *EListError* exception occurs if the specified value is less than the *Count* property or greater than 16380 (the maximum number of elements a list object can contain). Also, an *EOutOfMemory* exception occurs if there is not enough memory to expand the list to its new size.

When an element is added to a list whose *Capacity* and *Count* are equal (indicating that all allocated entries are in use), the *Capacity* is automatically increased by 16 elements. In situations where you are going to be adding a known number of elements to a list, you can reduce memory reallocations by first increasing the list's capacity. For example,

```
List.Clear;
List.Capacity := Count;
for I := 1 to Count do List.Add(...);
```

The assignment to *Capacity* before the for loop ensures that each of the following *Add* operations doesn’t cause the list to be reallocated, which in turn means that the *Add* operations are guaranteed to never raise an exception.

Example

The following code sets the *Capacity* of *List1* to 5.

```
List1.Capacity := 5;
```

See also

Count property, *Expand* method, *Items* property, *Pack* method

Caption property

Applies to

TBitBtn, *TButton*, *TCheckBox*, *TDBCheckBox*, *TDBRadioGroup*, *TForm*, *TGroupBox*, *TLabel*, *TMenuItem*, *TPanel*, *TRadioButton*, *TSpeedButton* components

The *Caption* property specifies text that will appear in a component.

For forms

Declaration

```
property Caption: string;
```

The *Caption* property is the text that appears in the form's title bar; this text also appears as the icon label when the form is minimized.

Example

The following code creates a caption that says "Hello, World!" on a form called *MyForm*:

```
   MyForm.Caption := 'Hello, World!';
```

See also

BorderStyle property

For all other components

Declaration

```
property Caption: string;
```

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

For menu items, you can use the *Caption* property to include a line that separates the menu into parts. Specify a hyphen character (**-**) as the value of *Caption* for the menu item.

The *Caption* property of a data grid is available at run time only.

Example

This code changes the caption of a group box:

```
   procedure TForm1.Button1Click(Sender: TObject);
   begin
     GroupBox1.Caption := 'Fancy options';
   end;
```

See also

FocusControl property, *ShowAccelChar* property, *Text* property

Cascade method

Applies to

TForm component

Declaration

```
procedure Cascade;
```

The *Cascade* method rearranges the child forms in your application so they overlap. The top of each form remains visible so that you can easily select one of the forms. The *Cascade* method applies only to MDI parent forms (with a *FormStyle* property value of *fsMDIForm*).

Example

This code arranges all MDI children of the current MDI parent form in a cascade pattern when the user chooses the Cascade menu command:

```
procedure TForm1.Cascade1Click(Sender: TObject);
begin
  Cascade;
end;
```

See also

ArrangeIcons method, *Next* method, *Previous* method, *Tile* method

CellRect method

Applies to

TDrawGrid, *TStringGrid* components

Declaration

```
function CellRect(ACol, ARow: Longint): TRect;
```

The *CellRect* method creates a rectangle of type *TRect* for the cell defined by the column *ACol* and the row *ARow*. If the cell indicated by *ACol* and *ARow* is not visible, *CellRect* returns an empty rectangle.

Example

This example uses a string grid, four labels, and a button on a form. When the user clicks the button, the coordinates of the cell in the second column and first row appear in the label captions:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Rectangle: TRect;
begin
  Rectangle := StringGrid1.CellRect(3, 2);
```

```
    Label1.Caption := IntToStr(Rectangle.Top) + ' is the top';
    Label2.Caption := IntToStr(Rectangle.Bottom) + ' is the bottom';
    Label3.Caption := IntToStr(Rectangle.Left) + ' is the left side';
    Label4.Caption := IntToStr(Rectangle.Right) + ' is the right side';
  end;
```

See also

MouseToCell method

Cells property

Applies to

TStringGrid component

Declaration

```
property Cells[ACol, ARow: Integer]: string;
```

Run-time only. The *Cells* property is an array of strings, one string for each cell in the grid. Use the *Cells* property to access a string within a particular cell. *ACol* is the column coordinate of the cell, and *ARow* is the row coordinate of the cell. The first row is row zero, and the first column is column zero.

The *ColCount* and *RowCount* property values define the size of the array of strings.

Example

This code fills each cell of a grid with the same string.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I, J: Integer;
begin
  with StringGrid1 do
    for I := 0 to ColCount - 1 do
      for J:= 0 to RowCount - 1 do
        Cells[I,J] := 'Delphi';
end;
```

See also

Cols property, *Objects* property, *Rows* property

Center property

Applies to

TDBImage, *TImage* components

Declaration

```
property Center: Boolean;
```

The *Center* property determines whether an image is centered in the image control. If *Center* is *True*, the image is centered. If *Center* is *False*, the image aligns with the top left corner of the control. The default value is *True*.

Example

The following code centers the image in *Image1* when the user checks *CheckBox1*:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  Image1.Center := CheckBox1.Checked;
end;
```

See also

AutoSize property, *Stretch* property

ChangedCount property

Applies to

TBatchMove component

Declaration

```
property ChangedCount: Longint;
```

Run-time and read only. *ChangedCount* is the number of records added to the table specified by *ChangedTableName*. If *ChangedTableName* is not specified, the count is still valid.

Example

```
with BatchMove1 do
  begin
  Execute;
  if ChangedCount <> Source.RecordCount then { something went wrong };
  end;
```

See also

ChangedTableName property

ChangedTableName property

Applies to

TBatchMove component

Declaration

```
property ChangedTableName: TFileName;
```

ChangedTableName, if specified, creates a local (Paradox) table containing all records in the destination table that changed as a result of the batch operation. The number of records placed in the new table is reported in the *ChangedCount* property.

Example

```
  BatchMove1.ChangedTableName := 'oldrecs.db';
```

ChangeFileExt function **SysUtils**

Declaration

```
function ChangeFileExt(const FileName, Extension: string): string;
```

The *ChangeFileExt* function takes the file name passed in *FileName* and changes the extension of the file name to the extension passed in *Extension*.

Example

The following code generates the name of an .INI file based on the name of the program:

```
  function INIFileName: string;
  begin
    Result := ChangeFileExt(ParamStr(0), '.INI');
  end;
```

ChangeLevelBy method

Applies to

TOutlineNode object

Declaration

```
procedure ChangeLevelBy(Value: TChangeRange);
```

The *ChangeLevelBy* method changes the level of an outline item. Specify a *Value* parameter value of -1 to move up (toward the root) one level. Specify a *Value* parameter value of 1 to move down (away from the root) one level.

When moving up one level, an item becomes the next sibling of its former parent. When moving down one level, an item becomes the last child of its former prior sibling. Therefore, you can not change the level of the first item in the outline, as it has no parent or prior sibling. Also, you can not move items that are already on the first level up one level.

ChangeLevelBy modifies the value of the *Level* property to reflect the new level. You can only move an item up or down one level at a time.

Example
Attach the following code to the *OnClick* event handlers of two buttons to allow the user to move the selected outline item up or down. The code for *UpBtn* checks to see if the selected item is not already on the first level before moving it up. The code for *DownBtn* checks to see if the selected item has a prior sibling before moving it down.

```
procedure TForm1.UpBtnClick(Sender: TObject);
begin
  with Outline1[Outline1.SelectedItem] do
    if Level > 1 then ChangeLevelBy(-1);
end;

procedure TForm1.DownBtnClick(Sender: TObject);
begin
  with Outline1[Outline1.SelectedItem] do
    if Outline1[Parent.GetPrevChild(Index)] <> -1 then
      ChangeLevelBy(1);
end;
```

See also
Level property, *MoveTo* method

CharCase property

Applies to
TDBEdit, *TEdit*, *TMaskEdit* components

Declaration
```
property CharCase: TEditCharCase;
```

The *CharCase* property determines the case of the *Text* property of the edit box. These are the possible values:

Value	Meaning
ecLowerCase	The text of the edit box displays in lowercase
ecNormal	The text of the edit box displays in mixed case
ecUpperCase	The text of the edit box displays in uppercase

If the user tries to enter a different case than the current value of *CharCase*, the characters the user enters appear in the case specified by *CharCase*. For example, if the value of *CharCase* is *ecLowerCase*, only lowercase characters appear in the edit box, even if the user tries to enter uppercase characters.

Example
This example uses an edit box and group box containing three radio buttons. When the user selects the first radio button, the text in the edit box becomes lowercase, and any text the user types in the edit box also appears in lowercase. When the user selects the

second radio button, the text in the edit box becomes uppercase, and any text the user types in the edit box also appears in uppercase. When the user selects the third radio button, the text in the edit box remains unchanged, but the user can type using either upper- or lowercase characters:

```
procedure TForm1.RadioButton1Click(Sender: TObject);
begin
  Edit1.CharCase := ecLowerCase;
end;

procedure TForm1.RadioButton2Click(Sender: TObject);
begin
  Edit1.CharCase := ecUpperCase;
end;

procedure TForm1.RadioButton3Click(Sender: TObject);
begin
  Edit1.CharCase := ecNormal;
end;
```

ChDir procedure **System**

Declaration

```
procedure ChDir(S: string);
```

The *ChDir* procedure changes the current directory to the path specified by *S*. If *S* specifies a drive letter, the current drive is also changed.

{**$I+**} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using {**$I–**}, you must use *IOResult* to check for I/O errors. *IOResult* returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

Example

```
begin
  {$I-}
  { Change to directory specified in Edit1 }
  ChDir(Edit1.Text);
  if IOResult <> 0 then
    MessageDlg('Cannot find directory', mtWarning, [mbOk], 0);
end;
```

See also

GetDir procedure, *MkDir* procedure, *RmDir* procedure

Check procedure DB

Declaration

```
procedure Check(Status: Integer);
```

The *Check* procedure tests *Status* for a nonzero value and calls *DbiError* passing *Status*.

CheckBreak typed constant WinCrt

Declaration

```
const CheckBreak: Boolean = True;
```

The *CheckBreak* typed constant controls user termination of an application using the CRT window.

When *CheckBreak* is *True*, the user can terminate the application at any time by

- Choosing the *Close* command on the CRT window’s Control menu
- Double-clicking the window's Control-menu box
- Pressing *Alt+F4*
- Pressing *Ctrl+Break*

The user can also press *Ctrl+C* or *Ctrl+Break* at any time to halt the application and force the CRT window into its inactive state.

All these features are disabled when *CheckBreak* is *False*.

At run time, *Crt* stores the old *Ctrl+Break* interrupt vector, $1B, in a global pointer called *SaveInt1B*.

CheckBrowseMode method

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure CheckBrowseMode;
```

The *CheckBrowseMode* method verifies that the *dataset* is open and has no pending changes. If the dataset’s *State* property is *dsEdit*, *dsInsert* or *dsSetKey*, the *Post* method is called to post any pending changes. If the dataset is closed, an *EDataBaseError* exception will be raised.

Checked property

Applies to
TCheckBox, *TDBCheckBox*, *TMenuItem*, *TRadioButton* components

Declaration

```
property Checked: Boolean;
```

Run-time only. The *Checked* property determines whether an option is selected. These are the possible values:

Component	Value	Meaning
Check box	*True*	A check mark appears in the check box, indicating the option is selected.
	False	No check mark appears, indicating the option is not selected. The value of the *Checked* property is *False* if the *State* of the check box is *cbGrayed* (the check box is grayed) or *cbUnChecked* (the check box is unchecked).
Radio button	*True*	A black circle appears in the radio button, indicating that the option is selected.
	False	No black circle appears in the radio button, indicating the option is not selected.
Menu item	*True*	A check mark appears next to the menu item in the menu, indicating the item is selected.
	False	No check mark appears, indicating the item is not selected.

Example
This example fills in a radio button at run time:

```
  RadioButton1.Checked := True;
```

This example uses a main menu component that contains a menu item named *SnapToGrid1* on a form. When the user chooses the Snap To Grid command, a check mark appears next to the command. When the user chooses the Snap To Grid command again, the check marks disappears:

```
  procedure TForm1.SnapToGrid1Click(Sender: TObject);
  begin
    SnapToGrid1.Checked := not SnapToGrid1.Checked;
  end;
```

See also
AllowGrayed property, *State* property

CheckEOF typed constant WinCrt

Declaration

```
const CheckEOF: Boolean = False;
```

The *CheckEOF* typed constant controls the end-of-file character checking in the CRT window.

When *CheckEOF* is *True*, an end-of-file marker is generated when the user presses *Ctrl+Z* while reading from a file assigned to the CRT window.

When *CheckEOF* is *False*, pressing *Ctrl+Z* has no effect.

CheckEOF is *False* by default.

Chord method

Applies to
TCanvas object

Declaration

```
procedure Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);
```

The *Chord* method draws a line on the canvas connecting two points on the ellipse bounded by the specified rectangle. The screen pixel coordinates (*X1*, *Y1*) and (*X2*, *Y2*) define the enclosing rectangle for the chord. (*X3*,*Y3*) is the starting point for the line, and (*X4*, *Y4*) is the ending point.

Example
This code draws a chord on the top of an ellipse bounded by the current window:

```
var
  R: TRect;
begin
  R := GetClientRect;     {Gets the rectangular coordinates of the current window}
  Canvas.Chord(R.Left, R.Top, R.Right, R.Bottom, R.Right, R.Top, R.Left, R.Top);
end;
```

See also
Arc method, *Draw* method, *Ellipse* method, *Pie* method

Chr function — System

Declaration

```
function Chr(X: Byte): Char;
```

The *Chr* function returns the character with the ordinal value (ASCII value) of the byte-type expression, *X*.

Example

```
begin
  Canvas.TextOut(10, 10, Chr(65));  { The letter ‘A’}
```

```
end;
```

See also

Ord function

ClassName method

Applies to

All objects and components

Declaration

```
class function ClassName: string;
```

The *ClassName* function returns the name of an object or a class. For example, *TButton.ClassName* returns the string 'TButton'.

The name returned by *ClassName* is the name of the actual class of the object, as opposed to the object's declared class. For example, the following code assigns 'TButton' to *S*, not 'TObject':

```
var
  MyObject: TObject;
  S: string;
begin
  MyObject := TButton.Create(Application);
  S := MyObject.ClassName;
  ...
  MyObject.Free;
end;
```

Example

This example uses a button, a label, a list box, a check box, and an edit box on a form. When the user clicks one of the controls, the name of the control's class appears in the edit box.

```
procedure FindClassName(AControl:TObject);
begin
  Form1.Edit1.Text := AControl.ClassName;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  FindClassName(Button1);
end;

procedure TForm1.Label1Click(Sender: TObject);
begin
  FindClassName(Label1);
end;

procedure TForm1.CheckBox1Click(Sender: TObject);
```

```
begin
  FindClassName(CheckBox1);
end;

procedure TForm1.ListBox1Click(Sender: TObject);
begin
  FindClassName(ListBox1);
end;
```

See also

ClassParent method, *ClassType* method

ClassParent method

Applies to

All objects and components

Declaration

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

The *ClassParent* method returns the parent class of an object or a class. The returned value is the immediate ancestor of the object or class. For example,*TScrollBar.ClassParent* returns *TWinControl* as *TScrollBar* is derived from *TWinControl*.

Note that *TObject.ClassParent* returns **nil** because *TObject* has no parent.

Example

This code example uses a button and a list box on a form. When the user clicks the button, the name of the button's class and the names of its parent classes are added to the list box.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ClassRef: TClass;
begin
  ListBox1.Clear;
  ClassRef := Sender.ClassType;
  while ClassRef <> nil do
  begin
    ListBox1.Items.Add(ClassRef.ClassName);
    ClassRef := ClassRef.ClassParent;
  end;
end;
```

The list box contains the following strings after clicking the button:

TButton
TButtonControl
TWinControl
TControl

TComponent
TPersistent
TObject

See also

ClassName method, *ClassType* method

ClassType method

Applies to

All objects and components

Declaration

```
function ClassType: TClass;
```

The *ClassType* function returns the class of an object.

Example

This example uses a button and a label on a form. When the user clicks the button, the type of the button component (*TButton*) appears in the caption of the label.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ButtonClassType: TClass;
begin
  ButtonClassType := Button1.ClassType;
  Label1.Caption := ButtonClassType.ClassName;
end;
```

See also

ClassName method, *ClassParent* method

Clear method

Applies to

TClipboard, *TFieldDefs*, *TIndexDefs*, *TList*, *TParam*, *TParams*, *TStringList*, *TStrings* objects; *TBCDField*, *TBlobField*, *TBooleanField*, *TBytesField*, *TComboBox*, *TDBComboBox*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TDBEdit*, *TDBListBox*, *TDBMemo*, *TDirectoryListBox*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TFilterComboBox*, *TFloatField*, *TGraphicField*, *TIndexDefs*, *TIntegerField*, *TListBox*, *TMaskEdit*, *TMemo*, *TMemoField*, *TOutline*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

For TParams objects

Declaration

procedure Clear;

The *Clear* method deletes all parameter information from *Items*.

Example

```
Params1.Clear;
```

For TParam objects

Declaration

procedure Clear;

The *Clear* method sets the parameter to NULL, erasing all previously assigned data. The *Name*, *DataType* and *ParamType* properties are not altered.

Example

```
{ Clear the CustNo parameter for Query 1 }
Query1.ParamByName('CustNo').Clear;
```

For TIndexDefs objects

Declaration

procedure Clear;

The *Clear* method frees all of the entries in the *Items* property.

For TFieldsDefs objects

Declaration

procedure Clear;

The *Clear* method frees all of the entries in the *Items* property, effectively removing all *TFieldDef* objects from *TFieldDefs*.

For fields

Declaration

procedure Clear;

Clear sets the value of the field to NULL.

For other objects and components

Declaration

```
procedure Clear;
```

The *Clear* method deletes all text from the control, or, in the case of list and string objects or outlines, deletes all items. For the Clipboard object, *Clear* deletes the contents of the Clipboard; this happens automatically each time data is added to the Clipboard (cut and copy operations).

Example

The following code removes the text from an edit box control called *NameField*:

```
NameField.Clear;
```

This example uses a list box and a button on a form. When the form is created, strings are added to the list box. When the user clicks the button, all the strings contained in the *Items* property, a *TStrings* object, are cleared.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ListBox1.Items.Add('One');
  ListBox1.Items.Add('Two');
  ListBox1.Items.Add('Three');
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Clear;
end;
```

See also

CopyToClipboard method, *CutToClipboard* method, *Items* property, *Pack* method, *PasteFromClipboard* method, *Text* property, *Strings* property

ClearFields method

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure ClearFields;
```

The *ClearFields* method clears all fields of the current record to their default values (normally NULL.) The dataset must be in Edit state or an *EDatabaseError* exception will be raised.

See also
Edit method, *State* property, *TField* component

ClearFormOLEDropFormats procedure **Toctrl**

Declaration

```
procedure ClearFormOleDropFormats(Form: TForm);
```

ClearFormOLEDropFormats deletes the object formats that can be dropped on a form that is registered for drag-and-drop by the *RegisterFormAsOLEDropTarget* procedure. If the form is cleared of OLE drag-and-drop formats, no OLE objects can be dropped into a *TOLEContainer* component.

Example
The following code clears *Form1* of object formats:

```
ClearFormOLEDropFormats(Form1);
```

See also
SetFormOLEDropFormats procedure, *TOLEDropNotify* object

ClearSelection method

Applies to
TDBEdit, *TDBMemo*, *TEdit*, *TMaskEdit*, *TMemo* components

Declaration

```
procedure ClearSelection;
```

The *ClearSelection* method deletes text selected in an edit box or memo control. If no text is selected in the control when *ClearSelection* is called, nothing happens.

Example
This code uses a memo control named *MyMemo* and a button on a form. When the user clicks the button, the text the user selected in the memo control is deleted.

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

See also
Clear method, *CopyToClipboard* method, *CutToClipboard* method, *PasteFromClipboard* method

Click method

Applies to
TBitBtn, *TButton*, *TDBNavigator*, *TMenuItem*, *TSpeedButton* components

For menu items and buttons

Declaration

```
procedure Click;
```

The *Click* method simulates a mouse click, as if the user had clicked a menu item or button, executing any code attached to the *OnClick* event.

Example

This example uses a main menu component and a button named *Print*. The main menu component has a *Print1* menu item on it. When the user clicks the button, the code attached to the *OnClick* event of the *Print1* menu item runs.

```
procedure TForm1.PrintClick(Sender: TObject);
begin
  Print1.Click;
end;
```

See also

OnClick event

For database navigator controls

Declaration

```
procedure Click(Button: TNavigateBtn);
```

The *Click* method simulates a mouse click, as if the user had clicked a button on the database navigator, executing any code attached to the *OnClick* event. Specify which button the *Click* method applies to using the *Button* parameter.

Example

The following code simulates a click on the *Next* button of *DBNavigator1*.

```
DBNavigator1.Click(nbNext);
```

ClientHandle property

Applies to
TForm component

Declaration

```
property ClientHandle: HWND;
```

Read only. The *ClientHandle* property value is the handle to the internal MDI (Multiple Document Interface) client window. The property value is meaningful only if the form is an MDI parent form with its *FormStyle* property set to *fsMDIForm*.

ClientHeight property

Applies to

All controls; *TForm* component

Declaration

```
property ClientHeight: Integer;
```

The *ClientHeight* property is the height of the control's client area in pixels. For most controls, *ClientHeight* is exactly the same as *Height*. For forms, however, *ClientHeight* represents the height of the usable area inside the form's frame.

ClientHeight is a run-time only property for all controls except forms.

Example

This example reduces the height of the form's client area by half when the user clicks the button on the form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.ClientHeight := Form1.ClientHeight div 2;
end;
```

Note that only the client area is halved, not the entire form.

See also

ClientWidth property, *Height* property

ClientOrigin property

Applies to

All controls; *TForm* component

Declaration

```
property ClientOrigin: TPoint;
```

Run-time and read only. The *ClientOrigin* property is used to determine the screen coordinates (in pixels) of the top left corner of a control or form client area. *ClientOrigin* returns X and Y coordinates in a record of type *Point*.

Example

This example displays the Y screen coordinate of the top right corner of the *Button1* button client area:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.Text := IntToStr(Button1.ClientOrigin.Y);
end;
```

See also

ClientRect property

ClientRect property

Applies to

All controls; *TForm* component

Declaration

```
property ClientRect: TRect;
```

Run-time and read only. The *ClientRect* property is used to determine the size (in pixels) of a control or form client area. *ClientRect* returns the *Top*, *Bottom*, *Left*, and *Right* coordinates in one record of type *TRect*.

Example

The following code uses *ClientRect* to find and draw a line from the top left to the bottom right of the current form:

```
with ClientRect do
begin
  Canvas.MoveTo(Left,Top);
  Canvas.LineTo(Right, Bottom);
end;
```

See also

ClientOrigin property

ClientToScreen method

Applies to

All controls

Declaration

```
function ClientToScreen(Point: TPoint): TPoint;
```

The *ClientToScreen* method translates the given point from client area coordinates to global screen coordinates. In client area coordinates (0, 0) corresponds to the upper left corner of the control's client area. In screen coordinates (0, 0) corresponds to the upper left corner of the screen.

Using the *ClientToScreen* and *ScreenToClient* methods you can convert from one control's coordinate system to another control's coordinate system. For example,

```
P := TargetControl.ScreenToClient(SourceControl.ClientToScreen(P));
```

which converts *P* from coordinates in *SourceControl* to coordinates in *TargetControl*.

Example

This example uses two edit boxes on a form. When the user clicks a point on the form, the *X* screen coordinate appears in *Edit1*, and the *Y* screen coordinate appears in *Edit2*.

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  P, Q : TPoint;
begin
  P.X := X;                          { P is the TPoint record for the form}
  P.Y := Y;
  Q := ClientToScreen(P);            { Q is the TPoint for the screen }
  Edit1.Text := IntToStr(Q.X) + ' is the X screen coordinate';
  Edit2.Text := IntToStr(Q.Y) + ' is the Y screen coordinate';
end;
```

See also

ScreenToClient method

ClientWidth property

Applies to

All controls

Declaration

```
property ClientWidth: Integer;
```

The *ClientWidth* property is the horizontal size of the control's client area in pixels. For most controls, *ClientWidth* is exactly the same as *Width*. For forms, however, *ClientWidth* represents the width of the usable area inside the form's frame.

ClientWidth is a run-time only property for all components except forms.

Example

This example uses a button on a form. Each time the user clicks the button, the button grows 10 pixels wider.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.ClientWidth := Button1.ClientWidth + 10;
end;
```

See also

ClientHeight property, *Width* property

Clipboard variable — Clipbrd

Declaration

```
Clipboard: TClipboard;
```

The *Clipboard* variable declares an instance of the *TClipboard* object. Use *Clipboard* when you want to use the *TClipboard* object.

Clipboard is declared in the *Clipbrd* unit. Whenever you use *Clipboard* and the *TClipboard* object you must add *Clipbrd* to the **uses** clause of your unit.

ClipRect property

Applies to

TCanvas object

Declaration

```
property ClipRect: TRect;
```

Read only. The *ClipRect* property specifies a bounding clipping rectangle. The rectangle specified by *ClipRect* defines the outer boundaries of the drawing area of the canvas. Any drawing that occurs at coordinates outside the *ClipRect* are clipped and don't appear onscreen. For example, the *ClipRect* of the canvas of a form is the same size as the client area of the form.

See also

ClientRect property

Close method

Applies to

TClipboard object; *TDataBase*, *TForm*, *TMediaPlayer*, *TQuery*, *TStoredProc*, *TTable* components

For forms

Declaration

```
procedure Close;
```

The *Close* method closes a form. Calling the *Close* method on a form corresponds to the user selecting the Close menu item on the form's System menu. The *Close* method first calls the *CloseQuery* method to determine if the form can close. If *CloseQuery* returns *False*, the close operation is aborted. Otherwise, if *CloseQuery* returns *True*, the code attached to the *OnClose* event is executed. The *CloseAction* parameter of the *OnClose* event controls how the form is actually closed.

Example

The following method closes a form when a button called *Done* is clicked:

```
procedure TForm1.DoneButtonClick(Sender: TObject);
begin
  Close;
end;
```

See also

Hide method, *Open* method

For Clipboard objects

Declaration

```
procedure Close;
```

For Clipboard objects, *Close* closes the Clipboard if it is open. The Clipboard can be opened with a call to *Open* multiple times before being closed. Because the Clipboard object counts each time it is opened, your application must close it the same number of times it was opened before the Clipboard is actually closed.

Example

The following code closes the Clipboard:

```
Clipboard.Close;
```

See also

Clipboard variable

For media player controls

Declaration

```
procedure Close;
```

The *Close* method closes the open multimedia device.

Upon completion, *Close* stores a numerical error code in the *Error* property, and the corresponding error message in the *ErrorMessage* property.

The *Wait* property determines whether control is returned to the application before the *Close* method is completed. The *Notify* property determines whether *Close* generates an *OnNotify* event.

Close is called automatically when the application is terminated.

See also
Open method

For tables, queries, and stored procedures

Declaration

```
procedure Close;
```

The *Close* method closes the dataset, returning it to Inactive state. Calling *Close* is equivalent to setting the *Active* property to *False*.

Note *Post* is not called implicitly by the *Close* method. Use the *BeforeClose* event to post any pending edits explicitly.

For databases

Declaration

```
procedure Close;
```

The *Close* method closes the *TDatabase* component and all the dataset components linked to it. This is the same as setting the *Connected* property to *False*.

Example

```
  Database1.Close;
```

See also
CloseDatasets method

Close procedure **System**

Declaration

```
procedure Close(var F);
```

The *Close* procedure provides compatibility with existing Borland Pascal code. When writing applications for Delphi, you should use *CloseFile*.

The *Close* procedure terminates the association between the file variable and an external disk file.

F is a file variable of any file type opened using *Reset*, *Rewrite*, or *Append*. The external file associated with *F* is completely updated and then closed, freeing the file handle for reuse.

{**$I+**} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using {**$I–**}, you must use *IOResult* to check for I/O errors.

Example

```
var
  F: TextFile;
begin
  if OpenDialog1.Execute then  { Bring up open file dialog }
  begin
    AssignFile(F, OpenDialog1.FileName);
            { File selected in dialog }
    Reset(F);
    Edit1.Text := IntToStr(FileSize(F);
        { Put file size string in a TEdit control }
    CloseFile(F);   { Close file }
  end;
end;
```

See also

Append procedure, *AssignFile* procedure, *Reset* procedure, *Rewrite* procedure

CloseApplication method

Applies to

TReport component

Declaration

```
function CloseApplication(ShowDialogs: Boolean): Boolean;
```

The *CloseApplication* method stops ReportSmith Runtime, if it is running. *CloseApplication* sends a DDE message to terminate ReportSmith Runtime and looks for a DDE message from ReportSmith in return. If *CloseApplication* returns *True*, the ReportSmith Runtime received the message to terminate successfully; if it returns *False*, ReportSmith Runtime was not able to receive the message at the current time.

The value of the *ShowDialogs* parameter determines whether ReportSmith displays dialog boxes prompting users to save the existing report before closing, and so on. If *ShowDialogs* is *True*, the dialog boxes appear before ReportSmith closes. If the parameter is *False*, no dialog boxes appear.

See also

CloseReport method

CloseDatabase method

Applies to

TSession component

Declaration

```
procedure CloseDatabase(Database: TDatabase);
```

The *CloseDatabase* method closes a *TDatabase* component. The parameter *Database* specifies the *TDatabase* component to close. Normally, this is handled automatically when an application closes the last table in the database associated with a *TDatabase* component. *CloseDatabase* decrements the *Session's* reference count of the number of open database connections.

You should always use *CloseDatabase* with *OpenDatabase*, typically in a **try...finally** block to ensure that database connections are handled properly.

Example

```
Database := Session.OpenDatabase('DBDEMOS');
try
  begin
   {Do Something}
finally
    Session.CloseDatabase('DBDEMOS');
end;
```

See also

Session variable

CloseDatasets method

Applies to

TDataBase component

Declaration

```
procedure CloseDatasets;
```

The *CloseDatasets* method closes all of the dataset components linked to the *TDatabase* component, but does not close the database connection itself.

Example

```
Database1.CloseDatasets;
```

See also

Close method

CloseDialog method

Applies to

TFindDialog, *TReplaceDialog* components

Declaration

```
procedure CloseDialog;
```

The *CloseDialog* method closes the Find and Replace dialog boxes.

See also

Execute method

CloseFile procedure System

Declaration

```
procedure CloseFile(var F);
```

Due to naming conflicts, the *CloseFile* procedure replaces the Borland Pascal *Close* procedure. Use the *CloseFile* procedure instead of *Close* to terminate the association between the file variable and an external disk file.

F is a file variable of any file type opened using *Reset*, *Rewrite*, or *Append*. The external file associated with *F* is completely updated and then closed, freeing the file handle for reuse.

{**$I+**} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using {**$I–**}, you must use *IOResult* to check for I/O errors.

CloseLink method

Applies to

TDDEClientConv component

Declaration

```
function CloseLink;
```

The *CloseLink* method terminates an ongoing DDE conversation. After a link is closed, no DDE communication can take place between the DDE client and server until another link is opened.

Example

The following code terminates the DDE conversation.

```
DDEClientConv1.CloseLink;
```

See also

OnClose event, *OpenLink* method

CloseQuery method

Applies to

TForm component

Declaration

```
function CloseQuery: Boolean;
```

The *CloseQuery* method is called as part of a form's *Close* method processing to determine if the form can actually close. *CloseQuery* executes the code attached to the *OnCloseQuery* event. If the *OnCloseQuery* event handler assigns *False* to its *CanClose* parameter, *CloseQuery* will return *False* indicating that the form cannot close. Otherwise *CloseQuery* returns *True*, indicating that the form is ready to close.

The *CloseQuery* method of the main form of an MDI application automatically calls the *CloseQuery* method of each MDI child form before executing its own *OnCloseQuery* event. If any of the child forms return *False*, the main form's *CloseQuery* stops and also returns *False*. Your application can use the *OnCloseQuery* event to ask users if they want special processing to occur, such as saving information on the form, before the form is closed.

Example

When the user attempts to close the form in this example, a message dialog appears that asks if it is OK to close the form. If the user chooses No, the form doesn't close. If the user chooses OK, the form closes.

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
var
  ButtonSelected: Word;
begin
  ButtonSelected := MessageDlg('Is it OK to close the form?', mtInformation,
    [mbOk, mbNo], 0);
  if ButtonSelected = mrOk then
    CanClose := True
  else
    CanClose := False;
end;
```

See also
Close method, *OnCloseQuery* event

CloseReport method

Applies to
TReport component

Declaration

```
function CloseReport(ShowDialogs: Boolean): Boolean;
```

The *CloseReport* method determines whether the running of a ReportSmith report terminates. *CloseReport* sends a DDE message to ReportSmith Runtime and looks for a DDE message from ReportSmith Runtime in return. If *CloseReport* returns *True*, ReportSmith Runtime received the message to terminate the report. If *CloseReport* returns *False*, ReportSmith Runtime could not receive the DDE message at the current time.

The *ShowDialogs* parameter determines whether dialog boxes that prompt the user about saving the report appear before the report closes, and so on. If *ShowDialogs* is *True*, the dialog boxes appear. If it is *False*, the dialog boxes are not shown.

Example
The following code terminates the running report if the user chooses Yes from a dialog box:

```
if MessageDlg('Do you want to stop running ' + Report1.ReportName + ' ?',
  mtConfirmation, [mbYes, mbNo], 0) = mrYes then
  if Report1.CloseReport(False) then MessageDlg(Report1.ReportName + ' canceled.',
    mtInformation, [mbOK], 0);
```

See also
CloseApplication method

CloseUp method

Applies to
TDBLookupCombo component

Declaration

```
procedure CloseUp;
```

The *CloseUp* method closes an opened or “dropped-down” database lookup combo box.

See also
DropDown method

ClrEol procedure WinCrt

Declaration

```
procedure ClrEol;
```

The *ClrEol* procedure clears all characters from the cursor position to the end of the line without moving the cursor.

ClrEol sets all character positions to blanks with the currently defined text attributes.

Example

```
uses WinCrt;

begin
  ClrScr;
  Writeln('Hello there, how are you today?');
  Writeln('Press <enter> key...');
  Readln;
  GotoXY(1,2);
  ClrEol;
  Writeln ('Glad to hear it!');
end;
```

See also

ClrScr procedure

ClrScr procedure WinCrt

Declaration

```
procedure ClrScr;
```

The *ClrScr* procedure clears the active windows and returns the cursor to the upper left corner.

ClrScr sets all character positions to blanks with the currently defined text attributes.

Example

```
uses WinCrt;

begin
  Writeln('Hello.   Please the <enter> key...');
  Readln;
  ClrScr;
end;
```

See also

ClrEol procedure

CmdLine variable **System**

Declaration

```
var CmdLine: PChar;
```

In a program, the *CmdLine* variable contains a pointer to a null-terminated string that contains the command-line arguments specified when the application was started.

In a library, *CmdLine* is **nil**.

CmdShow variable **System**

Declaration

```
var CmdShow: Integer;
```

In a program, the *CmdShow* variable contains the parameter value that Windows expects to be passed to *ShowWindow* when the application creates its main window.

In a library, *CmdShow* is always zero.

Col property

Applies to
TDrawGrid, *TStringGrid* components

Declaration

```
property Col: Longint;
```

Run-time only. The value of the *Col* property indicates the current column of the cell that has input focus. You can use the *Col* property along with the *Row* property to determine which cell is selected at run time.

Example
This example uses a string grid with a label above it on a form. When the user clicks a cell in the grid, the location of the cursor is displayed in the label caption.

```
procedure TForm1.StringGrid1Click(Sender: TObject);
begin
  Label1.Caption := 'The cursor is in column ' + IntToStr(StringGrid1.Col + 1)
     +  ', row ' + IntToStr(StringGrid1.Row + 1);
end;
```

See also
ColCount property, *ColWidths* property, *DefaultColWidth* property

ColCount property

Applies to
TDrawGrid, *TStringGrid* components

Declaration

```
property ColCount: Longint;
```

Run-time only. The value of the *ColCount* property determines the number of columns in the grid. The default value is 5.

Example
The following line of code adds one column to a string grid named *MyStrngGrd*:

```
   MyStrngGrd.ColCount := MyStrngGrd.ColCount + 1;
```

See also
Col property, *ColWidths* property, *RowCount* property

Collapse method

Applies to
TOutlineNode object

Declaration

```
procedure Collapse;
```

The *Collapse* method collapses an outline item by assigning *False* to its *Expanded* property. When an outline item is collapsed, its sub-items are hidden and the plus picture or closed picture might be displayed, depending on the outline style specified in the *OutlineStyle* property of the *TOutline* component.

Example
The following code collapses the first outline item.

```
   Outline1.Items[1].Collapse;
```

See also
Expand method, *FullCollapse* method, *FullExpand* method, *PictureClosed* property, *PicturePlus* property

Collate property

Applies to
TPrintDialog component

Declaration

```
property Collate: Boolean;
```

The *Collate* property determines if the Collate check box is checked and, therefore, if collating is selected. Regardless of the initial setting of the *Collate* property, the user can always check or uncheck the Collate check box (and change the *Collate* property) to choose or not choose to collate the print job. The default setting is *False*.

Color property

Applies to

TBrush, *TFont*, *TPen* objects; *TBitBtn*, *TCheckBox*, *TColorDialog*, *TComboBox*, *TDBCheckBox*, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBLookupList*, *TDBMemo*, *TDBRadioGroup*, *TDBText*, *TDirectoryListBox*, *TDrawGrid*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TForm*, *TGroupBox*, *TLabel*, *TListBox*, *TMaskEdit*, *TMemo*, *TNotebook*, *TOutline*, *TPaintBox*, *TPanel*, *TRadioButton*, *TScrollBox*, *TStringGrid* components

Declaration

```
property Color: TColor;
```

For all components or objects except the Color dialog box, the *Color* property determines the background color of a form or the color of a control or graphics object.

If a control's *ParentColor* property is *True*, then changing the *Color* property of the control's parent automatically changes the *Color* property of the control. When you assign a value to a control's *Color* property, the control's *ParentColor* property is automatically set to *False*. These are the possible values of *Color*:

Value	Meaning
clBlack	Black
clMaroon	Maroon
clGreen	Green
clOlive	Olive green
clNavy	Navy blue
clPurple	Purple
clTeal	Teal
clGray	Gray
clSilver	Silver
clRed	Red
clLime	Lime green
clBlue	Blue
clFuchsia	Fuchsia
clAqua	Aqua
clWhite	White
clBackground	Current color of your Windows background

Value	Meaning
clActiveCaption	Current color of the title bar of the active window
clInactiveCaption	Current color of the title bar of inactive windows
clMenu	Current background color of menus
clWindow	Current background color of windows
clWindowFrame	Current color of window frames
clMenuText	Current color of text on menus
clWindowText	Current color of text in windows
clCaptionText	Current color of the text on the title bar of the active window
clActiveBorder	Current border color of the active window
clInactiveBorder	Current border color of inactive windows
clAppWorkSpace	Current color of the application workspace
clHighlight	Current background color of selected text
clHightlightText	Current color of selected text
clBtnFace	Current color of a button face
clBtnShadow	Current color of a shadow cast by a button
clGrayText	Current color of text that is dimmed
clBtnText	Current color of text on a button
clInactiveCaptionText	Current color of the text on the title bar of an inactive window
clBtnHighlight	Current color of the highlighting on a button

The second half of the colors listed here are Windows system colors. The color that appears depends on the color scheme users are using for Windows. Users can change these colors using the Control Panel in Program Manager. The actual color that appears will vary from system to system. For example, the color fuchsia may appear more blue on one system than another.

For the Color dialog box

When you use the Color dialog box to select a color, you are assigning a new color value to the dialog box's *Color* property. You can then use the value within the *Color* property and assign it to the *Color* property of another control.

Example

This code colors a form red:

```
Form1.Color := clRed;
```

The following code changes the color of an edit box control using the Color dialog box. The example displays the Color dialog box when the *Button1* button is clicked, allowing the user to select a color with the dialog box. The example then assigns the color value selected with the dialog box to the *Color* property of the edit box control:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if ColorDialog1.Execute then
```

```
      Edit1.Color := ColorDialog1.Color;
  end;
```

See also
ColorToRGB function, *ParentColor* property, *TColorDialog* component

ColoredButtons property

Applies to
TMediaPlayer component

Declaration

```
property ColoredButtons: TButtonSet;
```

The *ColoredButtons* property determines which of the buttons on the media player control has color. If a button is not colored with *ColoredButtons*, it appears in black-and-white when visible. All media player control buttons are colored by default.

Button	Value	Action
Play	*btPlay*	Plays the media player
Record	*btRecord*	Starts recording
Stop	*btStop*	Stops playing or recording
Next	*btNext*	Skips to the next track, or to the end if the medium doesn't use tracks
Prev	*btPrev*	Skips to the previous track, or to the beginning if the medium doesn't use tracks
Step	*btStep*	Moves forward a number of frames
Back	*btBack*	Moves backward a number of frames
Pause	*btPause*	Pauses playing or recording. If already paused when clicked, resumes playing or recording.
Eject	*btEject*	Ejects the medium

Example
The following example displays all of the media player component's buttons in color:

```
  TMediaPlayer1.ColoredButtons := [btPlay, btPause, btStop, btNext, btPrev, btStep, btBack,
    btRecord, btEject]
```

See also
EnabledButtons property, *VisibleButtons* property

ColorToRGB function

Graphics

Declaration

```
function ColorToRGB(Color: TColor): Longint;
```

The *ColorToRGB* function returns the RGB value that Windows uses from a *TColor* type used by Delphi. If the color represents a system color, the current RGB value for that system color is returned.

Example

The following code converts the color of the current form, *Form1*, to a Windows RGB value:

```
var
  L : Longint;
begin
  L := ColorToRGB(Form1.Color);
end;
```

See also

Color property

Cols property

Applies to

TStringGrid component

Declaration

```
property Cols[Index: Integer]: TStrings;
```

The *Cols* property is an array of the strings and their associated objects in a column. The number of strings and associated objects is always equal to the value of the *ColCount* property, the number of columns in the grid. Use the *Cols* property to access the strings and their associated objects within a particular column in the grid. The *Index* parameter is the number of the column you want to access; the *Index* value of the first column in the grid is zero.

Example

The following line of code adds the string 'Hello' to the end of the list of strings in column four of the string grid named *StringGrid1*:

```
  StringGrid1.Cols[3].Add('Hello');
```

See also

Cells property, *Objects* property, *Rows* property

Columns property

Applies to

TDBRadioGroup, *TDirectoryListBox*, *TListBox*, *TRadioGroup* components

Declaration

```
property Columns: Longint;
```

The *Columns* property denotes the number of columns in the list box or radio group box. Specify the number of columns you want for the list box or radio group box as the value of *Columns*.

Example

This example uses a list box and a button on a form. Each time the user clicks the button, the string 'Hello' is added to the list box. When the list box is filled, a new column is created and subsequent new strings are added to the new column:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if ListBox1.Columns < 1 then
    ListBox1.Columns := 1;
  Listbox1.Items.Add('Hello');
  if Listbox1.Height <= ((Listbox1.ItemHeight * Listbox1.Items.Count)
     / ListBox1.Columns) then
        Listbox1.Columns := Listbox1.Columns + 1;
end;
```

ColWidths property

Applies to

TDrawGrid, *TStringGrid* components

Declaration

```
property ColWidths[Index: Longint]: Integer;
```

Run-time only. The *ColWidths* property determines the width in pixels of all the cells within the column referenced by the *Index* parameter.

By default, all the columns are the same width, the value of the *DefaultColWidth* property. To change the width of all columns within a grid, change the *DefaultColWidth* property value.

To change the width of one column without affecting others, change the *ColWidths* property. Specify the column you want to change as the value of the *Index* parameter. Remember the first column always has an *Index* value of 0.

Example

The following code changes the width of column 0 in the string grid called *StringGrid1* to twice the default value.

```
StringGrid1.ColWidths[0] := StringGrid1.DefaultColWidth * 2;
```

See also

RowHeights property

Command property

Applies to

TMenuItem component

Declaration

```
property Command: Word;
```

Run-time and read only. The *Command* property value is the command number passed to Windows and the number that arrives in the WM_COMMAND message sent by Windows to the form when the user chooses this menu item on the menu. *Command* is useful only if you are handling WM_COMMAND messages directly.

Example

The following procedure is a WM_COMMAND message handler. It checks the *ItemID* field of *Msg* to see if the message was generated by a menu item called *MenuThink*. If so, it displays a message dialog box. When writing message handlers, remember to call *Inherited* afterward, if necessary, so Windows can perform default message processing.

```
procedure TForm1.WMCommand(var Msg: TWMCommand);
begin
  if Msg.ItemID = MenuThink.Command then
    MessageDlg('This is the Think command', mtInformation, [mbOk], 0);
  Inherited;
end;
```

Commit method

Applies to

TDataBase component

Declaration

```
procedure Commit;
```

The *Commit* method commits the current transactions and thus all modifications made to the database since the last call to *StartTransaction*. If no transaction is active, Delphi will raise an exception. Use this method only when connected to a server database.

Example

```
with Database1 do
  begin
  StartTransaction;
{ Update one or more records in tables linked to Database1 }
...
  Commit;
  end;
```

See also

Rollback method

CompareStr function

SysUtils

Declaration

```
function CompareStr(const S1, S2: string): Integer;
```

CompareStr compares *S1* to *S2*, with case-sensitivity. The return value is less than 0 if *S1* is less than *S2*, 0 if *S1* equals *S2*, or greater than 0 if *S1* is greater than *S2*. The compare operation is based on the 8-bit ordinal value of each character and is not affected by the currently installed language driver.

Example

The following code compares *String1*, 'STEVE', to *String2*, 'STEVe'. Note that *CompareStr* returns a number less than 0 because the value of 'e' is greater than the value of 'E'.

```
var
  String1, String2 : string;
  I : integer;
begin
  String1 := 'STEVE';
  String2 := 'STEVe';
  I := CompareStr(String1, String2);  { the value of I is < 0 }
  if I < 0 then
    MessageDlg('The strings are not equal', mtWarning, [mbOK], 0)
end;
```

See also

CompareText function

CompareText function

SysUtils

Declaration

```
function CompareText(const S1, S2: string): Integer;
```

The *CompareText* function compares the strings *S1* and *S2* and returns 0 if they are equal. If *S1* is greater than *S2*, *CompareText* returns an integer greater than 0. If *S1* is less than *S2*, *CompareText* returns an integer less than 0. The *CompareText* function is not case sensitive. For example, *CompareText* finds 'object pascal' and 'Object Pascal' to be equal.

Example

The following code compares *String1*, 'ABC, to *String2*, 'aaa'. Because *CompareText* is case insensitive, *String2* is larger.

```
var
  String1, String2 : string;
```

```
    I : integer;
  begin
    String1 := 'ABC';
    String2 := 'aaa';
    I := CompareStr(String1, String2);  { the value of I is < 0 }
    if I < 0 then
      MessageDlg('The strings are not equal', mtWarning, [mbOK], 0)
  end;
```

See also

CompareStr function

ComponentCount property

Applies to

All components

Declaration

```
property ComponentCount: Integer;
```

Run-time and read only. The *ComponentCount* property indicates the number of components owned by the component as listed in the *Components* array property. For example, *ComponentCount* of a form contains the same number of items as in the *Components* list of a form.

Note *ComponentCount* is always 1 more than the highest *Components* index, because the first *Components* index is always 0.

Example

This code uses several controls on a form, including a button and an edit box. When the user clicks the button, the code counts all the components on the form and displays the number in the *Edit1* edit box. While the components are being counted, each is evaluated to see if it is a button component. If the component is a button, the code changes the font on the button face.

```
  procedure TForm1.Button1Click(Sender: TObject);
  var
    I: Integer;
  begin
    for I := 0 to ComponentCount -1 do
       if Components[I] is TButton then
         TButton(Components[I]).Font.Name := 'Courier';
    Edit1.Text := IntToStr(ComponentCount) + ' components';
  end;
```

See also

ComponentIndex property, *Components* property

ComponentIndex property

Applies to
All components

Declaration

```
property ComponentIndex: Integer;
```

Run-time and read only. The *ComponentIndex* property indicates the position of the component in its owner's *Components* property list. The first component in the list has a *ComponentIndex* value of 0, the second has a value of 1, and so on.

Example
The following code uses a button and a wide edit box on a form. When the user clicks the button, the edit box displays the index value of the button component:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.Text := 'The index of the button is ' + IntToStr(Button1.ComponentIndex);
end;
```

See also
ComponentCount property, *Components* property

Components property

Applies to
All components

Declaration

```
property Components[Index: Integer]: TComponent;
```

Run-time and read only. The *Components* array property is a list of all components owned by the component. You can use the *Components* property to access any of these owned components, such as the controls owned by a form. The *Components* property is most useful if you need to refer to owned components by number rather than name.

Don't confuse the *Components* property with the *Controls* property. The *Components* property lists all components that are owned by the component, whereas the *Controls* property lists all the controls that are child windows of this control. All components on a form are owned by the form, and therefore, they appear in the form's *Components* property list.

Consider this example. If you put a control in a group box, the form still owns the control, but the control's window parent is the group box control, and therefore, is listed in the group box's *Controls* property array.

Example

This code uses several controls on a form, including a button and an edit box. When the user clicks the button, the code counts all the components on the form and displays the number in the *Edit1* edit box. While the components are being counted, each is evaluated to see if it is a button component. If the component is a button, the code changes the font on the button face.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ComponentCount -1 do
     if Components[I] is TButton then
       TButton(Components[I]).Font.Name := 'Courier';
  Edit1.Text := IntToStr(ComponentCount) + ' components';
end;
```

See also

ComponentCount property, *ComponentIndex* property, *Owner* property, *Parent* property, *TabOrder* property

Concat function System

Declaration

```
function Concat(s1 [, s2,..., sn]: string): String;
```

The *Concat* function merges two or more strings into one large string.

Each parameter is a string-type expression. The result is the concatenation of all the string parameters. If the resulting string is longer than 255 characters, it is truncated after the 255th character.

Using the plus (+) operator has the same effect on two strings as using the *Concat* function:

```
S := 'ABC' + 'DEF';
```

Example

```
 var
  S: string;
begin
  S := Concat('ABC', 'DEF');     { 'ABCDE' }
end;
```

See also

Copy function, *Delete* procedure, *Insert* procedure, *Length* function, *Pos* function

ConfirmDelete property

Applies to
TDBNavigator component

Declaration

```
property ConfirmDelete: Boolean;
```

The *ConfirmDelete* property determines whether a message box asking you to confirm the deletion when the user uses the database navigator to delete the current record in the dataset. If *ConfirmDelete* is *True*, a prompting message box appears and the record isn't deleted unless the user chooses the OK button. If *ConfirmDelete* is *False*, no message box appears and the record is deleted.

The default value is *True*.

See also
VisibleButtons property

Connect method

Applies to
TReport component

Declaration

```
function Connect(ServerType: Word; const ServerName, UserName, Password,
   DatabaseName: string): Boolean;
```

The *Connect* method connects the report to a database, bypassing the ReportSmith log in dialog box. Specify the server type and name with the *ServerType* and *ServerName* parameters. Specify the user name, the log-in password, and the name of the database using the *UserName*, *Password*, and *DatabaseName* parameters.

Connected property

Applies to
TDataBase component

Declaration

```
property Connected: Boolean;
```

The *Connected* property indicates whether the *TDatabase* component has established a connection to a database. *Connected* will be set to *True* when an application opens a table in a database (logging in to a server, if required). It will be set back to *False* when the table is closed (unless *KeepConnection* is *True*). Set *Connected* to *True* to establish a connection to a database without opening a table. Set *Connected* to *False* to close a database connection.

The *KeepConnection* property of *TDatabase* specifies whether to maintain database connections when no tables in the database are open. The *KeepConnections* property of *TSession* specifies whether to maintain database connections when there is no explicit *TDatabase* component for the database.

Example

```
Database1.Connected := True;
```

ConnectMode property

Applies to
TDDEClientConv component

Declaration

```
property ConnectMode: TDataMode;
```

The *ConnectMode* property determines the type of connection to establish when initiating a link with a DDE server application. These are the possible values:

Value	Meaning
ddeAutomatic	The link is automatically established when the form containing the *TDDEClient* component is created at run time. This is the default value.
ddeManual	The link is established only when the *OpenLink* method is called.

Example
The following code sets the connect mode of *DDEClientConv1* to manual.

```
DDEClientConv1.ConnectMode := ddeManual;
```

ContainsControl method

Applies to
All windowed controls

Declaration

```
function ContainsControl(Control: TControl): Boolean;
```

The *ContainsControl* method indicates whether a specified control exists within a control. If the method returns *True*, the control specified as the value of the *Control* parameter exists within the control. If the method returns *False*, the specified control is not within the control.

Example

This example uses a label, a list box, and a button on a form. When the user clicks the button, the caption of the label reports that the form contains the list box:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if ContainsControl(ListBox1) then
    Label1.Caption := 'The form contains ListBox1';
end;
```

Continue procedure   System

Declaration

```
procedure Continue;
```

The *Continue* procedure has the flow of control proceed to the next iteration of the calling **for**, **while**, or **repeat** statement.

The compiler reports an error if a call to *Continue* isn't enclosed by a **for**, **while**, or **repeat** statement.

Example

```
var
  F: File;
  i: integer;
begin
  for i := 0 to (FileListBox1.Items.Count - 1) do begin
    if FileListBox1.Selected[i] then begin
      if not FileExists(FileListBox1.Items.Strings[i]) then begin
        MessageDlg('File: ' + FileListBox1.Items.Strings[i] +
                   ' not found', mtError, [mbOk], 0);
        Continue;
      end;
      AssignFile(F, FileListBox1.Items.Strings[i]);
      Reset(F, 1);
      ListBox1.Items.Add(IntToStr(FileSize(F)));
      CloseFile(F);
    end;
  end;
end;
```

See also

Break procedure, *Exit* procedure, *Halt* procedure

ControlAtPos method

Applies to

All windowed controls

Declaration

```
function ControlAtPos(Pos: TPoint; AllowDisabled: Boolean): TControl;
```

The *ControlAtPos* method returns the windowed control's child control (from those in the *Controls* array property) located at the screen coordinates passed in *Pos*. If there is no control at the specified position, *ControlAtPos* returns **nil**. The *AllowDisabled* parameter controls whether the search for controls includes disabled controls.

ControlCount property

Applies to

All controls

Declaration

```
property ControlCount: Integer;
```

Run-time and read only. The *ControlCount* property indicates the number of controls that are children of the control. The children are listed in the *Controls* property array.

Note The value of *ControlCount* is always 1 greater than the highest *Controls* index, because the first *Controls* index is 0.

Example

This example uses a group box on a form, with several controls contained within the group box. The form also has an edit box and a button outside of the group box. The code counts each control's child controls turning each of them invisible as they are counted. The total number of controls counted appears in the edit box.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  for I:= 0 to GroupBox1.ControlCount -1 do
     GroupBox1.Controls[I].Visible := False;
  Edit1.Text := IntToStr(GroupBox1.ControlCount) + ' controls';
end;
```

See also

Controls property

Controls property

Applies to

All controls

Declaration

```
property Controls[Index: Integer]: TControl;
```

Run-time and read only. The *Controls* property is an array of all controls that are children of the control. The *Controls* property is most useful if you have a need to refer to the children of a control by number rather than name.

Don't confuse the *Controls* property with the *Components* property. The *Components* property lists all components that are owned by the component, while the *Controls* property lists all the controls that are child windows of the control. All components put on a form are owned by the form, and therefore, they appear in the form's *Components* property list.

For example, if you put a control in a group box, the form still owns the control, but the control's window parent is the group box control, and therefore, is listed in the group box's *Controls* property array.

Example

This example uses a group box on a form, with several controls contained within the group box. The form also has an edit box and a button outside of the group box. The code counts each control's child controls turning each of them invisible as they are counted. The total number of controls counted displays in the edit box.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  for I:= 0 to GroupBox1.ControlCount -1 do
     GroupBox1.Controls[I].Visible := False;
  Edit1.Text := IntToStr(GroupBox1.ControlCount) + ' controls';
end;
```

See also

ControlCount property, *Owner* property, *Parent* property

ConvertDlgHelp property

Applies to

TOLEContainer component

Declaration

```
property ConvertDlgHelp: THelpContext;
```

The *ConvertDlgHelp* property specifies the context-sensitive help identification number for the Convert dialog box. If your application is programmed for online help, specify an integer value for *ConvertDlgHelp* to identify the online help topic to be called when the user chooses Help from the Convert dialog box. If the application is not programmed for context-sensitive online Help, or if zero is specified for*ConvertDlgHelp*, choosing Help from the Convert dialog box will have no effect.

The Convert dialog box enables the user to convert an OLE object to another object type. To enable the Convert dialog box, a menu item but be designated the OLE object menu item in the *ObjectMenuItem* property of a form. Then, when an OLE container containing an OLE object is selected at run time, the OLE Object menu item will be available on the menu bar of the form. If the OLE server application supports object conversion, choose Convert from the OLE object menu item to display the Convert dialog box.

Note You don't need to program your application to provide the functionality of the Convert dialog box and the OLE object menu item. This functionality comes from the OLE server application automatically when an OLE container has focus. The only step required is to identify the name of a menu item in the *ObjectMenuItem* property.

Example

The following code assigns 531 to the context-sensitive Help identification number of the OLE Convert dialog box.

```
OLEContainer1.ConvertDlgHelp := 531;
```

Copies property

Applies to

TPrintDialog component

Declaration

```
property Copies: Integer;
```

The value of the *Copies* property determines the number of copies of the print job to print. If you change the value of *Copies* at design time, the value you specify is the default value in the edit box control when the Print dialog box appears. The default value is 0.

Example

The following code sets the default number of copies for the print dialog box, *PrintDialog1*, to 3 before displaying the dialog box:

```
PrintDialog1.Copies := 3;
PrintDialog1.Execute;
```

Copy function — System

Declaration

```
function Copy(S: string; Index, Count: Integer): string;
```

The *Copy* function returns a substring of a string.

S is a string-type expression. *Index* and *Count* are integer-type expressions. *Copy* returns a string containing *Count* characters starting with at *S*[*Index*].

If Index is larger than the length of *S*, *Copy* returns an empty string.

If *Count* specifies more characters than are available, the only the characters from *S*[*Index*] to the end of *S* are returned.

Example

```
 var S: string;
begin
  S := 'ABCDEF';
  S := Copy(S, 2, 3);                                                { 'BCD' }
end;
```

See also

Concat function, *Delete* procedure, *Insert* procedure, *Length* function, *Pos* function

CopyMode property

Applies to

TCanvas object

Declaration

```
property CopyMode: TCopyMode;
```

The *CopyMode* property determines how a canvas treats an image copied from another canvas. By default, *CopyMode* is *cmSrcCopy*, meaning that pixels from the other canvas are copied to the canvas, overwriting any image already there. By changing *CopyMode*, you can create many different effects. The following table shows possible values of *CopyMode* and describes each:

Value	Meaning
cmBlackness	Turns all output black.
cmDstInvert	Inverts the destination bitmap.
cmMergeCopy	Combines the pattern and the source bitmap by using the Boolean AND operator.
cmMergePaint	Combines the inverted source bitmap with the destination bitmap by using the Boolean OR operator.
cmNotSrcCopy	Copies the inverted source bitmap to the destination.

Value	Meaning
cmNotSrcErase	Inverts the result of combining the destination and source bitmaps by using the Boolean OR operator.
cmPatCopy	Copies the pattern to the destination bitmap with the pattern by using the Boolean XOR operator.
cmPatInvert	Combines the destination bitmap with the pattern by using the Boolean XOR operator
cmPatPaint	Combines the inverted source bitmap with the pattern by using the Boolean OR operator. Combines the result of this operation with the destination bitmap by using the Boolean OR operator.
cmSrcAnd	Combines pixels from the destination and source bitmaps by using the Boolean AND operator.
cmSrcCopy	Copies the source bitmap to the destination bitmap.
cmSrcErase	Inverts the destination bitmap and combines the result with the source bitmap by using the Boolean AND operator.
cmSrcInvert	Combines pixels from the destination and source bitmaps by using the Boolean XOR operator.
cmSrcPaint	Combines pixels from the destination and source bitmaps by using the Boolean OR operator.
cmWhiteness	Turns all output white.

Example

The following code copies the the inverted source bitmap to the *Canvas* of *Form2*:

```
Form2.Canvas.CopyMode := cmNotSrcCopy;
Form2.Canvas.CopyRect(ClientRect, Canvas, ClientRect);
```

See also

CopyRect method

CopyParams method

Applies to

TStoredProc component

Declaration

```
procedure CopyParams(Value: TParams);
```

The *CopyParams* method copies all of the parameter information from the stored procedure component to *Value*. Use this method to copy parameters from one stored procedure component to another.

Example

```
{ Copy all parameters from StoredProc1 to StoredProc2 }
StoredProc1.CopyParams(StoredProc2.Params);
```

CopyRect method

Applies to

TCanvas object

Declaration

```
procedure CopyRect(Dest: TRect; Canvas: TCanvas; Source: TRect);
```

The *CopyRect* method copies part of an image from another canvas into the canvas object. The *Dest* property specifies the destination rectangle on the destination canvas where the image will be copied. The *Canvas* property specifies the source canvas. The *Source* property specifies the source rectangle from the source canvas that will be copied.

Example

The following code copies the the inverted source bitmap to the *Canvas* of *Form2*:

```
Form2.Canvas.CopyMode := cmNotSrcCopy;
Form2.Canvas.CopyRect(ClientRect, Canvas, ClientRect);
```

See also

CopyMode property

CopyToClipboard method

Applies to

TDBEdit, *TDBImage*, *TDBMemo*, *TDDEServerItem*, *TEdit*, *TMemo*, *TOLEContainer* components

For edit boxes and memos

Declaration

```
procedure CopyToClipboard;
```

The *CopyToClipboard* method copies the text selected in the control to the Clipboard, replacing any text that exists there. If no text is selected, nothing is copied.

Example

The following method copies the selected text from the memo control named *Memo1* to the Clipboard and pastes it into an edit box named *Edit1* when the user clicks the button named *Button1*:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.CopyToClipboard;
  Edit1.PasteFromClipboard;
end;
```

See also
Clear method, *ClearSelection* method, *CutToClipboard* method, *PasteFromClipboard* method

For OLE containers

Declaration

```
procedure CopyToClipboard(Clear: Boolean);
```

The *CopyToClipboard* method copies the OLE object contained in an OLE container to the Clipboard, as well as OLE information. You can then create a link by activating an OLE container application and executing an Edit|Paste Special command, or its equivalent in the command structure of the OLE container application. To paste an object into a *TOLEContainer* component, call the *PasteSpecialDlg* function.

For example, after your application calls the *CopyToClipboard* method of a OLE container component, you can manually activate Quattro Pro for Windows. Select a location in the worksheet and choose Paste Format from the Edit menu of Quattro Pro for Windows to embed the OLE object in the worksheet.

If the *Clear* parameter is *True*, the prior contents of the Clipboard are deleted before *CopyToClipboard* places its data on the Clipboard. If *Clear* is *False*, the Clipboard won't be cleared before the copy.

Example
The following code copies the OLE object in *OLEContainer1* to the Clipboard without clearing the contents first.

```
OLEContainer1.CopyToClipboard(False);
```

See also
Clear method

For DDE server items

Declaration

```
procedure CopyToClipboard;
```

The *CopyToClipboard* method copies the text data specified in the *Text* or *Lines* property of a DDE server item component to the Windows Clipboard, as well as DDE link information. You can then create a link by activating the DDE client application, selecting the topic and item of the DDE conversation, and executing an Edit|Paste Link command, or its equivalent in the command structure of the DDE client application.

CopyToClipboard can be used to create a DDE link at run-time only. To create a link at design time, select the DDE server item component and choose Edit|Copy from the menu. Then, activate the DDE server application and paste the link according to the rules of the DDE server application. See the documentation for the DDE server application for specific information about pasting the link.

If the *Clear* parameter is *True*, the prior contents of the Clipboard are deleted before *CopyToClipboard* places its data on the Clipboard. If *Clear* is *False*, the Clipboard won't be cleared before the copy.

Example

The following code copies the DDE link information of *DDEServerItem1* to the Clipboard, clearing the contents of the Clipboard before the copy.

```
DDEServerItem1.CopyToClipboard;
```

See also

Clear method

For database images

Declaration

```
procedure CopyToClipboard;
```

The *CopyToClipboard* method copies the image of the database image component to the Clipboard.

Example

The following code copies the contents of *DBImage1* to the Clipboard without clearing the contents of the Clipboard first.

```
DBImage1.CopyToClipboard(False);
```

See also

CutToClipboard method, *PasteFromClipboard* method

Cos function System

Declaration

```
function Cos(X: Real): Real;
```

The *Cos* function returns the cosine of the angle *X*, in radians.

Example

```
 var R: Real;
begin
  R := Cos(Pi);
end;
```

See also
ArcTan function, *Sin* function

Count property

Applies to
TIndexDefs, *TFieldDefs*, *TList*, *TParams*, *TStringList*, *TStrings* objects; *TMenuItem* component

For lists and menu items

Declaration

```
property Count: Integer;
```

Run-time and read only. The *Count* property contains the number of items in a list or in a menu item.

For string and string list objects, *Count* is the number of strings in the list of strings. For list objects, *Count* is the number of items in the list.

For menu items, *Count* contains the number of subitems that belongs to a menu item. Subitems can be the menu items in a drop-down or pop-up menu, or the items in a submenu.

For example, if you have a File menu item on the main menu bar, but haven't added any commands to the File menu yet, the File menu's *Count* property value is 0. If you add New and Open commands to the File menu, the *Count* property value is 2. Because New and Open are also menu items, they too have *Count* property values. Unless either of these menu items have submenus, their *Count* property values are 0.

Example
The following code displays the number of items in a list box in the caption of a label when the user clicks the *CountItems* button:

```
procedure TForm1.CountItemsClick(Sender: TObject);
begin
  Label1.Caption := 'There are ' + IntToStr(ListBox1.Items.Count) +
    ' items in the listbox.';
end;
```

The following example assumes the form contains a main menu component, which includes a File menu and a label. This code displays the number of menu items that make up the File menu.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := IntToStr(FileMenu.Count):
end;
```

See also

Items property, *List* property, *Strings* property

For TParams objects

Declaration

```
function Count: Integer;
```

The *Count* method returns the number of entries in *Items*.

Example

```
{ Assign 99999 to any integer parameter which does not have a value }
for I := 0 to Params.Count - 1 do
  if (Params.Items[I].IsNull) and (Params.Items[I].DataType = ftInteger) then
{ Items is the default property, so you can omit its name }
    Params[I].AsInteger := 99999;
```

For TFieldDefs objects

Declaration

```
property Count: Integer;
```

The *Count* property specifies the total number of *TFieldDef* objects in this *TFieldDefs* object.

See also

Items property

For TIndexDefs objects

Declaration

```
property Count: Integer;
```

Run-time and read only. The *Count* property holds the number of entries in the *Items* property.

Create method

Applies to

All objects and components

For TIniFile objects

Declaration

```
constructor Create(const FileName: string);
```

The *Create* method allocates memory to create a *TIniFile* object and passes it the file name of the .INI file. Delphi looks for the specified .INI file in the Windows directory unless you include a path in the file name.

Example

This code creates an .INI file object and passes it the name of the .INI file, SUPERAPP.INI:

```
var
  IniFile: TIniFile;
begin
  IniFile := TIniFile.Create('SUPERAPP.INI');
  IniFile.Free;
end;
```

For outline nodes

Declaration

```
constructor Create(AOwner: TCustomOutline);
```

The *Create* method creates a new outline node owned by the outline passed in the *AOwner* parameter. You shouldn't need to call *Create*, as this is done for you when you add a new subitem to the outline with the *Add* method.

For control scroll bars

Declaration

```
constructor Create(AControl: TScrollingWinControl; AKind: TScrollBarKind);
```

The *Create* method creates a new control scroll bar. *AControl* specifies the component that owns the control scroll bar. *AControl* is of type *TScrollingWinControl*, which is simply a base class for *TForm* and *TScrollBox* components. *AKind* specifies the type of scroll bar, either *sbHorizontal* or *sbVertical*.

For TIndexDef objects

Declaration

```
constructor Create(Owner: TIndexDefs; const Name, Fields: string; Options: TIndexOptions);
```

The *Create* constructor creates a new *TIndexDef* object using the *Name*, *Fields*, and *Options* parameters and adds it to the *Items* property of the *Owner* parameter.

For TIndexDefs objects

Declaration

```
constructor Create(Table: TTable);
```

The *Create* constructor creates a new *TIndexDefs* object for the *Table* parameter.

For blob streams

Declaration

```
constructor Create(Field: TBlobField; Mode: TBlobStreamMode);
```

The *Create* method links a *TBlobField, TBytesField or TVarBytesField to the TBlobStream. Mode may be one of the following elements of TBlobStreamMode: bmRead* to access existing data in the *field; bmWrite to clear the contents of the field and assign a new value; bmReadWrite to modify an existing value*

Example

```
{ Link BlobStream1 to MyBlobField for data access only }
BlobStream1 := TBlobStream.Create(MyBlobField, bmRead);
```

For all other components

Declaration

```
constructor Create(AOwner: TComponent);
```

The *Create* method allocates memory to create the component and initializes its data as needed. Each object can have a *Create* method customized to create that particular kind of object. The owner of the created component is passed in the *AOwner* parameter.

Usually you don't need to create objects manually. Objects you design in Delphi are automatically created for you when you run the application and destroyed when you close the application.

If you construct a component by calling *Create*, and give it an owner, the owner disposes of the component when the owner is destroyed. If you don't want another component to own the created component, pass *Self* in the *AOwner* parameter.

Example

The following code creates a *TButton* and makes *Form1* the owner.

```
var
  Button1: TButton;
begin
  Button1 := TButton.Create(Form1);
end;
```

See also
Free method

For all other objects

Declaration

```
constructor Create;
```

The *Create* method allocates memory to create the object and initializes its data as needed. Each object can have a *Create* method customized to create that particular kind of object.

Example
The following code creates a *TBitmap* and loads the bitmap graphic file C:\WINDOWS\256COLOR.BMP into it. Then, the bitmap is drawn in a paint box by the *OnPaint* event handler of *PaintBox1*.

```
procedure TForm1.PaintBox1Paint(Sender: TObject);
var
  Bitmap1: TBitmap;
begin
  Bitmap1 := TBitmap.Create;
  Bitmap1.LoadFromFile('c:\windows\256color.bmp');
  PaintBox1.Canvas.Draw(0, 0, Bitmap1);
  Bitmap1.Free;
end;
```

See also
Free method

CreateField method

Applies to
TFieldDef object

Declaration

```
function CreateField(Owner: TComponent): TField;
```

CreateField creates a *TField* component of the appropriate type that corresponds to the *TFieldDef* object itself. *Owner* is the dataset component containing the field.

CreateForm method

Applies to
TApplication component

Declaration

```
procedure CreateForm(FormClass: TFormClass; var Reference);
```

The *CreateForm* method creates a new form of the type specified by the *FormClass* parameter and assigns it to the variable given by the *Reference* parameter. The owner of the new form is the *Application* object. The form created by the first call to *CreateForm* in a project becomes the project's main form.

A Delphi project typically contains one or more calls to *CreateForm* in the project's main statement part, but there is seldom any need for you to call *CreateForm* yourself.

Example

The following code creates *Form1* of type *TForm1*.

```
  Application.CreateForm(TForm1, Form1);
```

CreateNew method

Applies to

TForm component

Declaration

```
constructor CreateNew(AOwner: TComponent);
```

The *CreateNew* method creates a new instance of the current form type.

CreateParam method

Applies to

TParams object

Declaration

```
function CreateParam(FldType: TFieldType; const ParamName: string;
  ParamType: TParamType): TParam;
```

The *CreateParam* method attempts to create a new entry in *Items*, using the *FieldType*, *ParamName*, and *ParamType* parameters.

Example

```
  { Create a new parameter for CustNo and assign a value of 999 to it }
  with Params.CreateParam(ftInteger, ‘CustNo’, ptInput) do
    AsInteger := 999;
```

CreateTable method

Applies to

TTable component

Declaration

```
procedure CreateTable;
```

The *CreateTable* method creates a new empty database table. Before calling this method, the *DatabaseName*, *TableName*, *TableType*, *FieldDefs* and *IndexDefs* properties must be assigned values.

Example

```
with Table1 do
  begin
  Active := False;
  DatabaseName := 'Delphi_Demos';
  TableName := 'CustInfo';
  TableType := ttParadox;
  with FieldDefs do
    begin
    Clear;
    Add('Field1', ftInteger, 0);
    Add('Field2', ftInteger, 0);
    end;
  with IndexDefs do
    begin
    Clear;
    Add('Field1Index', 'Field1', [ixPrimary, ixUnique]);
    end;
  CreateTable;
  end;
```

CSeg function **System**

Declaration

```
function CSeg: Word;
```

The *CSeg* function returns the current value of the CS register.

The result is the segment address of the code segment that called *CSeg*.

Example

```
function MakeHexWord(w: Word): string;
const
  hexChars: array [0..$F] of Char = '0123456789ABCDEF';
var
```

```
    HexStr : string;
  begin
    HexStr := '';
    HexStr := HexStr + hexChars[Hi(w) shr 4];
    HexStr := HexStr + hexChars[Hi(w) and $F];
    HexStr := HexStr + hexChars[Lo(w) shr 4];
    HexStr := HexStr + hexChars[Lo(w) and $F];
    MakeHexWord := HexStr;
  end;

  procedure TForm1.Button1Click(Sender: TObject);
  var
    i: Integer;
    Y: Integer;
    S: string;
  begin
    Y := 10;
    S := 'The current code segment is $' + MakeHexWord(CSeg);
    Canvas.TextOut(5, Y, S);
    Y := Y + Canvas.TextHeight(S) + 5;
    S := 'The global data segment is $' + MakeHexWord(DSeg);
    Canvas.TextOut(5, Y, S);
    Y := Y + Canvas.TextHeight(S) + 5;
    S := 'The stack segment is $' + MakeHexWord(SSeg);
    Canvas.TextOut(5, Y, S);
    Y := Y + Canvas.TextHeight(S) + 5;
    S := 'The stack pointer is at $' + MakeHexWord(SPtr);
    Canvas.TextOut(5, Y, S);
    Y := Y + Canvas.TextHeight(S) + 5;
    S := 'i is at offset $' + MakeHexWord(Ofs(i));
    Canvas.TextOut(5, Y, S);
    Y := Y + Canvas.TextHeight(S) + 5;
    S := 'in segment $' + MakeHexWord(Seg(i));
    Canvas.TextOut(5, Y, S);
  end;
```

See also

DSeg function, *SSeg* function

Ctl3D property

Applies to

TBitBtn, *TButton*, *TCheckBox*, *TColorDialog*, *TComboBox*, *TDBCheckBox*, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBLookupList*, *TDBMemo*, *TDBNavigator*, *TDBRadioGroup*, *TDirectoryListBox*, *TDrawGrid*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TFindDialog*, *TFilterComboBox*, *TFontDialog*, *TForm*, *TGroupBox*, *TListBox*, *TMaskEdit*, *TMemo*, *TNotebook*, *TOLEContainer*, *TOpenDialog*, *TOutline*, *TPanel*, *TRadioButton*, *TReplaceDialog*, *TSaveDialog*, *TScrollBar*, *TScrollBox*, *TStringGrid* components

Declaration

```
property Ctl3D: Boolean;
```

The *Ctl3D* property determines whether a control has a three-dimensional (3-D) or two-dimensional look. If *Ctl3D* is *True*, the control has a 3-D appearance. If *Ctl3D* is *False*, the control appears normal or flat. The default value of *Ctl3D* is *True*.

For dialog boxes, the value of *Ctl3D* affects the dialog box and all the controls it contains.

If a control's *ParentCtl3D* property is *True*, then changing in the *Ctl3D* property of the control's parent automatically changes the *Ctl3D* property of the control. When you assign a value directly to a control's *Ctl3D* property, the control's *ParentCtl3D* property is automatically set to *False*.

Note For *Ctl3D* to work with radio buttons, check boxes, and any of the common dialog boxes, the CTL3DV2.DLL dynamic-link library must be present on the path.

Example

The following code toggles the 3-D look of a memo control when the user clicks a button named Toggle:

```
procedure TForm1.ToggleClick(Sender: TObject);
begin
  Memo1.Ctl3D := not Memo1.Ctl3D; {Toggles the Ctl3D property of Memo1}
end;
```

See also

ParentCtl3D property

Currency and date/time formatting variables SysUtils

Declaration

```
CurrencyString: string[7];

CurrencyFormat: Byte;

NegCurrFormat: Byte;

ThousandSeparator: Char;

DecimalSeparator: Char;

CurrencyDecimals: Byte;

DateSeparator: Char;

ShortDateFormat: string[15];

LongDateFormat: string[31];

TimeSeparator: Char;

TimeAMString: string[7];
```

```
TimePMString: string[7];

ShortTimeFormat: string[15];

LongTimeFormat: string[31];

ShortMonthNames: array[1..12] of string[3];

LongMonthNames: array[1..12] of string[15];

ShortDayNames: array[1..7] of string[3];

LongDayNames: array[1..7] of string[15];
```

The *SysUtils* unit includes a number of variables that are used by the date and time routines. You can assign new values to these variables to change the formats of date and time strings.

Typed constant	Defines
CurrencyString	The currency symbol used in floating-point to decimal conversions. The initial value is fetched from the sCurrency variable in the [intl] section of WIN.INI.

CurrencyFormat	The currency symbol placement and separation used in floating-point to decimal conversions. Possible values are:
	`0 = '$1'`
	`1 = '1$'`
	`2 = '$ 1'`
	`3 = '1 $'`
	The initial value is fetched from the iCurrency variable in the [intl] section of WIN.INI.

NegCurrFormat	The currency format for used in floating-point to decimal conversions of negative numbers. Possible values are:
	`0 = ($1)    4 = (1$)`
	`1 = -$1     5 = -1$`
	`2 = $-1     6 = 1-$`
	`3 = $1-     7 = 1$-`
	The initial value is fetched from the iNegCurr variable in the [intl] section of WIN.INI.

ThousandSeparator	The character used to separate thousands in numbers with more than three digits to the left of the decimal separator. The initial value is fetched from the sThousand variable in the [intl] section of WIN.INI.
DecimalSeparator	The character used to separate the integer part from the fractional part of a number. The initial value is fetched from the sDecimal variable in the [intl] section of WIN.INI.
CurrencyDecimals	The number of digits to the right of the decimal point in a currency amount. The initial value is fetched from the sCurrDigits variable in the [intl] section of WIN.INI.
DateSeparator	The character used to separate the year, month, and day parts of a date value. The initial value is fetched from the sDate variable in the [intl] section of WIN.INI.

Typed constant	Defines
ShortDateFormat	The format string used to convert a date value to a short string suitable for editing. For a complete description of date and time format strings, refer to the documentation for the FormatDateTime function. The short date format should only use the date separator character and the m, mm, d, dd, yy, and yyyy format specifiers. The initial value is fetched from the sShortDate variable in the [intl] section of WIN.INI.
LongDateFormat	The format string used to convert a date value to a long string suitable for display but not for editing. For a complete description of date and time format strings, refer to the documentation for the FormatDateTime function. The initial value is fetched from the sLongDate variable in the [intl] section of WIN.INI.
TimeSeparator	The character used to separate the hour, minute, and second parts of a time value. The initial value is fetched from the sTime variable in the [intl] section of WIN.INI.
TimeAMString	The suffix string used for time values between 00:00 and 11:59 in 12-hour clock format. The initial value is fetched from the s1159 variable in the [intl] section of WIN.INI.
TimePMString	The suffix string used for time values between 12:00 and 23:59 in 12-hour clock format. The initial value is fetched from the s2359 variable in the [intl] section of WIN.INI.
ShortTimeFormat	The format string used to convert a time value to a short string with only hours and minutes. The default value is computed from the iTime and iTLZero variables in the [intl] section of WIN.INI.
LongTimeFormat	The format string used to convert a time value to a long string with hours, minutes, and seconds. The default value is computed from the iTime and iTLZero variables in the [intl] section of WIN.INI.
ShortMonthNames	Array of strings containing short month names. The mmm format specifier in a format string passed to FormatDateTime causes a short month name to be substituted.
LongMonthNames	Array of strings containing long month names. The mmmm format specifier in a format string passed to FormatDateTime causes a long month name to be substituted.
ShortDayNames	Array of strings containing short day names. The ddd format specifier in a format string passed to FormatDateTime causes a short day name to be substituted.
LongDayNames	Array of strings containing long day names. The dddd format specifier in a format string passed to FormatDateTime causes a long day name to be substituted.

Example

This example uses a label and a button on a form. When the user clicks the button, the current date displays in the caption of the label. Because some of the date variables are assigned new values, the format of the date in the label changes. For example, if the date is 9/15/94, the date displays as 15-09-1994.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  DateOrder := doDMY;
  DateSeparator := '-';
  DateFullYear := True;
  DateLeadZero := True;
  Label1.Caption := DateToStr(Date);
end;
```

Currency property

Applies to

TBCDField, *TCurrencyField*, *TFloatField* components

Declaration

```
property Currency: Boolean;
```

Run-time only. The *Currency* property is used to control the format of the value of a *TBCDField*, *TCurrencyField*, and *TFloatField* when both *DisplayFormat* and *EditFormat* properties are not assigned.

Currency is *True* by default for *TCurrencyField* and *False* for *TFloatField* and *TBCDField*. When *Currency* is *True* formatting is performed by *FloatToText* using *ffCurrency* for display text or *ffFixed* for editable text. When *Currency* is *False*, the formatting is performed by *FloatToTextFmt*.

See also

DisplayFormat property, *EditFormat* property, *FloatToText* function, *FloatToTextFmt* function

Cursor property

Applies to

All controls, *TScreen* component

For all controls

Declaration

```
property Cursor: TCursor;
```

The *Cursor* property is the image used when the mouse passes into the region covered by the control. These are the possible images:

Value	Image	Value	Image	Value	Image
crDefault	↖	*crSizeNESW*	⤢	*crHourglass*	⧗
crArrow	↖	*crSizeNS*	↕	crDrag	↖
crCross	✚	*crSizeNWSE*	⤡	*crNoDrop*	⊘

Value	Image	Value	Image	Value	Image
crIBeam	I	*crSizeWE*	⇔	*crHSplit*	⇹
crSize	✥	*crUpArrow*	⇧	*crVSplit*	⇳

To learn how to make a custom cursor available to your application, see the *Cursors* property.

Example

This line of code changes the display of the image to the cross cursor when the user moves the mouse pointer over *Button1:*

```
Button1.Cursor := crCross;
```

See also

Cursors property, *DragCursor* property

For screen objects

Declaration

```
property Cursor: TCursor;
```

The *Screen* object's *Cursor* property controls the mouse cursor shape at a global level. Assigning any value but *crDefault* to the *Screen* object's *Cursor* property sets the mouse cursor shape for all windows belonging to the application. The global mouse cursor shape remains in effect until you assign *crDefault* to the *Screen* object's *Cursor* property, at which point normal cursor behavior is restored.

To see a list of possible cursor shapes, see the *Cursor* property for all controls.

Example

Assignments to the *Screen* object's cursor property are typically guarded by a **try...finally** statement to ensure that normal cursor behavior is restored, for example:

```
Screen.Cursor := crHourglass;                { Show hourglass cursor }
try
  { Do some lengthy operation }
finally
  Screen.Cursor := crDefault;                 { Always restore to normal }
end;
```

Cursor typed constant **WinCrt**

Declaration

```
const Cursor: TPoint = (X: 0; Y: 0);
```

The *Cursor* variable contains the current position of the cursor within the virtual screen.

The upper left corner corresponds to (0, 0). *Cursor* is a read-only variable; do not assign values to it.

CursorPosChanged method

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure CursorPosChanged;
```

The *CursorPosChanged* method is needed only if you use the *Handle* property to make direct calls to the Borland Database Engine (BDE) API which cause the cursor position to change. To notify the dataset that the underlying BDE cursor's position has changed, call *CursorPosChanged* after the direct calls to the BDE.

See also
UpdateCursorPos method

Cursors property

Applies to
TScreen component

Declaration

```
property Cursors[Index: Integer]: HCursor;
```

Run-time only. The *Cursors* property gives you access to the list of cursors available for your application. To access a particular cursor, specify its position in the list of cursors as the value of the *Index* parameter with the first position in the list having an index of 0, the second having an index of 1, and so on.

Using the *Cursors* property, you can make custom cursors available to your application.

These are the cursor constants and their position in the *Cursors* property array:

Cursor	Value	Cursor	Value
crDefault	0	*crSizeWE*	–9
crNone	–1	*crUpArrow*	–10
crArrow	–2	*crHourglass*	–11
crCross	–3	*crDrag*	–12
crIBeam	–4	*crNoDrop*	–13
crSize	–5	*crHSplit*	–14
crSizeNESW	–6	*crVSplit*	–15
crSizeNS	–7	*crMultiDrag*	–16
crSizeNWSE	–8	*crSQLWait*	–17

To make a custom cursor available to your application,

1 Create the cursor resource using a resource editor.

2 Declare a cursor constant with a value that does not conflict with an existing cursor constant.

3 Use the WinAPI function *LoadCursor* to make your cursor available to your application, specifying the newly declared cursor constant as the value of the *Index* parameter for the *Cursors* property array.

Note You don’t need to call the WinAPI function *DestroyCursor* when you are finished using the custom cursor; Delphi does this automatically.

Example

This example assumes you have created a cursor resource with the name *NewCursor*. The code loads the new cursor into the *Cursors* property array and makes the newly loaded cursor the cursor of the form:

```
const
  crMyCursor = 5;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Screen.Cursors[crMyCursor] := LoadCursor(HInstance, 'NewCursor');
  Cursor := crMyCursor;
end;
```

See also

Cursor property, *DragCursor* property

CursorTo procedure **WinCrt**

Declaration

```
procedure CursorTo(X, Y: Integer);
```

The *CursorTo* procedure moves the cursor to the given coordinates (X, Y) within the virtual screen.

The coordinates of the upper left corner of the CRT window are (0, 0). *CursorTo* sets the *Cursor* variable to (X, Y).

See also

GoToXY procedure

CustomColors property

Applies to

TColorDialog component

Declaration

```
property CustomColors: TStrings;
```

The value of the *CustomColors* property determines the custom colors that are available in the Color dialog box. Each custom color is represented as a string that follows this format:

```
ColorX=HexValue
```

For example, this string could indicate that the first custom color box in the Color dialog box:

```
ColorA=808022
```

This is the same format that your CONTROL.INI file uses to specify the custom colors that are available in the Windows Color dialog box.

You can have up to 16 custom colors, *ColorA* through *ColorP*.

Use the string list of custom colors to save the custom colors specified in the dialog box so you can use them elsewhere. For example, you might save them to an .INI file for your application so your application can use the custom colors.

Example

This example displays the Color dialog box, allowing the user to create custom colors, then displays the custom color strings in a list box:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if ColorDialog1.Execute then
    ListBox1.Items.AddStrings(ColorDialog1.CustomColors);
end;
```

CutToClipboard method

Applies to

TDBEdit, *TDBImage*, *TDBMemo*, *TEdit*, *TMaskEdit*, *TMemo* components

Declaration

```
procedure CutToClipboard;
```

The *CutToClipboard* method deletes the text selected in the control and copies it to the Clipboard, replacing any text that exists there. If no text is selected, nothing is copied.

For database images, *CutToClipboard* deletes the image in the control and copies it to the Clipboard, replacing the contents of the Clipboard.

Example

The following method cuts the text the user selects in *Memo1* to the Clipboard and pastes it from the Clipboard in an edit box control when the user clicks the button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.CutToClipboard;
  Edit1.PasteFromClipboard;
end;
```

See also

Clear method, *ClearSelection* method, *CopyToClipboard* method, *PasteFromClipboard* method

Data property

Applies to

TOutlineNode component

Declaration

```
property Data: Pointer;
```

Run-time only. The *Data* property specifies any data you want associated with an outline item.

Example

The following code creates a *TBitmap* and adds it to the *Data* of the selected outline item.

```
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create;
```

```
    Outline1.Items[Outline1.SelectedItem].Data := Bitmap;
  end;
```

See also

GetDataItem method, *Text* property

Database property

Applies to

TTable, *TQuery* component

Declaration

```
property Database: TDatabase;
```

Run-time and read only. *Database* specifies the database (*TDatabase*) component associated with the dataset component. If you did not create a *TDatabase* at design time, then Delphi will create one at run time. Use the *Database* property to reference the properties and methods of the database.

Example

```
  { Do a transaction }
  with Table1.Database do
    begin
    StartTransAction;
  { Post some records with Table1 }
    Commit;
    end;
```

DatabaseCount property

Applies to

TSession component

Declaration

```
property DatabaseCount: Integer;
```

Run-time and read only. *DatabaseCount* is the number of *TDataBase* components currently attached to *Session*.

Example

```
  { Close all databases }
  with Session do
    while DatabaseCount <> 0 do
      Databases[0].Close;
```

See also

Databases property

DatabaseError procedure **DB**

Declaration

```
procedure DatabaseError(const Message: string);
```

The *DatabaseError* procedure creates and raises the *EDatabaseError* exception object, using *Message* as the text for the exception.

Example

```
   { Test for an error and raise an exception if so }
   if { some error has occured } then DatabaseError('Some error has occured');
```

DatabaseName property

Applies to

TDataBase, *TQuery*, *TStoredProc*, *TTable* components

For database components

Declaration

```
property DatabaseName: TFileName;
```

Set the *DatabaseName* property to define an application-specific alias. Dataset components can reference this name instead of a BDE alias, directory path, or database name. In other words, this is the name of an application-specific alias defined by the dataset component that will show up in the *DatabaseName* drop-down list of *TTable*, *TQuery*, and *TStoredProc* components.

If you try to set *DatabaseName* of a *TDatabase* for which *Connected* is *True*, Delphi will raise an exception.

Example

```
   Database1.DatabaseName := 'Delphi_Demos';
```

For tables, queries, and stored procedures

Declaration

```
property DatabaseName: TFileName;
```

Set the *DatabaseName* property to specify the database to access. This property can specify:

- A defined BDE alias,
- A directory path for desktop database files,
- A directory path and file name for a Local InterBase Server database,
- An application-specific alias defined by a *TDatabase* component

Note Use the *Close* method to put a dataset in Inactive state before changing *DatabaseName.*

Example

```
{ Close the DBDataSet }
Table1.Active := False;
try
{ First try to use an alias }
  Table1.DatabaseName := 'Delphi_Demos';
  Table1.Active := True;
except
  on EDatabaseError do
{ If that fails, try to use the drive and directory }
    Table1.DatabaseName := 'c:\delphi\demos\database';
    Table1.Active := True;
```

See also

Active property

Databases property

Applies to

TSession component

Declaration

```
property Databases[Index: Integer]: TDatabase;
```

Run-time and read only. The *Databases* property holds a list of all of the currently active *TDatabase* components.

Example

```
{ Close all databases }
with Session do
  while DatabaseCount <> 0 do
    Databases[0].Close;
```

See also

DatabaseCount property

DataField property

Applies to

TDBCheckBox, *TDBComboBox*, *TDBEdit*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBLookupList*, *TDBMemo*, *TDBRadioGroup*, *TDBText* components

Declaration

```
property DataField: string;
```

The *DataField* property identifies the field from which the data-aware control displays data. The dataset the field is located in is specified in a data source component (*TDataSource*). The *DataSource* property of the data-aware control specifies which data source component.

If the *DataField* value of a database edit box (*TDBEdit*) is an integer or floating-point value, only characters that are valid in such a field can be entered in the edit box. Characters that are not legal are not accepted.

Example

The following code specifies that the *DataField* of *DBEdit1* is 'FNAME'.

```
  DBEdit1.DataField := 'FNAME';
```

DataFormat property

Applies to

TOLEDropNotify object

Declaration

```
property DataFormat: Word;
```

The *DataFormat* property specifies the Clipboard format of data dropped on a form. The form must be registered with the *RegisterFormAsOLEDropTarget* function for a *TOLEDropNotify* object to be the *Source* in an *OnDragDrop* event handler. If *DataFormat* specifies an OLE object format, the *PInitInfo* property points to initialization information for the dropped OLE object. If the dropped data is not an OLE object, *DataFormat* specifies some other format (such as *CF_BITMAP* for bitmap graphic data) and *PInitInfo* won't point to valid OLE initialization information and can't be used to initialize a *TOLEContainer* component.

Example

The following code is the *OnDragDrop* event handler for a form that is registered as an OLE drop target with *RegisterFormAsOLEDropTarget*. If a text object is dropped, a label is created to display the data. If a metafile object is dropped, an image is created to display the data. Otherwise, it is assumed that an OLE object was dropped and an OLE container is created to contain the object.

```
procedure TXMdiX.DoDrop(DragTgt, DragSource: TObject; X, Y: Integer);
var
  Ctrl  : TOleContainer;
  Image : TImage;
  Pict : TPicture;
  ClipPict : TPicture;
  FLabel : TLabel;
  Ptr  : PChar;
  Str  : String;
  Dropper : TOleDropNotify;
begin
  if DragSource is TOleDropNotify then
  begin
    Dropper := TOleDropNotify (DragSource);
    if Dropper.DataFormat = CF_TEXT then
    begin
      FLabel := TLabel.Create (TForm(DragTgt));
      FLabel.Left := X;
      FLabel.Top := Y;
      FLabel.Width := 30;
      FLabel.Height := 10;
      Ptr := GlobalLock (Dropper.DataHandle);
      Str := StrPas (Ptr);
      GlobalUnlock (Dropper.DataHandle);
      Str := Format('DropText = %s', [@Str]);
      FLabel.Caption := Str;
      GlobalFree (Dropper.DataHandle);
      FLabel.visible := True;
      FLabel.enabled := True;
      TForm (DragTgt).InsertControl (FLabel);
    end
    else if Dropper.DataFormat = CF_METAFILEPICT then
    begin
      Image := TImage.Create (TForm(DragTgt));
      Image.Left := X;
      Image.Top := Y;
      Image.Width := 30;
      FLabel.Height := 10;
      Pict := TPicture.Create;
      Pict.LoadFromClipboardFormat(Dropper.DataFormat, Dropper.DataFormat, 0);
      Image.Picture := Pict;
      GlobalFree (Dropper.DataHandle);
      Image.visible := True;
      Image.enabled := True;
      TForm (DragTgt).InsertControl (Image);
    end;
    else if Dropper.PInitInfo <> Nil then
    begin
      Ctrl := TOleContainer.Create (TForm(DragTgt));
      Ctrl.top     := Y;
      Ctrl.left    := X;
      Ctrl.Width   := 100;
```

```
        Ctrl.Height  := 100;
        Ctrl.visible := True;
        Ctrl.enabled := True;
        Ctrl.AutoSize := True;
        TForm (DragTgt).InsertControl (Ctrl);
        Ctrl.PInitInfo := Dropper.PInitInfo;
      end;
    end;
  end;
```

See also
DataHandle property

DataHandle property

Applies to
TOLEDropNotify object

Declaration

```
property DataHandle: THandle;
```

The *DataHandle* property specifies a handle to the data dropped on a form. The form must have been registered with the *RegisterFormAsOLEDropTarget* function for a *TOLEDropNotify* object to be the *Source* in an *OnDragDrop* event handler. If the data is any type other than an OLE object, you can use *DataHandle* to access the data.

Example
The following code locks the data handle of a *TOLEDropNotify* object named *Dropper*.

```
  Ptr := GlobalLock (Dropper.DataHandle);
```

See also
DataFormat property

DataSet property

Applies to
TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDataSource*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

For data source components

Declaration

```
property DataSet: TDataSet
```

DataSet specifies the dataset component (*TTable*, *TQuery*, and *TStoredProc*) that is providing data to the data source. Usually you set *DataSet* at design time with the Object Inspector, but you can also set it programmatically. The advantage of this interface approach to connecting data components is that the dataset, data source, and data-aware controls can be connected and disconnected from each other through the *TDataSource* component. In addition, these components can belong to different forms.

Example

```
DataSource1.DataSet := Table1; {get data from this form's Table1}
DataSource1.DataSet := Form2.Table1; {get data from Form2's Table1}
```

For field components

Declaration

```
property DataSet: TDataSet;
```

Run-time only. *DataSet* identifies the dataset to which a *TField* component belongs. Only assign a value to this property if you are programmatically creating *TField* component .

DatasetCount property

Applies to
TDataBase component

Declaration

```
property DatasetCount: Integer;
```

DatasetCount is the number of dataset components (*TTable*, *TQuery*, and *TStoredProc*) that are currently using the *TDatabase* component. Read-only and run time only.

Example

```
{ Check to see if any record associated with this database has pending updates }
Changed := False;
with Database1 do
  for I := 0 to DatasetCount - 1 do
    Changed := Changed or DataSets[I].Modified;
```

See also
Datasets property

Datasets property

Applies to

TDataBase component

Declaration

```
property Datasets[Index: Integer]: TDBDataSet;
```

Run-time and read only. *Datasets* is the set of dataset components that are currently sharing the *TDatabase* component.

Example

```
{ Check to see if any record associated with this database has pending updates }
Changed := False;
with Database1 do
  for I := 0 to DatasetCount - 1 do
    Changed := Changed or DataSets[I].Modified;
```

See also

DatasetCount property

DataSize property

Applies to

TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
property DataSize: Word;
```

Run-time and read only. The value of *DataSize* is the number of bytes required to store the field in memory.

For *TBoolean*, *TSmallint,* and *TWordField*, the value is two bytes. For *TDateField, TIntegerField*, and *TTimeField*, the value is four bytes. For *TCurrencyField*, *TDateTimeField*, and *TFloatField*, the value is eight bytes. For *TBCDField*, the value is eighteen bytes. For *TStringField*, the value is the maximum size of the text plus one (not more than 255 characters). For *TBlobField*, *TBytesField*, *TGraphicField*, *TMemoField*, and *TVarBytesField*, the value is the size of the field as stored in the record buffer.

DataSource property

Applies to
TDBCheckBox, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBLookupList*, *TDBMemo*, *TDBNavigator*, *TQuery*, *TDBRadioGroup*, *TDBText* components

For data-aware controls

Declaration

```
property DataSource: TDataSource;
```

The *DataSource* property determines where the component obtains the data to display. Specify the data source component that identifies the dataset the data is found in.

Example
The following code specifies *DataSource1* to be the *DataSource* of *DBGrid1*.

```
DBGrid1.DataSource := DataSource1;
```

See also
DataField property, *SQL* property

For queries

Declaration

```
property DataSource: TDataSource;
```

Set the *DataSource* property to the name of a *TDataSource* component in the application to assign values to parameters not bound to values programmatically with *Params* or *ParamByName*. If the unbound parameter names match any column names in the specified data source, Delphi binds the current values of those fields to the corresponding parameters. This capability enables applications to have linked queries.

Example
The LINKQRY sample application illustrates the use of the *DataSource* property to link a query in a master-detail form. The form contains a *TQuery* component (named *Orders*) with the following in its *SQL* property:

```
SELECT Orders.CustNo, Orders.OrderNo, Orders.SaleDate
   FROM Orders
   WHERE Orders.CustNo = :CustNo
```

The form also contains:

- A *TDataSource* named *OrdersSource*, linked to *Orders* by its *DataSet* property.
- A *TTable* component (named *Cust*).

- A *TDataSource* named *CustSource* linked to *Cust*.
- Two data grids; one linked to *CustSource* and the other to *OrdersSource*.

Orders' *DataSource* property is set to *CustSource.* Because the parameter :*CustNo* does not have any value assigned to it, at run time Delphi will try to match it with a column name in *CustSource*, which gets its data from the Customer table through *Cust*. Because there is a CustNo column in *Cust*, the current value of CustNo in the *Cust* table is assigned to the parameter, and the two data grids are linked in a master-detail relationship. Each time the *Cust* table moves to a different row, the *Orders* query automatically re-executes to retrieve all the orders for the current customer.

See also

SQL property

DataType property

Applies to

TFieldDef, *TParam* objects; *TBCDField*, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

For field definition objects

Declaration

```
property DataType: TFieldType;
```

Run-time and read only. Read *DataType* to determine a physical field's type. Possible values are those of the *TFieldType* type: *ftUnknown*, *ftString*, *ftSmallint*, *ftInteger*, *ftWord*, *ftBoolean*, *ftFloat*, *ftCurrency*, *ftBCD*, *ftDate*, *ftTime*, *ftDateTime*, *ftBytes*, *ftVarBytes*, *ftBlob*, *ftMemo* or *ftGraphic*.

For field definitions

Declaration

```
property DataType: TFieldType;
```

Run-time and read only. *DataType* identifies the data type of the *TField*. Possible values are those of the *TFieldType* type: *ftBoolean*, *ftBCD*, *ftBlob*, *ftBytes*, *ftCurrency*, *ftDate*, *ftDateTime*, *ftFloat*, *ftGraphic*, *ftInteger*, *ftMemo*, *ftSmallint*, *ftString*, *ftTime*, *ftUnknown*, *ftVarBytes*, and *ftWord*.

For TParam objects

Declaration

```
property DataType: TFieldType;
```

The *DataType* property is the type of the parameter. Possible values are those of the *TFieldType* type: *ftUnknown*, *ftString*, *ftSmallint*, *ftInteger*, *ftWord*, *ftBoolean*, *ftFloat*, *ftCurrency*, *ftBCD*, *ftDate*, *ftTime*, *ftDateTime*, *ftBytes*, *ftVarBytes*, *ftBlob*, *ftMemo* or *ftGraphic*.

Example

```
with Query1.Parameters do
  for I := 0 to Count - 1 do
    if Params[I].DataType = ftUnknown then
      MessageDlg('Parameter ' + IntToStr(I) + ' is undefined', mtWarning, [mbOK], 0);
```

Date function   SysUtils

Declaration

```
function Date: TDateTime;
```

The *Date* function returns the current date.

Example

This example uses a label and a button on a form. When the user clicks the button, the current date is displayed in the caption of the label:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := 'Today is  ' + DateToStr(Date);
end;
```

See also

DateToStr function, *DayOfWeek* function, *DecodeDate* procedure, *Now* function, *Time* function

DateTimeToFileDate function   SysUtils

Declaration

```
function DateTimeToFileDate(DateTime: TDateTime): Longint;
```

DateTimeToFileDate converts a *TDateTime* value to a DOS date-and-time value. The *FileAge*, *FileGetDate*, and *FileSetDate* routines operate on DOS date-and-time values, and the Time field of a *TSearchRec* used by the *FindFirst* and *FindNext* functions contains a DOS date-and-time value.

See also

FileDateToDateTime function

DateTimeToString procedure SysUtils

Declaration

```
procedure DateTimeToString(var Result: string; const Format: string; DateTime: TDateTime);
```

DateTimeToString converts the date and time value given by *DateTime* using the format string given by *Format* into the string variable given by *Result*. For further details, see the description of the *FormatDateTime* function.

See also

TDateTime type

DateToStr function SysUtils

Declaration

```
function DateToStr(Date: TDateTime): string;
```

The *DateToStr* function converts a variable of type *TDateTime* to a string. The conversion uses the format specified by the *ShortDateFormat* global variable.

Example

This example uses a label and a button on a form. When the user clicks the button, the current date is converted to a string and displayed as the caption of the label:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := DateToStr(Date);
end;
```

See also

Date function, *DateTimeToStr* function, *StrToDate* function, *TimeToStr* function

DateTimeToStr function SysUtils

Declaration

```
function DateTimeToStr(DateTime: TDateTime): string;
```

The *DateTimeToStr* function converts a variable of type *TDateTime* to a string. If *DateTime* parameter does not contain a date value, the date displays as 00/00/00. If the *DateTime* parameter does not contain a time value, the time displays as 00:00:00 AM. You can change how the string is formatted by changing some of the date and time typed constants.

Example

This example uses a label and a button on a form. When the user clicks the button, the current date and time is converted to a string and displayed as the caption of the label:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := DateTimeToStr(Now);
end;
```

See also

Date function, *DateToStr* function, *Now* function, *StrToDate* function, *Time* function, *TimeToStr* function

DayOfWeek function SysUtils

Declaration

```
function DayOfWeek(Date: TDateTime): Integer;
```

The *DayOfWeek* function returns the day of the week of the specified date as an integer between 1 and 7. Sunday is the first day of the week and Saturday is the seventh.

Example

This example uses a button, an edit box, and a label on a form. When the user enters a date in the edit box using the Month/Day/Year format, the caption of the label reports the day of the week for the specified date.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ADate: TDateTime;
begin
  ADate := StrToDate(Edit1.Text);
  Label1.Caption := 'Day ' + IntToStr(DayOfWeek(ADate)) + ' of the week';
end;
```

See also

Date function, *EncodeDate* function, *Now* function, *StrToDate* function, *StrToDateTime* function

DBHandle property

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
property DBHandle: HDBIDB;
```

Run-time and read only. The *DBHandle* property enables an application to make direct calls to the Borland Database Engine (BDE) API. Many BDE function calls require a database handle. This property provides the requisite database handle.

Under most circumstances you should not need to use this property, unless your application requires some functionality not encapsulated in the VCL.

DbiError procedure **DB**

Declaration

```
procedure DbiError(ErrorCode: Integer);
```

The *DbiError* procedure creates an error message by querying the Borland Database Engine for the last error number and text and calls *DatabaseError* passing the result. *ErrorCode* is used to obtain a text message from the engine if the error has already been cleared.

DBLocale property

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
property DBLocale: TLocale;
```

Run-time and read only. The *DBLocale* property allows you to make direct calls to the Borland Database Engine using this specification of the language driver. Under most circumstances you should not need to use this property, unless your application requires some functionality not encapsulated in the VCL.

DDEConv property

Applies to
TDDEClientItem component

Declaration

```
property DdeConv: TDdeClientConv;
```

The *DDEConv* property specifies the DDE client conversation component to associate with the DDE client item component. The value of *DDEConv* is the name of the DDE client conversation component that defines the DDE conversation.

Example

The following code specifies *DDEClientConv1* as the conversation of *DDEClientItem1*.

```
DDEClientItem1.DDEConv := DDEClientConv1.
```

See also

Name property

DDEItem property

Applies to

TDDEClientItem component

Declaration

```
property DDEItem: String;
```

The *DDEItem* property specifies the item of a DDE conversation. The value of *DDEItem* depends on the linked DDE server application. *DDEItem* is typically a selectable portion of text, such as a spreadsheet cell or a database field in an edit box. If the DDE server is a Delphi application, *DDEItem* is the name of the linked DDE server component. For example, to link to a DDE server component named *DDEServer1*, set *DDEItem* to 'DDEServer1'.

See the documentation for the DDE server application for the specific information about specifying *DDEItem*.

At design time, you can specify *DDEItem either* by typing the item string in the object inspector or by pasting a link using the DDE Info dialog box, which appears if you click the ellipsis (...) button for *DDEService* or *DDETopic* in the Object Inspector. After you choose Paste Link in the DDE Info dialog box, you can choose the item from a list of possible items for *DDEItem* in the object inspector if link information is still on the Clipboard.

Example

The following code specifies a DDE item of 'DDEServer1'.

```
DDEClientItem1.DDEItem := 'DDEServer1';
```

See also

DDEService property, *DDETopic* property

DDEService property

Applies to

TDDEClientConv component

Declaration

```
property DDEService: string;
```

The *DDEService* property specifies the DDE server application to be linked to a DDE client. Typically, *DDEService* is the file name (and path, if necessary) of the DDE server application's main executable file without the .EXE extension. If the DDE server is an Delphi application, *DDEService* is the project name without the .DPR or .EXE extension. For example, to link to a *TDDEServerConv* component in PROJ1.DPR, set *DDEService* to 'PROJ1'.

See the documentation for the DDE server application for the specific information about specifying *DDEService*.

At design time, you can specify *DDEService either* by typing the DDE server application name in the object inspector or by choosing Paste Link in the DDE Info dialog box.

Example

The following code specifies a DDE service of 'Project1'.

```
DDEClientConv1.DDEService := 'Project1';
```

See also

DDEItem property, *DDETopic* property

DDETopic property

Applies to

TDDEClientConv component

Declaration

```
property DDETopic: string;
```

The *DDETopic* property specifies the topic of a DDE conversation. Typically, *DDETopic* is a file name (and path, if necessary) used by the application specified in *DDEService*. If the DDE server is an Delphi application, by default *DDETopic* is the caption of the form containing the linked component. For example, to link to a component on a form named *Form1*, set *DDETopic* to 'Form1'. However, if the DDE client is linked to a *TDDEServerConv* component, *DDETopic* is the name of the server conversation component instead of the form caption. For example, to link to *DDEServerConv1*, set *DDETopic* to 'DDEServerConv1'.

See the documentation for the DDE server application for the specific information about specifying *DDETopic*.

At design time, you can specify *DDETopic either* by typing the DDE server application name in the object inspector or by choosing Paste Link in the DDE Info dialog box.

Example

The following code spGecifies a DDE topic of 'Form1'.

```
   DDEClientConv1.DDETopic := 'Form1';
```

See also

DDEItem property

Dec procedure

System

Declaration

```
procedure Dec(var X[ ; N: Longint]);
```

The *Dec* procedure subtracts one or *N* from a variable.

Dec(*X*) corresponds to X := X – 1, and *Dec*(*X*, *N*) corresponds to X := X – N.

X is an ordinal-type variable or a variable of type *PChar* if the extended syntax is enabled, and *N* is an integer-type expression.

Dec generates optimized code and is especially useful in a tight loop.

Example

```
    var
     IntVar: Integer;
     LongintVar: Longint;
   begin
     Intvar := 10;
     LongintVar := 10;
     Dec(IntVar);                                                { IntVar := IntVar - 1 }
     Dec(LongintVar, 5);                                 { LongintVar := LongintVar - 5 }
   end;
```

See also

Inc procedure, *Pred* function, *Succ* function

DecodeDate procedure

SysUtils

Declaration

```
procedure DecodeDate(Date: TDateTime; var Year, Month, Day: Word);
```

The *DecodeDate* procedure breaks the value specified as the *Date* parameter into *Year*, *Month*, and *Day* values. If the given *TDateTime* value is less than or equal to zero, the year, month, and day return parameters are all set to zero.

Example

This example uses a button and two labels on a form. When the user clicks the button, the current date and time are reported in the captions of the two labels.

```
   procedure TForm1.Button1Click(Sender: TObject);
```

```
var
  Present: TDateTime;
  Year, Month, Day, Hour, Min, Sec, MSec: Word;
 begin
  Present:= Now;
  DecodeDate(Present, Year, Month, Day);
  Label1.Caption := 'Today is Day ' + IntToStr(Day) + ' of Month '
    + IntToStr(Month) + ' of Year ' + IntToStr(Year);
  DecodeTime(Present, Hour, Min, Sec, MSec);
  Label2.Caption := 'The time is Minute ' + IntToStr(Min) + ' of Hour '
    + IntToStr(Hour);
end;
```

See also

DecodeTime procedure

DecodeTime procedure — SysUtils

Declaration

```
procedure DecodeTime(Time: TDateTime; var Hour, Min, Sec, MSec: Word);
```

The *DecodeTime* procedure breaks the value specified as the *Time* parameter into hours, minutes, seconds, and milliseconds.

Example

This example uses a button and two labels on a form. When the user clicks the button, the current date and time are reported in the captions of the two labels.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Present: TDateTime;
  Year, Month, Day, Hour, Min, Sec, MSec: Word;
 begin
  Present:= Now;
  DecodeDate(Present, Year, Month, Day);
  Label1.Caption := 'Today is Day ' + IntToStr(Day) + ' of Month '
    + IntToStr(Month) + ' of Year ' + IntToStr(Year);
  DecodeTime(Present, Hour, Min, Sec, MSec);
  Label2.Caption := 'The time is Minute ' + IntToStr(Min) + ' of Hour '
    + IntToStr(Hour);
end;
```

See also

DecodeDate procedure, *EncodeTime* function, *Time* function

Default property

Applies to
TBitBtn, *TButton* components

Declaration

```
property Default: Boolean;
```

The *Default* property indicates whether a push or bitmap button is the default button. If *Default* is *True*, any time the user presses Enter, the *OnClick* event handler for that button runs. The only exception to this is if the user selects another button before pressing Enter, in which case the *OnClick* event handler for that button runs. Although your application can have more than one button designated as a default button, the form calls the *OnClick* event handler for the first button in the tab order.

Whenever any button has focus, it becomes the default button temporarily. When the focus moves to a control that isn't a button, the button with its *Default* property set to *True* becomes the default button once again.

Example
This example makes the button named *OK* the default button:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  OK.Default := True;
end;
```

See also
Cancel property

DefaultColWidth property

Applies to
TDrawGrid, *TStringGrid* components

Declaration

```
property DefaultColWidth: Integer;
```

The *DefaultColWidth* property determines the width of all the columns within the grid.

If you want to change the width of a single column within a grid without changing other columns, use the *ColWidths* property during run time. If you change the *DefaultColWidth* property value after changing the width of specified columns, all the columns become the height specified in the *DefaultColWidth* property once again.

The default value is 64 pixels.

Example

The following line of code changes the default width of the columns in a string grid to twice the original value:

```
   StringGrid1.DefaultColWidth := StringGrid1.DefaultColWidth * 2;
```

See also

ColWidth property, *DefaultRowHeight* property

DefaultDrawing property

Applies to

TDBGrid, *TDrawGrid*, *TStringGrid* components

Declaration

```
property DefaultDrawing: Boolean;
```

The *DefaultDrawing* property determines if the cell is painted and the item it contains is drawn automatically. If *True*, the default drawing occurs. If *False*, your application must handle all the drawing details in the *OnDrawCell* event handler, or in the *OnDrawDataCell* event handler for the data grid.

When *DefaultDrawing* is *True*, the *Paint* method initializes the *Canvas*' font and brush to the control font and the cell color. The cell is prepainted in the cell color and a focused *TRect* object is drawn in the cell. The state of the cell is returned. The possible states are a fixed cell, a focused cell, or a cell within the area the user has selected.

Example

The following code sets *DefaultDrawing* to *False* for *DrawGrid1*.

```
   DrawGrid1.DefaultDrawing := False;
```

See also

OnDrawCell event, *OnDrawDataCell* event

DefaultExt property

Applies to

TOpenDialog, *TSaveDialog* components

Declaration

```
property DefaultExt: TFileExt;
```

The *DefaultExt* property specifies the extension that is added to the file name the user types in the File Name edit box if the user doesn't include a file-name extension in the file name. If the user specifies an extension for the file name, the value of the *DefaultExt* property is ignored. If the *DefaultExt* value remains blank, no extension is added to the file name entered in the File Name edit box.

Legal property values include strings up to 3 characters in length. Don't include the period (.) that divides the file name and its extension.

Example

This example sets the default file extension to TXT, displays the Open dialog box, then assigns the file name the user selects with the dialog box to a variable the application can use to open a file:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NameOfFile : TFileName;
begin
  OpenDialog1.DefaultExt := 'TXT';
  if OpenDialog1.Execute then
    NameOfFile := OpenDialog1.FileName;
end;
```

When this code runs, if the user types a file name in the File Name edit box in the Open dialog box, but doesn't specify an extension, the TXT extension is added to the file name, and the entire file name is saved in the *NameOfFile* variable. For example, if the user types MYNOTES as the file name, the string saved in the *NameOfFile* variable is MYNOTES.TXT.

See also

FileName property, *TOpenDialog* component, *TSaveDialog* component

DefaultRowHeight property

Applies to

TDrawGrid, *TStringGrid* components

Declaration

```
property DefaultRowHeight: Integer;
```

The *DefaultRowHeight* property determines the height of all the rows within the grid. The default value is 24 pixels.

If you want to change the height of a single row within a grid without changing other rows, use the *RowHeights* property during run time. If you change the *DefaultRowHeight* property value after changing the height of specified rows, all the rows become the height specified in the *DefaultRowHeight* property once again.

Example

The following line of code changes the default height of the rows in a string grid control to 10 pixels more than the original value:

```
StringGrid1.DefaultRowHeight := StringGrid1.DefaultRowHeight + 10;
```

See also

DefaultColWidth property, *RowHeights* property

Delete method

Applies to

TList, *TStringList*, *TStrings* objects; *TMenuItem*, *TOutline*, *TQuery*, *TTable* components

For list and string objects and menu items

Declaration

```
procedure Delete(Index: Integer);
```

The *Delete* method removes the item specified with the *Index* parameter. The item can be deleted from

- the list of a list object
- the strings and their associated objects of a string or string list object
- a menu

In all cases, the index is zero-based, so the first item has an *Index* value of 0, the second item has an *Index* value of 1, and so on.

If a string is deleted from a string object, the reference to its associated object is also deleted.

If the item deleted is a menu item that has a submenus, the submenus are also deleted.

If the item is deleted in a list object, the list contains a **nil** value in the item's position in the list.

Example

FileMenu in the following code is a menu that contains four menu items (menu commands). They are New, Open, Save, and Save As, in that order. This event handler deletes the Save command from the menu:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  FileMenu.Delete(2);
end;
```

This example uses a list box and a button on a form. When the form appears, five items are in the list box. When the user clicks the button, the second item in the list box is deleted:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
```

```
begin
  for I := 1 to 5 do
    ListBox1.Items.Add('Item ' + IntToStr(I));
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Delete(1);
end;
```

See also

Add method, *Clear* method, *Insert* method, *Remove* method

For outlines

Declaration

```
procedure Delete(Index: LongInt);
```

The *Delete* method removes the outline item with an *Index* property value equal to the *Index* parameter from the list outline. If that item is has subitems, the subitems are also deleted.

Outline items that appear after the deleted item are moved up and reindexed with valid *Index* values. This is done automatically unless *BeginUpdate* has been called.

Example

The following code deletes the selected item from *Outline1*.

```
Outline1.Delete(Outline1.SelectedItem);
```

See also

Add method, *AddChild* method, *Insert* method

For queries and tables

Declaration

```
procedure Delete;
```

The *Delete* method deletes the current record from the *dataset*. The next record then becomes the new current record. If the record deleted was the last record in the dataset, then the previous record becomes the current record.

This method is valid only for datasets that return a live result set.

Delete procedure — System

Declaration

```
procedure Delete(var S: string; Index, Count:Integer);
```

The *Delete* procedure removes a substring of *Count* characters from string *S* starting at *S*[*Index*].

S is a string-type variable. *Index* and *Count* are integer-type expressions.

If *Index* is larger than the length of *S*, no characters are deleted. If *Count* specifies more characters than remain starting at the *S*[*Index*], *Delete* removes the rest of the string.

Example

```
var
  s: string;
begin
  s := 'Honest Abe Lincoln';
  Delete(s,8,4);
  Canvas.TextOut(10, 10, s); { 'Honest Lincoln' }
end;
```

See also

Concat function, *Copy* function, *Insert* procedure, *Length* function, *Pos* function

DeleteFile function — SysUtils

Declaration

```
function DeleteFile(const FileName: string): Boolean;
```

The *DeleteFile* function erases the file named by *FileName* from the disk.

If the file cannot be deleted or does not exist, the function returns *False* but does not raise an exception.

Example

The following code erases the file DELETE.ME in the current directory:

```
DeleteFile('DELETE.ME');
```

DeleteIndex method

Applies to

TTable component

Declaration

```
procedure DeleteIndex(const Name: string);
```

The *DeleteIndex* method deletes a secondary index for a *TTable*. *Name* is the name of the index. You must have opened the table with exclusive access (*Exclusive* = True).

Example

```
   Table1.DeleteIndex('NewIndex');
```

See also

AddIndex method

DeleteTable method

Applies to

TTable component

Declaration

```
procedure DeleteTable;
```

The *DeleteTable* method deletes an existing database table. Before calling this method, the *DatabaseName*, *TableName* and *TableType* properties must be assigned values. The table must be closed.

Example

```
   with Table1 do
   begin
     Active := False;
     DatabaseName := 'DBDEMOS';
     TableName := 'Customer';
     TableType := ttParadox;
     DeleteTable;
   end;
```

DescriptionsAvailable method

Applies to

TStoredProc component

Declaration

```
function DescriptionsAvailable: Boolean;
```

The *DescriptionsAvailable* method indicates whether stored procedure parameter information is available from the server. If the information is available, it returns *True*.

Otherwise, it returns *False*. Different servers may require additional information to obtain the parameter information. If *DescriptionsAvailable* returns *False*, you will have to specify parameters either with the Parameters Editor or with explicit code.

Example

```
if not StoredProc1.DescriptionsAvailable then
  begin
  { Build the Parameters property explicitly }
  end;
```

See also

Overload property, *StoredProcName* property

Destination property

Applies to

TBatchMove component

Declaration

```
property Destination: TTable;
```

Destination specifies a *TTable* component corresponding to the database table that will be the destination of the batch move operation. The destination table may or may not already exist.

Example

```
BatchMove1.Destination := Table1;
```

Destroy method

Applies to

All objects and components

Declaration

```
destructor Destroy;
```

The *Destroy* method destroys the object, component, or control and releases the memory allocated to it.

You seldom need to call *Destroy*. Objects designed with Delphi create and destroy themselves as needed, so you don't have to worry about it. If you construct an object by calling the *Create* method, you should call *Free* to release memory and dispose of the object.

See also
Free method, *Release* method

Device property

Applies to
TFontDialog component

Declaration

```
property Device: TFontDialogDevice;
```

The *Device* property determines which device the returned font affects. These are the possible values:

Value	Meaning
fdScreen	Affects the screen
fdPrinter	Affects the printer
fdBoth	Affects both the screen and the printer

Example
This example lets the user select a font to use for printing a file:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  FontName: TFont;
begin
  FontDialog1.Device := fdPrinter;
  FontDialog1.Execute;
  FontName := FontDialog1.Font;
end;
```

See also
TFont object

DeviceID property

Applies to
TMediaPlayer component

Declaration

```
property DeviceID: Word;
```

Run-time and read only. The *DeviceID* property specifies the device ID for the currently open multimedia device.

The value of *DeviceID* is determined when an device is opened with the *Open* method. If no device is open, *DeviceID* is 0.

Example

The following code opens *MediaPlayer1* and displays the *DeviceID* in *Edit1*. If an exception occurs, a message window displays the error number and string.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
var
  MyErrorString: string;
begin
  try
    MediaPlayer1.Open;
    Edit1.Text := IntToStr(MediaPlayer1.DeviceID);
  except
    MyErrorString := 'ErrorCode: ' + IntToStr(Error) + #13#10;
    MessageDlg(MyErrorString + MediaPlayer1.ErrorMessage, mtError, [mbOk], 0);
  end;
end;
```

DeviceType property

Applies to

TMediaPlayer component

Declaration

```
property DeviceType: TMPDeviceTypes;
```

The *DeviceType* property specifies a multimedia device type to open with the *Open* method. The default is *dtAutoSelect*. The valid values for *DeviceType* are *dtAutoSelect*, *dtAVIVideo*, *dtCDAudio*, *dtDAT*, *dtDigitalVideo*, *dtMMMovie*, *dtOther*, *dtOverlay*, *dtScanner*, *dtSequencer*, *dtVCR*, *dtVideodisc*, or *dtWaveAudio*.

If *DeviceType* is *dtAutoSelect*, the device type is determined by the file extension specified in the *FileName* property. If no device type is associated with the extension, you must explicitly specify the correct device type by setting *DeviceType* to a value other than *dtAutoSelect*.

A multimedia device is typically associated with an appropriate file-name extension when you install the device. Associations are specified in the [mci extensions] section of the Windows WIN.INI file. See the documentation for your specific device for instructions about how to associate file-name extensions with the device.

Example

The following code checks to make sure that a filename is specified for *MediaPlayer1* if the *DeviceType* is set to *dtAutoSelect* before opening the device.

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

```
    with MediaPlayer1 do
      if (DeviceType = dtAutoSelect) and (FileName = '') then
        MessageDlg('You must specify a filename for the MediaPlayer', mtError, [mbOk], 0)
      else
        Open;
  end;
```

Directory property

Applies to
TDirectoryListBox, *TFileListBox* components

Declaration

```
property Directory: string;
```

The value of the *Directory* property determines the current directory for the file list box and directory list box components. The file list box displays the files in the directory specified in the *Directory* property. The directory list box displays the value of the *Directory* property as the current directory in the list box.

Examine the example to see how a directory list box and a file list box can work together through their *Directory* properties.

Example
If you have a file list box and a directory list box on a form, this code changes the current directory in the directory list box and displays the files in that directory in the file list box when the user changes directories using the directory list box:

```
procedure TForm1.DirectoryListBox1Change(Sender: TObject);
begin
  FileListBox1.Directory := DirectoryListBox1.Directory;
end;
```

See also
DirLabel property, *Drive* property, *FileList* property

DirectoryExists function **FileCtrl**

Declaration

```
function DirectoryExists(Name: string): Boolean;
```

The *DirectoryExists* function determines whether the directory specified as the value of the *Name* parameter exists. If the directory exists, the function returns *True*. If the directory does not exist, the function returns *False*.

If only a directory name is entered as the value of *Name*, *DirectoryExists* searches for the directory within the current directory. If a full path name is entered, *DirectoryExists* searches for the directory along the designated path.

Example

This example uses an edit box, a label, and a button on a form. When the user enters a directory name in the edit box and clicks the button, whether or not the directory exists is reported in the caption of the label:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if DirectoryExists(Edit1.Text) then
    Label1.Caption := Edit1.Text + ' exists'
  else
    Label1.Caption := Edit1.Text + ' does not exist';
end;
```

See also

ForceDirectories procedure, *SelectDirectory* function

DirLabel property

Applies to

TDirectoryListBox component

Declaration

```
property DirLabel: TLabel;
```

The *DirLabel* property provides a simple way to display the current directory as the caption of a label control. When the current directory changes in the directory list box, the change is reflected in the caption of the label.

Specify the label you want updated with the current directory as the value of the *DirLabel* property.

Example

This example uses a button, an edit box, a label, a drive combo box, a directory list box, a file list box, and a filter combo box on a form. When the user clicks the button, the rest of the controls of the form begin working together like the controls in an Open or Save dialog box.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  DriveComboBox1.DirList := DirectoryListBox1;
  DirectoryListBox1.FileList := FileListBox1;
  DirectoryListBox1.DirLabel := Label1;
  FileListBox1.FileEdit := Edit1;
  FilterComboBox1.FileList := FileListBox1;
end;
```

See also
Caption property, *Directory* property, *DirList* property, *FileEdit* property, *FileList* property

DirList property

Applies to
TDriveComboBox component

Declaration

```
property DirList: TDirectoryListBox;
```

The *DirList* property provides a simple way to connect a drive combo box with a directory list box. When a new drive is selected in the drive combo box, the specified directory list box updates to display the directory structure and the current directory on the new drive.

Specify the directory list box you want updated as the value of *DirList*.

Example
This example uses a button, an edit box, a label, a drive combo box, a directory list box, a file list box, and a filter combo box on a form. When the user clicks the button, the rest of the controls of the form begin working together as the controls in an open or save dialog box do.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  DriveComboBox1.DirList := DirectoryListBox1;
  DirectoryListBox1.FileList := FileListBox1;
  DirectoryListBox1.DirLabel := Label1;
  FileListBox1.FileEdit := Edit1;
  FilterComboBox1.FileList := FileListBox1;
end;
```

See also
Directory property, *DirLabel* property, *Drive* property, *FileEdit* property, *FileList* property

DisableControls method

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure DisableControls;
```

The *DisableControls* method temporarily disconnects the dataset from all *TDataSource* components. While the data sources are disconnected, associated data-aware controls will not reflect changes to datasets. When iterating over a dataset with *Next* or *Prior* methods, calling *DisableControls* first will speed the process, eliminating the need to update the screen each time.

Use *EnableControls* to restore the connection. The dataset maintains a count of the number of calls to *DisableControls* and *EnableControls*, so only the last call to *EnableControls* will actually update the data sources.

Example

```
with Table1 do
  begin
  DisableControls;
{ Move forward five records }
  try
    for I := 1 to 5 do Next;
  finally
{ Update the controls to the current record }
    EnableControls;
  end;
```

DiskFree function — SysUtils

Declaration

```
function DiskFree(Drive: Byte): Longint;
```

DiskFree returns the number of free bytes on the specified drive number, where 0 = Current, 1 = A, 2 = B, and so on.

DiskFree returns -1 if the drive number is invalid.

Example

```
var
  S: string;
begin
  S := IntToStr(DiskFree(0) div 1024) + ' Kbytes free.';
  Canvas.TextOut(10, 10, S);
end;
```

See also

DiskSize function

DiskSize function

SysUtils

Declaration

```
function DiskSize(Drive: Byte): Longint;
```

DiskSize returns the size in bytes of the specified drive number, where 0 = Current, 1 = A, 2 = B, etc. *DiskSize* returns -1 if the drive number is invalid.

Example

```
var
  S: string;
begin
  S := IntToStr(DiskSize(0) div 1024) + ' Kbytes capacity.';
  Canvas.TextOut(10, 10, S);
end;
```

See also

DiskFree function

Display property

Applies to

TMediaPlayer component

Declaration

```
property Display: TWinControl;
```

The *Display* property specifies the display window for an multimedia device that uses a window for output. Assign the name of a windowed control such as a form or panel to *Display* to display output in that control.

The default value of *Display* is **nil**. If the value of *Display* is **nil**, the device creates its own window to display output. Also, if you *Free* the control assigned to *Display* after the device has been opened, video output will be in its own default window.

Examples of multimedia devices that use a window to display output are Animation, AVI Video, Digital Video, Overlay, and VCR.

Example

The following example displays the .AVI video file 'FOOTBALL.AVI' in the client area of *Form2*.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  with MediaPlayer1 do begin
  try
    FileName := 'football.avi';
```

```
      Open;
      Display := Form2;
      Form2.Show;
      Play;
    except
      MessageDlg(MediaPlayer1.ErrorMessage, mtError, [mbOk], 0);
    end;
    end;
  end;
```

See also

Capabilities property, *DeviceType* property, *DisplayRect* property, *Open* method

DisplayFormat property

Applies to

TDateField, *TDateTimeField*, *TIntegerField*, *TSmallintField*, *TTimeField*, *TWordField* components

Declaration

```
property DisplayFormat: string
```

The *DisplayFormat* property is used to format the value of the field for display purposes.

For *TIntegerField*, *TSmallintField*, and *TWordField*, formatting is performed by *FloatToTextFmt*. If *DisplayFormat* is not assigned a string, the value is formatted by *Str*.

For *TDateField*, *TDateTimeField*, and *TTimeField*, formatting is performed by *DateTimeToStr*. If *DisplayFormat* is not assigned a string, the value is formatted according to the default Windows specifications in the [International] section of the WIN.INI file.

For *TBCDField*, *TCurrencyField*, and *TFloatField*, formatting is performed by *FloatToTextFmt*. If *DisplayFormat* is not assigned a string, the value is formatted according to the value of the *Currency* property.

See also

FmtStr procedure, *Format* function, *FormatBuf* function, *FormatDateTime* function, *FormatFloat* function, *StrFmt* function, *StrLFmt* function

DisplayLabel property

Applies to

TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
property DisplayLabel: string;
```

DisplayLabel contains the column heading for a field displayed by a *TDBGrid* component. If *DisplayLabel* is null, the *FieldName* property is used to supply the column heading.

DisplayName property

Applies to

TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
property DisplayName: Pstring;
```

Run-time and read only. *DisplayName* returns the name of the field for display purposes. Use *DisplayName* in your code to use the same algorithm that other Delphi components use when they need the *DisplayLabel* or *FieldName* of a field.

DisplayRect property

Applies to

TMediaPlayer component

Declaration

```
property DisplayRect: TRect;
```

Run-time only. The *DisplayRect* property specifies the rectangle area within the form specified in the *Display* property used to display output from a multimedia device. *DisplayRect* is ignored if *Display* is **nil**.

Assign a *TRect* record to *DisplayRect* to display output in a specific rectangle area on a form. The *Rect* function can be used to create a *TRect* record.

Examples of multimedia devices that use a window to display output are Animation, AVI Video, Digital Video, Overlay, and VCR.

Media that use a rectangle to display output usually perform best if the default *DisplayRect* size is used. To set *DisplayRect* to the default size, use 0, 0 for the lower right corner. Position the rectangle with the upper left corner.

Note You must set *DisplayRect* after the media device is opened.

Example

The following example positions the upper left corner of the display rectangle to 10, 10 and uses the default display size:

```
MediaPlayer1.DisplayRect := Rect(10, 10, 0, 0);
```

See also

Capabilities property, *DeviceType* property, *Open* method

DisplayText property

Applies to

TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
property DisplayText: string;
```

Run-time and read only. The string value for the field when it is displayed in a data-aware control that is not in Edit mode. Data-aware controls such as *TDBEdit* rely on *DisplayText* to provide the formatting for each field.

The default string depends on a field's data type. You can control the strings returned by *DisplayText* by specifying a *DisplayFormat* string or by providing an *OnGetText* event handler.

For a *TStringField*, the contents of the field is formatted using the *EditMask* property.

For a *TIntegerField*, *TSmallintField*, or *TWordField*, if *DisplayFormat* has been assigned a value, *FloatToTextFmt* is called with it; otherwise *Str* is called.

For a *TFloatField* or *TBCDField*, *FloatToTextFmt* is called with the *DisplayFormat* property.

For a *TCurrencyField*, if *DisplayFormat* has been assigned a value, *FloatToTextFmt* is called with it; otherwise, *FloatToTextFmt* is called with the *ffCurrency* flag and *CurrencyDecimals* variable.

For a *TDateTimeField*, *DateTimeToStr* is called with the *DisplayFormat* property. For a *TDateField*, *DateTimeToStr* is called with the *DisplayFormat* property, except that the *ShortDateFormat* variable will be substituted if *DisplayFormat* is unassigned. For a *TTimeField*, *DateTimeToStr* is called with the *DisplayFormat* property, except that the *LongTimeFormat* variable will be substituted if *DisplayFormat* is unassigned.

Example

```
{ Display a message that the current value is invalid }
  MessageDlg(Field1.DisplayText + ' is invalid', mtWarning, [mbOK], 0);
```

DisplayValue property

Applies to
TDBLookupCombo, *TDBLookupList* components

Declaration

```
property DisplayValue : string;
```

Run-time only. The *DisplayValue* is the string that appears in the database lookup combo box or database lookup list box. Its value is contained in the field specified as the *LookupDisplay* field. The current value of the *Value* property, which determines the current record in the lookup table, also determines which string is the *DisplayValue* string.

Example
The following code makes the caption of a button equal to the *DisplayValue* of *DBLookupCombo1*.

```
Button1.Caption := DBLookupCombo1.DisplayValue;
```

See also
LookupField property

DisplayValues property

Applies to
TBooleanField component

Declaration

```
property DisplayValues: string;
```

DisplayValues controls the manner in which the *TBooleanField* is translated to and from display format. Set *DisplayValues* to 'T;F' to use 'T' and 'F' for values of *True* and *False*. You can use any pair of phrases you want, separated by a semicolon. If one phrase is omitted, no text is displayed and a data-aware control with no text assigns the corresponding value to the field. The default value is 'True;False'.

Example

```
Field1.DisplayValues := 'Yes;No';
Field2.DisplayValues := 'Oui;Non';
```

DisplayWidth property

Applies to

TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
property DisplayWidth: Integer;
```

DisplayWidth specifies the number of characters that should be used to display a field in a *TDBGrid* control. For *TStringField, DisplayWidth* is the number of characters in the field. For all other fields the default value is 10.

See also

DisplayLabel property, *DisplayText* property

Dispose procedure **System**

Declaration

```
procedure Dispose(var P: Pointer);
```

The *Dispose* procedure releases memory allocated for a dynamic variable.

After a call to *Dispose*, the value of *P* is undefined and it is an error to reference *P*. If {**$I+**}, you can use exceptions to handle this error. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

Example

```
type
  Str18 = string[18];
 var
  P: ^Str18;
begin
  New(P);
  P^ := 'Now you see it...';
  Dispose(P);                                                   { Now you don't... }
end;
```

See also

FreeMem procedure, *GetMem* procedure, *New* procedure

DisposeStr procedure **SysUtils**

Declaration

```
procedure DisposeStr(P: PString);
```

The *DisposeStr* procedure disposes of the dynamically allocated string pointed to by *P*. *P* must have been allocated previously with *NewStr* function. If the given pointer is **nil** or points to an empty string, *StrDispose* does nothing.

Example

The following code allocates and frees heap space for a copy of string *S* pointed to by *P*, then deallocates the heap space pointed to by *P*:

```
var
  P: PString;
  S: string;
begin
  S := 'Ask me about Blaise';
  P := NewStr(S);
  DisposeStr(P):
end;
```

See also

NewStr function

DitherBackground property

Applies to

TTabSet component

Declaration

```
property DitherBackground: Boolean;
```

The *DitherBackground* property determines whether the selected background color set with the *BackgroundColor* property is dithered. Dithering means the background is lightened by 50%, which is intended to make the tabs easier to see. If *DitherBackground* is *True*, the tab set control background is dithered. If it is *False*, there is no dithering.

The default value is *True*.

Example

This event handler toggles the dithering of the tab set control's background each time the user clicks the form:

```
procedure TForm1.FormClick(Sender: TObject);
begin
  if TabSet1.DitherBackground = True then
    TabSet1.DitherBackground := False
  else
    TabSet1.DitherBackground := True;
end;
```

See also

Color property

DoneWinCrt procedure **WinCrt**

Declaration

```
procedure DoneWinCrt;
```

The *DoneWinCrt* procedure destroys the CRT window.

Calling *DoneWinCrt* before the program ends prevents the CRT window from entering the inactive state.

Down property

Applies to

TSpeedButton component

Declaration

```
property Down: Boolean;
```

The *Down* property of a speed button determines if the button appears in an up (unselected) or down (selected) state. Speed buttons are initially in their up (unselected) state. This occurs because the default setting of the *Down* property is *False*.

To initially display a speed button in its down state, set the *Down* property to *True*. For example, if you use a panel component with several speed buttons to create a tools palette, you might want one of the speed buttons selected when the palette first appears.

Although you can use a group of speed buttons with the *AllowAllUp* property set to *False* to make the tool palette buttons work as a group, you must set the *Down* property for the button you want to be selected initially. You can also use the *Down* property at run time any time you want to put a button in a down state without the user clicking it first.

Example

This code displays the speed button in a down state:

```
  SpeedButton1.Down := True;
```

See also

AllowAllUp property, *GroupIndex* property

DragCursor property

Applies to

TBitBtn, *TButton*, *TCheckBox*, *TComboBox*, *TDBCheckBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBLookupList*, *TDBMemo*, *TDBNavigator*, *TDBRadioGroup*, *TDBText*, *TDirectoryListBox*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TFilterComboBox*, *TGroupBox*, *TImage*, *TLabel*, *TListBox*, *TMaskEdit*, *TMemo*, *TOutline*, *TPaintBox*, *TPanel*, *TRadioButton*, *TScrollBar*, *TScrollBox*, *TShape*, *TNotebook* controls

Declaration

```
property DragCursor: TCursor;
```

The *DragCursor* property determines the shape of the mouse pointer when the pointer is over a component that will accept an object being dragged. These are the possible images:

Value	Image	Value	Image	Value	Image
crDefault	↖	*crSizeNESW*	⤢	*crHourglass*	⌛
crArrow	↖	*crSizeNS*	↕	crDrag	↖
crCross	✚	*crSizeNWSE*	⤡	*crNoDrop*	⊘
crIBeam	I	*crSizeWE*	↔	*crHSplit*	↔
crSize	✥	*crUpArrow*	⇧	*crVSplit*	↕

Example

The following code changes the *DragCursor* of *Memo1* to *crIBeam*.

```
Memo1.DragCursor := crIBeam;
```

See also

BeginDrag method, *Cursor* property, *Cursors* property, *Dragging* method, *EndDrag* method, *OnDragDrop* event, *OnDragOver* event, *OnEndDrag* event

Dragging method

Applies to

All controls

Declaration

```
function Dragging: Boolean;
```

The *Dragging* method specifies whether a control is being dragged. If *Dragging* returns *True*, the control is being dragged. If *Dragging* is *False*, the control is not being dragged.

Example

This example uses three check boxes on a form. When the user begins dragging one of the check boxes, the color of the form changes:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  CheckBox1.DragMode := dmAutomatic;
  CheckBox2.DragMode := dmAutomatic;
  CheckBox3.DragMode := dmAutomatic;
end;

procedure TForm1.FormDragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  if CheckBox1.Dragging then
    Color := clAqua;
  if CheckBox2.Dragging then
    Color := clYellow;
  if CheckBox3.Dragging then
    Color := clLime;
end;
```

See also

BeginDrag method, *DragMode* property, *EndDrag* method, *OnDragDrop* event, *OnDragOver* event, *TDragState* type

DragMode property

Applies to

TBitBtn, *TButton*, *TCheckBox*, *TComboBox*, *TDBCheckBox*, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBText*, *TDBListBox*, *TDBLookupCombo*, *TDBLookupList*, *TDBMemo*, *TDBNavigator*, *TDBRadioGroup*, *TDirectoryListBox*, *TDrawGrid*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TFilterComboBox*, *TGroupBox*, *TImage*, *TLabel*, *TListBox*, *TMaskEdit*, *TMemo*, *TOLEContainer*, *TOutline*, *TPaintBox*, *TRadioButton*, *TScrollBar*, *TScrollBox*, *TShape*, *TStringGrid*, *TNotebook* controls

Declaration

```
property DragMode: TDragMode;
```

The *DragMode* property determines the drag and drop behavior of a control. These are the possible values:

Value	Meaning
dmAutomatic	If *dmAutomatic* is selected, the control is ready to be dragged; the user just clicks and drags it.
dmManual	If *dmManual* is selected, the control can't be dragged until the application calls the *BeginDrag* method.

If a control's *DragMode* property value is *dmAutomatic*, the application can disable the drag and drop capability at run time by changing the *DragMode* property value to *dmManual*.

Example

This example determines whether the drag mode of the button on the form is manual. If it is, the dragging the button becomes possible.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Button1.DragMode = dmManual then
    Button1.BeginDrag(True);
end;
```

See also

BeginDrag method, *EndDrag* method

Draw method

Applies to

TCanvas object

Declaration

```
procedure Draw(X, Y: Integer; Graphic: TGraphic);
```

The *Draw* method draws the graphic specified by the *Graphic* parameter on the canvas at the location given in the screen pixel coordinates (*X*, *Y*). Graphics can be bitmaps, icons, or metafiles.

Example

The following code draws the graphic in C:\WINDOWS\TARTAN.BMP centered in *Form1* when the user clicks *Button1*. Attach this code to the *OnClick* event handler of *Button1*.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Bitmap1: TBitmap;
begin
  Bitmap1 := TBitmap.Create;
```

```
    Bitmap1.LoadFromFile('c:\windows\tartan.bmp');
    Form1.Canvas.Draw((Form1.Width div 2)-(Bitmap1.Width div 2),
      (Form1.Height div 2) - (Bitmap1.Height div 2), Bitmap1);
  end;
```

See also
StretchDraw method, *TBitmap* object, *TIcon* object, *TMetafile* object

DrawFocusRect method

Applies to
TCanvas object

Declaration

```
procedure DrawFocusRect(const Rect: TRect);
```

The *DrawFocusRect* method draws a rectangle in the style used to indicate that the rectangle has the focus. Because this is an XOR function, calling it a second time and specifying the same rectangle removes the rectangle from the screen.

The rectangle this function draws cannot be scrolled. To scroll an area containing a rectangle drawn by this function, call *DrawFocusRect* to remove the rectangle from the screen, scroll the area, and then call *DrawFocusRect* to draw the rectangle in the new position.

Example
This examples uses a radio button and a button on a form. When the user clicks the button, the code draws a rectangle around the radio button.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NewRect: TRect;
begin
  NewRect := RadioButton1.BoundsRect;
  with NewRect do
  begin
    Left := Left - 10;
    Top := Top - 10;
    Right := Right + 10;
    Bottom := Bottom + 10;
  end;
  Form1.Canvas.DrawFocusRect(NewRect);
end;
```

See also
Arc method, *Chord* method, *Ellipse* method, *FrameRect* method, *Pie* method, *Rectangle* method

Drive property

Applies to

TDirectoryListBox, *TDriveComboBox*, *TFileListBox* components

Declaration

```
property Drive: Char;
```

Run-time only. For the drive combo box, the *Drive* property determines which drive is displayed in the edit control of the combo box. When the user uses the drive combo box to select a new drive, the selected drive becomes the value of the *Drive* property. The value of the *Text* property also changes to the new volume name when the *Drive* property value changes.

For the directory list box, the *Drive* property determines which drive the list box displays the directory structure on. When the value of *Drive* changes, the *Directory* value changes also to the current directory on the specified drive.

For the file list box, the *Drive* property determines which drive the list box displayed the files on. When the value of *Drive* changes, the *Directory* value also changes to the current directory on the specified drive.

Example

The following example assumes that a drive combo box, a file list box, and a directory list box are on a form. This code changes the drive displayed in the drive combo box, displays the current directory of the selected drive in the directory list box, and displays the files in the current directory of the selected drive in the file list box when the user selects a drive in the drive combo box:

```
procedure TForm1.DriveComboBox1Change(Sender: TObject);
begin
  DirectoryListBox1.Drive := DriveComboBox1.Drive;
  FileListBox1.Directory := DirectoryListBox1.Directory;
end;
```

See also

Directory property, *DirList* property, *Text* property

DriverName property

Applies to

TDataBase component

Declaration

```
property DriverName: TSymbolStr;
```

DriverName is the name of a BDE driver, such as STANDARD (for dBASE and Paradox), ORACLE, SYBASE, INFORMIX or INTERBASE. This property will be cleared if *AliasName* is set, because an *AliasName* specifies a driver type. Conversely, setting this property will clear *AliasName*.

If you try to set *DriverName* of a *TDatabase* for which *Connected* is *True*, Delphi will raise an exception.

Example

```
Database1.DriverName := 'STANDARD';
```

DropConnections method

Applies to

TSession component

Declaration

```
procedure DropConnections;
```

The *DropConnections* method drops all inactive database connections. By default, temporary database components keep their connections to the server open even when not in use so that they do not have to log in to the server each time a datasetcomponent is opened.

Example

```
Session.DropConnections;
```

See also

Session variable, *Temporary property*

DropDown method

Applies to

TDBLookupCombo component

Declaration

```
procedure DropDown;
```

The *DropDown* method opens or "drops down" the database lookup combo box so that the user has a list of values to choose from.

See also

CloseUp method

DropDownCount property

Applies to
TComboBox, *TDBComboBox*, *TDBLookupCombo* components

Declaration

```
property DropDownCount: Integer;
```

The *DropDownCount* property determines how long the drop-down list of a combo box is. By default, the drop-down list is long enough to contain eight items without requiring the user to scroll to see them all. If you would like the drop-down list to be smaller or larger, specify a number larger or smaller than eight as the *DropDownCount* value.

If the *DropDownCount* value is larger than the number of items in the drop-down list, the drop-down list is just large enough to hold all the items and no larger. For example, if the list contains three items, the drop-down list is only long enough to display the three items, even if the *DropDownCount* is eight.

Example
The following code assigns three to the *DropDownCount* property of *ComboBox1*. To see more than three items in the drop-down list, the user must scroll.

```
   ComboBox1.DropDownCount := 3;
```

See also
DropDownWidth property

DropDownWidth property

Applies to
TDBLookupCombo component

Declaration

```
property DropDownWidth: Integer;
```

The *DropDownWidth* property determines how wide the drop-down list of the combo box is in pixels. The default value is 0, which means the drop-down list is the same width as the combo box.

The *DropDownWidth* property is useful when you are displaying multiple fields, and therefore, multiple columns in the database lookup combo box.

Example
This code displays three fields in the drop-down list of the database lookup combo box. Each column has a title and is separated from the other columns by a line. The combo

box displays ten items at a time; therefore, the user must scroll to view the rest of the items. The drop-down list is 600 pixels wide so all the fields fit in the drop-down list.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 DBLookupCombo1.LookupDisplay := 'Company;City;Country';
 DBLookupCombo1.Options := [loColLines,loTitles];
 DBLookupCombo1.DropDownCount := 10;
 DBLookupCombo1.DropDownWidth := 600;
end;
```

See also

DropDownCount property, *LookupDisplay* property, *Options* property

DroppedDown property

Applies to

TComboBox, *TDBComboBox* components

Declaration

```
property DroppedDown: Boolean;
```

Run-time only. The *DroppedDown* property determines whether the drop-down list of the combo box is open or closed. If *DroppedDown* is *True*, the drop-down list is visible. If *DroppedDown* is *False*, the drop-down list is closed. The default value is *False*.

See also

DropDownCount property

dsEditModes const

Declaration

```
dsEditModes = [dsEdit, dsInsert, dsSetKey];
```

dsEditModes is the subset of *TDataSetState* elements which the *State* property of a dataset component will have if the current record of the dataset is being modified. It is also uses by the *UpdateRecord of a dataset component.*

DSeg function System

Declaration

```
function DSeg: Word;
```

The *DSeg* function returns the current value of the DS register.

The result is the segment address of the data segment.

Example

```
function MakeHexWord(w: Word): string;
const
  hexChars: array [0..$F] of Char = '0123456789ABCDEF';
var
  HexStr : string;
begin
  HexStr := '';
  HexStr := HexStr + hexChars[Hi(w) shr 4];
  HexStr := HexStr + hexChars[Hi(w) and $F];
  HexStr := HexStr + hexChars[Lo(w) shr 4];
  HexStr := HexStr + hexChars[Lo(w) and $F];
  MakeHexWord := HexStr;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  Y: Integer;
  S: string;
begin
  Y := 10;
  S := 'The current code segment is $' + MakeHexWord(CSeg);
  Canvas.TextOut(5, Y, S);
  Y := Y + Canvas.TextHeight(S) + 5;
  S := 'The global data segment is $' + MakeHexWord(DSeg);
  Canvas.TextOut(5, Y, S);
  Y := Y + Canvas.TextHeight(S) + 5;
  S := 'The stack segment is $' + MakeHexWord(SSeg);
  Canvas.TextOut(5, Y, S);
  Y := Y + Canvas.TextHeight(S) + 5;
  S := 'The stack pointer is at $' + MakeHexWord(SPtr);
  Canvas.TextOut(5, Y, S);
  Y := Y + Canvas.TextHeight(S) + 5;
  S := 'i is at offset $' + MakeHexWord(Ofs(i));
  Canvas.TextOut(5, Y, S);
  Y := Y + Canvas.TextHeight(S) + 5;
  S := 'in segment $' + MakeHexWord(Seg(i));
  Canvas.TextOut(5, Y, S);
end;
```

See also

CSeg function, *SSeg* function

Duplicates property

Applies to
TStringList object

Declaration

```
property Duplicates: TDuplicates;
```

The *Duplicates* property determines whether duplicate strings are allowed in the sorted list of strings of a string list object. If the list is not sorted, the value of *Duplicates* has no effect. These are the possible values:

Value	Meaning
dupIgnore	Attempts to add a duplicate string to a sorted string list are ignored
dupAccept	Duplicate strings can be added to a sorted string list
dupError	Adding a duplicate string results in an *EListError* exception

Example
The following code makes *StringList1* ignore duplicate entries.

```
  StringList1.Duplicates := dupIgnore;
```

See also
Sort method, *Sorted* property

EAbort object SysUtils

Declaration

```
EAbort = class(Exception)
```

The *EAbort* exception is Delphi's "silent" exception. When it is raised, no message box appears to inform the user. Your application can handle the exception without the user ever knowing it occurred.

EBreakpoint object SysUtils

Declaration

```
EBreakpoint = class(EProcessorException);
```

The *EBreakpoint* exception is a hardware exception. It occurs when your application generates a breakpoint interrupt. Usually Delphi's integrated debugger handles breakpoint exceptions.

EClassNotFound object

Classes

Declaration

```
EClassNotFound = class(EFilerError);
```

The *EClassNotFound* exception is raised when a component exists on a form, but it has been deleted from the type declaration. For example, this form type declaration includes two panel components:

```
type
  TForm1 = class(TForm)
    Panel1: TPanel;
    SpeedButton1: TSpeedButton;
    SpeedButton2: TSpeedButton;
    Panel2: TPanel;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

If you compile the application, then delete *Panel2*, for example, from the *TForm1* type declaration, the next time you attempt to run the application, the class not found exception is raised.

EComponentError object

Classes

Declaration

```
EComponentError = class(Exception);
```

The *EComponentError* exception is raised when an attempt is made to register a component outside of the *Register* procedure. It is also raised when your application changes the name of a component at run time so that it has the same name as another component. It can also occur if the name of a component is changed to a name that is not a valid Object Pascal identifier.

EConvertError object

SysUtils

Declaration

```
EConvertError = class(Exception);
```

The *EConvertError* exception is raised when either the *StrToInt* or *StrToFloat* functions are unable to convert the specified string to a valid integer or floating-point value, respectively. For example, this code raises the convert error exception because 3.4 is not a valid integer:

```
var
  X: Integer;
begin
  X := StrToInt('3.4');
end;
```

EDatabaseError object

DB

Declaration

```
EDatabaseError = class(Exception);
```

The *EDatabaseError* type is the exception type raised when a database error is detected by a component. Use *EDatabaseError* with an exception handling block or to create a database exception. With an exception handling block, you can detect the condition and handle it yourself. If something in your code encounters an error, you can create and raise the exception yourself.

```
{ Try to open Table1 }
repeat { until successful or Cancel button is pressed }
  try
    Table1.Active := True; { See if it will open }
    Break; { If no error, exit the loop }
  except
    on EDatabaseError do
      { Ask if it is OK to retry }
      if MessageDlg('Could not open Table1 - check server', mtError,
        [mbOK, mbCancel], 0) <> mrOK then raise; { If not, reraise to abort }
      { Otherwise resume the repeat loop }
  end;
  until False;
{ Test for an error and raise an exception if so }
if { some error has occured } then
  raise EDatabaseError.Create('Some error has occured');
```

EDBEngineError object

DB

```
EDBEngineError = class(EDatabaseError)
private
  FErrors: TList;
  function GetError(Index: Integer): TDBError;
  function GetErrorCount: Integer;
public
  constructor Create(ErrorCode: DBIResult);
  destructor Destroy;
  property ErrorCount: Integer;
  property Errors[Index: Integer]: TDBError;
end;
```

Description

The *EDBEngineError* exception is raised whenever a BDE error occurs. The exception contains two **public** properties of significance:

Property	How used
Errors	A list of the entire Borland Database Engine error stack. The first error has an index value of 0.
ErrorCount	The total number of errors contained in the *Errors* property.

The objects contained in the *Errors* property are of type *TDBError*, which is declared like this:

```
TDBError = class
private
  FErrorCode: DBIResult;
  FNativeError: Longint;
  FMessage: TMessageStr;
  function GetCategory: Byte;
  function GetSubCode: Byte;
public
  constructor Create(Owner: EDBEngineError; ErrorCode: DBIResult;
    NativeError: Longint; Message: PChar);
  property Category: Byte;
  property ErrorCode: DBIResult;
  property SubCode: Byte;
  property Message: TMessageStr;
  property NativeError: Longint;
end;
```

These are the **public** properties of the *TDBError* object:

Property	How used
ErrorCode	The error code returned by the Borland Database Engine
Category	The category of the error referenced by *ErrorCode*
SubCode	The subcode of the error code
NativeError	The remote error code returned from the server. If *NativeError* is 0, the error is not a server error.
Message	The server message for native errors, or the BDE message for non-server errors.

EDBEditError object

Mask

Declaration

```
EDBEditError = class(Exception);
```

The *EDBEditError* exception is raised when the data is not compatible with the mask specified for the field.

EDDEError object

DDEMan

Declaration

```
EDDEError = class(Exception);
```

The *EDDEError* exception is raised when your application can't find the specified server or conversation, or when a session is unexpectedly terminated.

Edit method

Applies to

TDataSource, *TQuery*, *TTable* components

For tables and queries

Declaration

```
procedure Edit;
```

The *Edit* method prepares the current record of the *dataset* for changes and puts the dataset in Edit state, setting the *State* property to *dsEdit*. Data-aware controls cannot modify existing records unless the dataset is in Edit state.

Calling this method for a dataset that cannot be modified raises an exception. The *CanModify* property will be *True* for datasets that can be modified. This method is valid only for datasets that return a live result set.

Example

```
   Table1.Edit;
```

See also

AutoEdit property

For datasource components

Declaration

```
procedure Edit;
```

Edit calls the dataset's *Edit* method if *AutoEdit* is *True* and *State* is *dsBrowse.*

See also

DataSet property, *Insert* method

EditFormat property

Applies to
TIntegerField, *TSmallintField*, *TWordField* components

Declaration

```
property EditFormat: string;
```

EditFormat is used to format the value of the field for editing purposes.

For *TIntegerField*, *TSmallintField*, and *TWordField*, formatting is performed by *FloatToTextFmt*. If *EditFormat* is not assigned a string, but *DisplayFormat* does have a value, the *DisplayFormat* string is used. Otherwise, the value is formatted by to the shortest possible string.

For *TBCDField*, *TCurrencyField*, and *TFloatField*, formatting is performed by *FloatToTextFmt*. If *EditFormat* is not assigned a string but *DisplayFormat* does have a value, the *DisplayFormat* string will be used. Otherwise, the value is formatted according to the value of the *Currency* property.

EditKey method

Applies to
TTable component

Declaration

```
procedure EditKey;
```

Use the *EditKey* method to modify the contents of the search key buffer. This method is useful only when searching on multiple fields after calling *SetKey*. Call *GotoKey* to move the cursor to the record with the corresponding key.

EditKey differs from *SetKey* in that the latter clears all the elements of the search key buffer to the default values (or NULL). *EditKey* leaves the elements of the search key buffer with their current values.

Example

```
with Table1 do
  begin
  EditKey;
  FieldByName('State').AsString := 'CA';
  FieldByName('City').AsString := 'Santa Barbara';
  GotoKey;
  end;
```

See also
IndexFields property

EditMask property

Applies to

TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMaskEdit*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
property EditMask: string;
```

The *EditMask* property is the mask that is used to limit the data that can be put into a masked edit box or entered into a data field. A mask restricts the characters the user can enter to valid characters and formats. If the user attempts to enter a character that is not valid, the edit box does not accept the character. Validation is performed on a character-by-character basis. Use an *OnValidate* event to validate the entire input.

For a *TStringField*, *EditMask* can be used to format output with the *DisplayText* property.

A mask consists of three fields with semicolons separating the fields. The first part of the mask is the mask itself. The second part is the character that determines whether the literal characters of a mask are saved as part of the data. The third part of the mask is the character used to represent a blank in the mask.

These are the special characters used to create masks:

Character	Meaning in mask
!	If a ! character appears in the mask, leading blanks don't appear in the data. If a ! character is not present, trailing blanks don't appear in the data.
>	If a > character appears in the mask, all characters that follow are in uppercase until the end of the mask or until a < character is encountered.
<	If a < character appears in the mask, all characters that follow are in lowercase until the end of the mask or until a > character is encountered.
<>	If these two characters appear together in a mask, no case checking is done and the data is formatted with the case the user uses to enter the data.
\	The character that follows a \ character is a literal character. Use this character when you want to allow any of the mask special characters as a literal in the data.
L	The L character requires only an alphabetic character only in this position. For the US, this is A-Z, a-z.
l	The l character permits only an alphabetic character in this position, but doesn't require it.
A	The A character requires an alphanumeric character only in this position. For the US, this is A-Z, a-z, 0-9.
a	The a character permits an alphanumeric character in this position, but doesn't require it.
C	The C character requires a character in this position.
c	The c character permits a character in this position, but doesn't require it.
0	The 0 character requires a numeric character only in this position.
9	The 9 character permits a numeric character in this position, but doesn't require it.
#	The # character permits a numeric character or a plus or minus sign in this position, but doesn't require it.

Character	Meaning in mask
:	The : character is used to separate hours, minutes, and seconds in times. If the character that separates hours, minutes, and seconds is different in the International settings of the Control Panel utility on your computer system, that character is used instead of :.
/	The / character is used to separate months, days, and years in dates. If the character that separates months, days, and years is different in the International settings of the Control Panel utility on your computer system, that character is used instead of /.
;	The ; character is used to separate masks.
_	The _ character automatically inserts a blank the edit box. When the user enters characters in the field, the cursor skips the blank character. When using the EditMask property editor, you can change the character used to represent blanks. You can also change this value programmatically. See the following table.

These characters (already mentioned in previous table) are typed constants declared in the *Mask* unit whose value you can change at run time, although the need for this should be limited:

Typed constant	Initial value	Meaning
DefaultBlank	_	Blanks in the mask are represented by the _ character
MaskFieldSeparator	;	The ; character separates the fields of a mask.
MaskNoSave	0	The 0 character means that the mask is not saved as part of the data. The 1 character means that the mask is saved as part of the data. For example, a telephone number could have parentheses around the area code as part of the mask. If *MaskNoSave* is 0, the parentheses do not become part of the data, making the size of the field slightly smaller.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Mask.MaskFieldSeparator := ',';
  Mask.DefaultBlank := '@';
  MaskEdit1.EditMask := '999-999,1,@';
end;
```

Example

This example assigns an edit mask to the masked edit box on the form. The edit mask makes it easy to enter American telephone numbers in the edit box.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MaskEdit1.EditMask := '!\(999\)000-0000;1;
  MaskEdit1.Text := '';
  MaskEdit1.AutoSelect := False;
end;
```

See also

OnGetEditMask event, *EditText* property, *Text* property

EditMaskPtr property

Applies to
TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
property EditMask: string;
```

Run-time and read only. *EditMaskPtr* is a pointer to the *EditMask* property of a string.

Example

```
  Field1.EditMask := Field2.EditMaskPtr^;
```

EditorMode property

Applies to
TDBGrid, *TDrawGrid*, *TStringGrid* components

Declaration

```
property EditorMode: Boolean;
```

Run-time only. The *EditorMode* property determines whether the grid is in automatic Edit mode. When the grid is in automatic edit mode, the user can type in a cell without having to press *Enter* or *F2* first. If the *Options* property set includes the value *goAlwaysShowEditor* (*goAlwaysShowEditor* is *True*), the grid is in automatic edit mode, just as if *EditorMode* is *True*. While you are most likely to set the *Options* property values at design time, the *EditorMode* property makes it easy to control when editing is permitted at run time.

If *EditorMode* is *True*, the grid is in automatic edit mode as long as the *Options* property set includes the value *goEditing* (or *dgEditing* for the data grid). If *goEditing* or *dgEditing* is not in the *Options* set (*goEditing* or *dgEditing* is *False*), setting *EditorMode* to *True* has no effect, and the user cannot edit the contents of a cell.

If *EditorMode* is *False* and the *Options* property set includes the value *goEditing* or *dgEditing*, but not the value *goAlwaysShowEditor* (or *dgAlwaysShowEditor* for the data grid), the user can enter edit mode by pressing either *Enter* or *F2* before editing the contents of each cell.

Example
The following code sets *EditorMode* to *True* for *StringGrid1*.

```
  StringGrid1.EditorMode := True;
```

See also
Options property

EditRangeEnd method

Applies to
TTable component

Declaration

```
procedure EditRangeEnd;
```

EditRangeEnd enables you to modify the beginning range of the dataset filter established with *SetRangeEnd*. Subsequent assignments to field values will modify the values of the ending field range previously set with *SetRangeEnd*. Call *ApplyRange* to apply the new range and filter the dataset.

EditRangeEnd differs from *SetRangeEnd* in that the latter clears all the elements of the search key buffer to the default values (NULL). *EditRangeEnd* leaves the elements of search key buffer with their current values.

Note With Paradox or dBASE tables, these methods work only with indexed fields. With SQL databases, they can work with any columns specified in the *IndexFieldNames* property.

Example

```
{ Limit the range from 'Goleta' to 'Santa Barbara'}
with Table1 do
  begin
  EditRangeStart; { Set the beginning key }
  FieldByName('City').AsString := 'Goleta';
  EditRangeEnd; { Set the ending key }
  FieldByName('City').AsString := 'Santa Barbara';
  ApplyRange; { Tell the dataset to establish the range }
  end;
```

See also
KeyExclusive property, *KeyFieldCount* property

EditRangeStart method

Applies to
TTable component

Declaration

```
procedure EditRangeStart;
```

EditRangeStart enables you to modify the lower key limit established with *SetRangeStart*. Call *ApplyRange* to apply the new range and filter the dataset.

EditRangeStart differs from *SetRangeStart* in that the latter clears all the elements of the search key buffer to the default values (NULL). *EditRangeStart* leaves the elements of the search key buffer with their current values.

Note With Paradox or dBASE tables, these methods work only with indexed fields. With SQL databases, they can work with any columns specified in the *IndexFieldNames* property.

Example

```
{ Limit the range from 'Goleta' to 'Santa Barbara'}
with Table1 do
  begin
  EditRangeStart; { Set the beginning key }
  FieldByName('City').AsString := 'Goleta';
  EditRangeEnd; { Set the ending key }
  FieldByName('City').AsString := 'Santa Barbara';
  ApplyRange; { Tell the dataset to establish the range }
  end;
```

See also

KeyExclusive property, *EditRangeEnd* method, *SetRange* method, *SetRangeEnd* method

EditText property

Applies to

TDBEdit, *TMaskEdit* components

Declaration

```
property EditText: string;
```

Run-time only. The *EditText* property is the value of the *Text* property as it appears in the edit box at run time with the mask specified in the *EditMask* property applied. If literal mask characters are not saved and no character is substituted for blanks, the values of *EditText* and *Text* are the same.

EditText is what the user actually sees in the edit box at run time.

See also

EditMask property, *Text* property

EDivByZero object SysUtils

Declaration

```
EDivByZero = class(EIntError);
```

The *EDivByZero* exception is an integer math exception. The exception occurs when your application attempts to divide an integer type by zero. For example, this code raises an *EDivByZero* exception:

```
var
 X, Y: Integer;
begin
 X := 0;
 Y := 10;
 Y := Y div X;
end;
```

EFault object — SysUtils

Declaration

```
EFault = class(EProcessorException);
```

The *EFault* exception is the base exception object from which all other exception fault objects descend. These are the fault exceptions:

Exception	Meaning
EGPFault	A general protect fault, which is usually caused by an uninitialized pointer or object.
EStackFault	Illegal access to the processor's stack segment.
EPageFault	The Windows memory manager was unable to correctly use the Windows swap file.
EInvalidOpCode	The processor encountered an undefined instruction. Usually this means the processor was trying to execute data or uninitialized memory.

EFCreateError object — Classes

Declaration

```
EFCreateError = class(EStreamError);
```

The *EFCreateError* exception is raised when an error occurs as a file is being created. For example, the specified file might have an invalid file name, or the file can't be recreated because it is read only.

EFilerError object — Classes

Declaration

```
EFilerError = class(EStreamError);
```

The *EFilerError* is raised when an attempt is made to register the same class twice. It is also the parent of these exceptions that occur when reading or writing streams:

Exception	Description
EReadError	The *ReadBuf* method cannot read the specified number of bytes
EWriteError	The *WriteBuf* method cannot write the specified number of bytes
EClassNotFound	A component on the form has been deleted from the form type declaration

EFOpenError object Classes

Declaration

```
EFOpenError = class(EStreamError);
```

The *EFOpenError* exception is raised when an attempt is made to create a file stream object and the specified file cannot be opened.

EGPFault object SysUtils

Declaration

```
EGPFault = class(EFault);
```

The *EGPFault* is a hardware exception that is raised when your application attempts to access memory that isn’t legal for your application to access. These are the most common causes of general protection faults (GPF):

1. Loading invalid values into segment registers
2. Accessing memory beyond a segment’s limit
3. Writing to read-only code segments
4. Attempting to access an uninitialized pointer or object

The most likely cause in Delphi programs is probably the fourth one: attempting to access an uninitialized pointer or object.

EInOutError object SysUtils

Declaration

```
EInOutError = class(Exception)
public
  ErrorCode: Integer;
end;
```

The *EInOutError* is raised any time an input/output MS-DOS error occurs. The resulting error code is returned in the *ErrorCode* field.

The **$I+** directive must be in effect or input/output errors will not raise an exception. If an I/O error occurs when your application is in the **$I-** state, your application must call the *IOResult* function to clear the error.

EIntError object SysUtils

Declaration

```
EIntError = class(Exception);
```

Description

The *EIntError* exception is a generic integer math exception. Although it is never raised in the run-time library, it is the base from which other integer math exceptions descend. These are the integer math exceptions:

Exception	Meaning
EDivByZero	An attempt was made to divide by zero
ERangeError	Number or expression out of range
EIntOverflow	Integer operation overflowed

EIntOverflow object SysUtils

Declaration

```
EIntOverFlow = class(EIntError);
```

Description

The *EIntOverFlow* exception is an integer math exception. It occurs when a calculated result is too large to fit within the register allocated for it and therefore, data is lost. For example, this code results in an overflow condition as the calculation result overflows a register:

```
var
  SmallNumber: Shortint;
   X, Y: Integer;
begin
  X := 127;
  Y := 127;
  SmallNumber := X * Y * 100;
end;
```

The *EIntOverFlow* occurs only if range checking is turned on (your code includes the **$O+** directive or you set the Overflow-checking option using the Options | Project dialog box).

See also the *ERangeError* exception.

EInvalidCast object SysUtils

Declaration

```
EInvalidCast = class(Exception);
```

The *EInvalidCast* exception occurs when your application tries to typecast an object into another type using the **as** operator, and the requested typecast is illegal. For example, an invalid typecast exception is raised if in this expression *AnObject* is not of a type compatible with *TObjectType*:

```
AnObject as TObjectType
```

EInvalidGraphic object Graphics

Declaration

```
EInvaldGraphic = class(Exception);
```

An *EInvalidGraphic* exception is raised when your application attempts to access a file that is not a valid bitmap, icon, metafile, or user-defined graphic type when your application expects it to be. For example, this code raises an invalid graphic exception:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Image1.Picture.LoadFromFile('README.TXT');
end;
```

Because the README.TXT file doesn't contain a valid graphic, the exception is raised.

EInvalidGraphicOperation object Graphics

Declaration

```
EInvalidGraphicOperation = class(Exception);
```

Description

An *EInvalidGraphicOperation* is raised when an illegal operation is attempted on a graphic. For example, if your application attempts to resize an icon, the invalid graphic operation is raised:

```
var
  AnIcon: TIcon;
begin
  AnIcon := TIcon.Create;
  AnIcon.LoadFromFile('C:\WINDOWS\DIRECTRY.ICO');
  AnIcon.Width := 100;        { an invalid graphic operation exception is raised }
  ...
```

EInvalidGridOperation object **Grids**

Declaration

```
EInvalidGridOperation = class(Exception);
```

An *EInvalidGridOperation* is raised when an illegal operation is attempted on a grid. For example, your application might try to access a cell that does not exist within the grid.

EInvalidImage object **Classes**

Declaration

```
EInvalidImage = class(EFilerError);
```

The *EInvalidImage* exception is raised when your application attempts to read a resource file and the specified file is not a resource file. When your application calls the *ReadComponentRes* method, it must also use the corresponding *WriteComponentRes* method to write to a resource file. Similarly, when you application calls the *ReadComponent* method, it must use the corresponding *WriteComponent* method.

EInvalidOp object **SysUtils**

Declaration

```
EInvalidOp = class(EMathError);
```

The *EInvalidOp* exception is a floating-point math exception. It occurs whenever the processor encounters an undefined instruction. For example, if your application uses an opcode that is not available to the 80287 floating-point unit and you run the application on a 80286 computer, the invalid opcode exception is raised.

EInvalidOpCode object **SysUtils**

Declaration

```
EInvalidOpCode = class(EFault);
```

The *EInvalidOpCode* exception is a hardware fault exception. It occurs when the processor encounters an undefined instruction. Usually this means the processor was attempting to execute data or uninitialized memory. It could also happen if your application jumps to the middle of an instruction somehow. An invalid opcode exception represents a serious failure in the operating environment. Your application should encounter it rarely.

EInvalidOperation object — Controls

Declaration

```
EInvalidOperation = class(Exception);
```

Description

An *EInvalidOperation* exception is raised when your application does some operation that requires a window handle and your component does not have a parent (Parent = **nil**). It can also occur if you try to perform drag and drop operations from the form such as *Form1.BeginDrag*.

EInvalidPointer object — SysUtils

Declaration

```
EInvalidPointer = class(Exception);
```

The *EInvalidPointer* exception is raised when your application attempts an invalid pointer operation. For example, it can occur if your application tries to dispose of the same pointer twice, or your application calls the *Free* method twice to destroy an object.

Eject method

Applies to

TMediaPlayer component

Declaration

```
procedure Eject;
```

The *Eject* method ejects the loaded medium from the open multimedia device. *Eject* is called when the Eject button on the media player control is clicked at run time.

Upon completion, *Eject* stores a numerical error code in the *Error* property, and the corresponding error message in the *ErrorMessage* property.

The *Wait* property determines whether control is returned to the application before the *Eject* method has been completed. The *Notify* property determines whether *Eject* generates an *OnNotify* event.

Example

This code ejects the CD from the CD-ROM player after 10 seconds. For the code to run correctly, you must have your CD audio device installed correctly, and the device must have software ejecting capabilities.

```
var
  TimerOver: Word;
```

```
procedure TForm1.FormClick(Sender: TObject);
begin
  MediaPlayer1.DeviceType := dtCDAudio;
  MediaPlayer1.Open;
  MediaPlayer1.Play;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if TimeOver = 10 then
  begin
    MediaPlayer1.Eject;
    MediaPlayer1.Close;
    Timer1.Enabled := False;
  end
  else
    Inc(TimeOver);
end;
```

See also

Capabilities property

EListError object — Classes

Declaration

```
EListError = class(Exception);
```

The *EListError* is an exception that is raised when an error is made in a list, string, or string list object. List error exceptions commonly occur when your application refers to an item in a list that is out of the list's range. For example, the following code is an event handler that attempts to access an item in a list box that does not exist. The *EListError* is raised and handled:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add('First item');                 { Items[0] }
  ListBox1.Items.Add('Another item');               { Items{1] }
  ListBox1.Items.Add('Still another item');         { Items[2] }
  try
    ListBox1.Items[3] := 'This item does not exist';
  except
    on EListError do
      MessageDlg('List box contains fewer than 4 strings', mtWarning, [mbOK], 0);
    end;
end;
```

Also, a list error occurs when your application tries to add a duplicate string to a string list object when the value of the *Duplicates* property is *dupError*.

A list error exception is raised when you insert a string into a sorted string list, as the string you insert at the specified position may put the string list out of sorted order. For example, this code raises the list error exception:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 5 do
    ListBox1.Items.Add('Item ' + IntToStr(I));
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  try
    ListBox1.Items.Insert(0, 'Try to insert here');
  except
    on EListError do
       MessageDlg('Attempt to insert into a sorted list', mtWarning, [mbOK], 0);
    end;
  end;
```

Ellipse method

Applies to
TCanvas object

Declaration

```
procedure Ellipse(X1, Y1, X2, Y2: Integer);
```

The *Ellipse* method draws an ellipse defined by a bounding rectangle on the canvas. The top left point of the bounding rectangle is at pixel coordinates (*X1*, *Y1*) and the bottom right point is at (*X2*, *Y2*). If the points of the rectangle form a square, a circle is drawn.

Example
The following code draws an ellipse filling the background of a form:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Ellipse(0, 0, ClientWidth, ClientHeight);
end;
```

See also
Arc method, *Chord* method, *Draw* method, *DrawFocusRect* method, *Pie* method, *StretchDraw* method

EMathError object

SysUtils

Declaration

```
EMathError = class(Exception);
```

The *EMathError* exception is never raised on its own, but it provides a base exception object from which all the specific floating-point math exceptions descend. These are the floating-point math exceptions:

Exception	Meaning
EInvalidOp	Processor encountered an undefined instruction
EZeroDivide	Attempt to divide by zero
EOverflow	Floating-point operation overflowed
EUnderflow	Floating-point operation underflowed

EMCIDeviceError object

MPlayer

Declaration

```
EMCIDeviceError = class(Exception);
```

Description

The *EMCIDeviceError* exception is raised if an error occurs when accessing a multimedia device. The most common cause for the exception is trying to access a multimedia device before it has been opened with the *Open* method.

EMenuError object

Menus

Declaration

```
EMenuError = class(Exception);
```

The *EMenuError* exception is raised if an error occurs when your application is working with menu items. For example, if you application attempts to delete a menu item that doesn’t exist, a menu error exception occurs.

EMPNotify type

MPlayer

Declaration

```
EMPNotify = procedure (Sender: TObject; Button: TMPBtnType; var DoDefault: Boolean) of object;
```

The *EMPNotify* type is used for the *OnClick* event for *TMediaPlayer* components.

The *Button* argument can be one of the following values: *btBack*, *btEject*, *btNext*, *btPause*, *btPlay*, *btPrev*, *btRecord*, *btStep*, or *btStop*.

The default value of the *DoDefault* argument is *True*. If *DoDefault* is *True*, the media player control calls the method that corresponds to the clicked button. For example, if the user clicks the Play button (*btPlay*), the *Play* method is called.

If *DoDefault* is *False*, you must supply the code that executes when a media player control button is clicked in the *OnClick* event handler. The following table lists the default methods corresponding to the media player control buttons:

Control button	*Button* value	Method called
Play	*btPlay*	*Play*
Record	*btRecord*	*StartRecording*
Stop	*btStop*	*Stop*
Next	*btNext*	*Next*
Prev	*btPrev*	*Previous*
Step	*btStep*	*Step*
Back	*btBack*	*Back*
Pause	*btPause*	*Pause*
Eject	*btEject*	*Eject*

EMPPostNotify type — MPlayer

Declaration

```
EMPPostNotify = procedure (Sender: TObject; Button: TMPBtnType) of object;
```

The *EMPPostNotify* type is used for the *OnPostClick* event for *TMediaPlayer* components.

The *Button* argument can be one of the following values: *btBack*, *btEject*, *btNext*, *btPause*, *btPlay*, *btPrev*, *btRecord*, *btStep*, or *btStop*.

Empty property

Applies to

TBitmap, *TGraphic*, *TIcon*, *TMetafile* objects

Declaration

```
property Empty: Boolean;
```

Read-only. The *Empty* property specifies whether the graphics object contains a graphic. If *Empty* is *True*, no graphic has been loaded into the graphics object. If *Empty* is *False*, a graphic is contained by the graphics object.

Example

The following loads a file into *Graphic1* if it does not already contain a graphic.

```
   if Graphic1.Empty then Graphic1.LoadFromFile('myfile.bmp');
```

See also

LoadFromFile method

EmptyStr constant **SysUtils**

Declaration

```
EmptyStr: string[1] = '';
```

EmptyStr declares an empty string.

EmptyTable method

Applies to

TTable component

Declaration

```
procedure EmptyTable;
```

The *EmptyTable* method deletes all records from the database table specified by *TableName*. Before calling this method, the *DatabaseName*, *TableName* and *TableType* properties must be assigned values.

Note If the table is open, it must have been opened with the *Exclusive* property set to *True*.

Example

```
   with Table1 do
     begin
     Active := False;
     DatabaseName := ‘Delphi_Demos’;
     TableName := ‘CustInfo’;
     TableType := ttParadox;
     EmptyTable;
     end;
```

EnableControls method

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure EnableControls;
```

The *EnableControls* method restores the connections from the *dataset* to all *TDataSource* components that were disconnected by a call to the *DisableControls* method. While the data sources are disconnected, changes in the active record will not be reflected in them. The dataset maintains a count of the number of calls to *DisableControls* and *EnableControls*, so only the last call to *EnableControls* will actually update the data sources.

Example

```
with Table1 do
  begin
  DisableControls;
{ Move forward five records }
  try
    for I := 1 to 5 do Next;
  finally
{ Update the controls to the current record }
    EnableControls;
  end;
```

See also
Enabled property

Enabled property

Applies to
All controls; *TDataSource*, *TForm*, *TMenuItem*, *TTimer* components

The *Enabled* property determines if the control responds to mouse, keyboard, or timer events, or if the data-aware controls update each time the dataset they are connected to changes.

For all controls, menu items, and timers

Declaration

```
property Enabled: Boolean;
```

The *Enabled* property controls whether the control responds to mouse, keyboard, and timer events. If *Enabled* is *True*, the control responds normally. If *Enabled* is *False*, the control ignores mouse and keyboard events, and in the case of a timer control, the *OnTimer event.* Disabled controls appear dimmed.

Example
To disable a button called *FormatDiskButton*,

```
FormatDiskButton.Enabled := False;
```

This code alternately dims or enables a menu command when a user clicks the button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenCommand.Enabled then
    OpenCommand.Enabled := False
  else
    OpenCommand.Enabled := True;
end;
```

For data source components

Declaration

```
property Enabled: Boolean;
```

Description

Enabled specifies if the display in data-aware controls connected to *TDataSource* is updated when the current record in the dataset changes. For example, when *Enabled* is *True* and the *Next* method of a dataset component is called many times, each call updates all controls. Setting *Enabled* to *False* allows the *Next* calls to be made without performing updates to the controls. Once you reach the desired record, set *Enabled* to *True* to update the controls to that record.

Note Setting *Enabled* to *False* clears the display in data-aware controls until you set it to *True* again. If you want to leave the controls with their current contents while moving through the table or query, call the *DisableControls* and *EnableControls*.

Example

```
DataSource1.Enabled := False;
while not DataSource1.DataSet.EOF do DataSource1.DataSet.Next;
DataSource1.Enabled := True;
```

EnabledButtons property

Declaration

```
property EnabledButtons: TButtonSet;
```

The *EnabledButtons* property determines which buttons on the media player are enabled. An enabled button is colored and usable. A disabled button is dimmed and not usable. If a button is not enabled with *EnabledButtons*, it is disabled. By default, all buttons are enabled.

If the *AutoEnable* property is *True*, *AutoEnable* supersedes *EnabledButtons*. The buttons automatically enabled or disabled by the media player override any buttons enabled or disabled with the *EnabledButtons* property.

Button	Value	Action
Play	*btPlay*	Plays the media player
Record	*btRecord*	Starts recording
Stop	*btStop*	Stops playing or recording
Next	*btNext*	Skips to the next track, or to the end if the medium doesn't use tracks
Prev	*btPrev*	Skips to the previous track, or to the beginning if the medium doesn't use tracks
Step	*btStep*	Moves forward a number of frames
Back	*btBack*	Moves backward a number of frames
Pause	*btPause*	Pauses playing or recording. If already paused when clicked, resumes playing or recording.
Eject	*btEject*	Ejects the medium

Example
The following example enables all of the media player component's buttons:

```
TMediaPlayer1.EnabledButtons := [btPlay, btPause, btStop, btNext, btPrev, btStep, btBack,
  btRecord, btEject]
```

See also
ColoredButtons property, *VisibleButtons* property

EnableExceptionHandler procedure SysUtils

Declaration

```
procedure EnableExceptionHandler(Enable: Boolean);
```

The *EnableExceptionHandler* procedure enables or disables the standard processing of hardware exceptions or language exceptions. This requires setting notification hooks using the ToolHelp DLL. If you want to implement your own hardware exception processing, you should disable the default exception handler.

ENavClick type DBCtrls

Declaration

```
ENavClick = procedure (Sender: TObject; Button: TNavigateBtn) of object;
```

The *ENavClick* type is the type of the *OnClick* event for a database navigator component (*TDBNavigator*).

EncodeDate function SysUtils

Declaration

```
function EncodeDate(Year, Month, Day: Word): TDateTime;
```

The *EncodeDate* function returns a *TDateTime* type from the values specified as the *Year*, *Month*, and *Day* parameters.

The year must be between 1 and 9999.

Valid *Month* values are 1 through 12.

Valid *Day* values are 1 through 28, 29, 30, or 31, depending on the *Month* value. For example, the possible *Day* values for month 2 (February) are 1 through 28 or 1 through 29, depending on whether or not the *Year* value specifies a leap year.

If the specified values are not within range, an *EConvertError* exception is raised. The resulting value is one plus the number of days between 1/1/0001 and the given date.

Example

This example uses a button and a label on a form. When the user clicks the button, a specified date is encoded as a *MyDate* variable of type *TDateTime*. *MyDate* is then displayed as a string in the caption of the label.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyDate: TDateTime;
begin
  MyDate := EncodeDate(83, 12, 31);
  Label1.Caption := DateToStr(MyDate);
end;
```

See also

DateToStr function, *DecodeDate* procedure, *EncodeTime* function

EncodeTime function SysUtils

Declaration

```
function EncodeTime(Hour, Min, Sec, MSec: Word): TDateTime;
```

The *EncodeTime* function returns a *TDateTime* type from the values specified as the *Hour*, *Min*, *Sec*, and *MSec* parameters.

If the value of the *Time24Hour* typed constant is *False*, valid *Hour* values are 0 through 12. If the value of *Time24Hour* is *True*, valid *Hour* values are 0 through 23.

Valid *Min* and *Sec* values are 0 through 59. Valid *MSec* values are 0 through 999.

If the specified values are not within range, an *EConvertError* exception is raised. The resulting value is a number between 0 (inclusive) and 1 (not inclusive) that indicates the fractional part of a day given by the specified time. The value 0 corresponds to midnight, 0.5 corresponds to noon, 0.75 corresponds to 6:00 pm, etc.

Example

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyTime: TDateTime;
  Hour, Min, Sec, MSec: Word;
begin
  MyTime := EncodeTime(0, 45, 45, 7);
  Label1.Caption := TimeToStr(MyTime);
  Time24Hour := True;
  Label2.Caption := TimeToStr(MyTime);
end;
```

See also

DecodeTime procedure, *EncodeDate* function

EndDoc method

Applies to

TPrinter object

Declaration

```
procedure EndDoc;
```

The *EndDoc* method ends the current print job and closes the text file variable. After the application calls *EndDoc*, the printer begins printing. Use *EndDoc* after successfully sending a print job to the printer. If the print job isn't successful, use the *Abort* method.

The *Close* procedure calls the *EndDoc* method.

Example

This example uses a button on a form. When the user clicks it, the event handler prints a rectangle on the printer and displays a message on the form.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with Printer do
  begin
    BeginDoc;
    Canvas.Rectangle(20, 20, 400, 300);
    EndDoc;
  end;
   Canvas.TextOut(10, 10, 'Printed');
end;
```

To use the *EndDoc* method, you must add the *Printers* unit to the **uses** clause of your unit.

See also

BeginDoc method

EndDrag method

Applies to

All controls

Declaration

```
procedure EndDrag(Drop: Boolean);
```

The *EndDrag* method stops an object from being dragged any further. If the *Drop* parameter is *True*, the object being dragged is dropped. If the *Drop* parameter is *False*, the object is not dropped and dragging is canceled.

Example

The following code cancels the dragging of *Label1* without dropping the object.

```
   Label1.EndDrag(False);
```

See also

BeginDrag method, *DragMode* property, *OnEndDrag* event

EndMargin property

Applies to

TTabSet component

Declaration

```
property EndMargin: Integer;
```

The *EndMargin* property determines how far in pixels the rightmost tab appears from the right edge of the tab set control. The default value is 5. Together with the *StartMargin* property, *EndMargin* can play a role in determining how many tabs can fit within the tab set control.

If *AutoScroll* is *True* and scroll buttons appear in the tab set control, *EndMargin* determines how far in pixels the rightmost tab appears from the left edge of the scroll buttons, rather than the edge of the tab set control.

Example

This example displays the tab set control so the tabs are no closer than 20 pixels from the edge of the tab control on the left and from the scroll buttons on the right:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  with TabSet1 do
  begin
    AutoScroll := True;
    StartMargin := 20;
    EndMargin := 20;
  end;
end;
```

See also

StartMargin property

EndPage property

Applies to

TReport component

Declaration

```
property EndPage: Word;
```

The value of the *EndPage* property specifies the last page of the report that is printed. The default value is 9999. If the report is fewer than 9999 pages and you don't change the value of *EndPage*, your entire report is printed.

Example

The following code prints only the first page of *Report1*.

```
Report1.EndPage := 1;
Report1.Run;
```

See also

PrintCopies property, *StartPage* property

EndPos property

Applies to

TMediaPlayer component

Declaration

```
property EndPos: Longint;
```

Run-time only. The *EndPos* property specifies the position within the currently loaded medium at which to stop playing or recording. *EndPos* is specified using the current time format, which is specified in the *TimeFormat* property.

The *EndPos* property affects only the next *Play* or *StartRecording* method called after setting *EndPos*. You must reset *EndPos* to affect any subsequent calls to *Play* or *StartRecording*.

Example

The following procedure begins playing the .WAV audio file from the beginning of the file to middle only.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with MediaPlayer1 do
  begin
    FileName := 'D:\WINAPPS\SOUNDS\CARTOON.WAV';
    Open;
    EndPos := TrackLength[1] div 2;
    Play;
  end;
end;
```

See also

StartPos property

EndUpdate method

Applies to

TStringList, *TStrings* objects; *TOutline* component

Declaration

```
procedure EndUpdate;
```

The *EndUpdate* method re-enables screen repainting and outline item reindexing that was turned off with the *BeginUpdate* method.

Example

BeginUpdate and *EndUpdate* should always be used in conjunction with a **try...finally** statement to ensure that *EndUpdate* is called if an exception occurs. A block that uses *BeginUpdate* and *EndUpdate* typically looks like this:

```
ListBox1.Items.BeginUpdate;
try
  ListBox1.Items.Clear;
  ListBox1.Items.Add(...);
  ...
  ListBox1.Items.Add(...);
finally
```

```
    ListBox1.Items.EndUpdate;                    { Executed even in case of an exception }
  end;
```

See also

BeginUpdate method

Eof function

System

Declaration

Typed or untyped files:

```
function Eof(var F): Boolean;
```

Text files:

```
function Eof [ (var F: Text) ]: Boolean;
```

The *Eof* function tests whether or not the current file position is the end-of-file.

F is a text file variable. If *F* is omitted, the standard file variable Input is assumed.

Eof(*F*) returns *True* if the current file position is beyond the last character of the file or if the file contains no components; otherwise, *Eof*(*F*) returns *False*.

{**$I+**} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using {**$I–**}, you must use *IOResult* to check for I/O errors.

Example

```
var
  F1, F2: TextFile;
  Ch: Char;
begin
  if OpenDialog1.Execute then begin
    AssignFile(F1, OpenDialog1.Filename);
    Reset(F1);
    if SaveDialog1.Execute then begin
      AssignFile(F2, OpenDialog1.Filename);
      Rewrite(F2);
      while not Eof(F1) do
      begin
        Read(F1, Ch);
        Write(F2, Ch);
      end;
      CloseFile(F2);
    end;
    CloseFile(F1);
  end;
end;
```

See also
Eoln function, *SeekEof* function

EOF property

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
property EOF: Boolean;
```

Run-time and read only. *EOF* is a Boolean property that indicates whether a dataset is known to be at its last row. The *EOF* property returns a value of True after:

- An application opens an empty dataset
- A call to a table's *Last* method
- A call to a table's *Next* fails because the cursor is on the last row

Example

```
Table1.First;
while not Table1.EOF do
begin
  {Do Something}
  Table1.Next;
end;
```

See also
MoveBy method

Eoln function **System**

Declaration

```
function Eoln [(var F: Text) ]: Boolean;
```

The *Eoln* function test whether the current file position is the end-of-line of a text file.

F, if specified, is a text file variable. If *F* is omitted, the standard file variable Input is assumed.

Eoln(*F*) returns *True* if the current file position is at an end-of-line or if *Eof*(*F*) is *True*; otherwise, *Eoln*(*F*) returns *False*.

{**$I+**} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using {**$I–**}, you must use *IOResult* to check for I/O errors.

Example

```
uses WinCrt;

begin
{ Tells program to wait for keyboard input }
  WriteLn(Eoln);
end;
```

See also

Eof function, *SeekEoln* function

EOutlineChange type — Outline

Declaration

```
EOutlineChange = procedure (Sender: TObject; Index: LongInt) of object;
```

EOutlineChange is the type of the events which occur when an item in a *TOutline* component is changed by being expanded (*OnExpand*) or collapsed (*OnCollapse*). The *Index* parameter specifies the *Index* property value of the changed item.

EOutlineError object — Outline

Declaration

```
EOutlineError = class(Exception);
```

The *EOutlineError* exception is raised when an error occurs as your application works with an outline component.

EOutOfMemory object — SysUtils

Declaration

```
EOutOfMemory = class(Exception);
```

The *EOutOfMemory* exception is a heap exception. It occurs when your application attempts to allocate dynamic memory, but there wasn't enough free memory in the system to complete the requested operation.

EOutOfResources object — Controls

Declaration

```
EOutOfResources = class(Exception);
```

The *EOutOfResources* exception occurs when your application attempts to create a Windows handle and Windows has no more handles to allocate.

EOverflow object SysUtils

Declaration

```
EOverflow = class(EMathError);
```

Description

The *EOverflow* exception is a floating-point math exception. It occurs when a calculated result is too large to fit within the register allocated for it and therefore, data is lost. For example, this code results in an overflow condition:

```
var
 X, Y: Single;
begin
  X := 3.3e37;
  Y := 2.4e36;
  X := X * Y;
end;
```

EPageFault object SysUtils

Declaration

```
EPageFault = class(EFault);
```

The *EPageFault* exception is a hardware fault exception. It occurs when the Windows memory manager is unable to use the Windows swap file correctly. A page fault exception indicates a serious failure in the operating environment. Your applications should encounter it rarely.

EParserError object Classes

Declaration

```
EParser = class(Exception);
```

The *EParserError* is raised when your application attempts to read from a text form and it is unable to read some part of it, due to a “syntax error.”

EPrinter object

Printers

Declaration

```
EPrinter = class(Exception);
```

The *EPrinter* exception is raised when a printing error occurs. For example, if your application attempts to print to a printer that doesn't exist, or if the print job can't be sent to the printer for some reason, a printer exception occurs.

EProcessorException object

Declaration

```
EProcessorException = class(Exception);
```

The *EProcessorException* is a hardware exception. Although the *EProcessorException* is never called by the run-time library, it provides a base from which specific hardware exceptions descend. Hardware exception handling is not compiled into DLLs, only into standalone applications. These are the descendants of *EProcessorException*:

Exception	Meaning
EFault	The base exception object from which all fault objects descend.
EGPFault	A general protect fault, which is usually caused by an uninitialized pointer or object.
EStackFault	Illegal access to the processor's stack segment.
EPageFault	The Windows memory manager was unable to correctly use the Windows swap file.
EInvalidOpCode	The processor encountered an undefined instruction. Usually this means the processor was trying to execute data or uninitialized memory.
EBreakpoint	Your application generated a breakpoint interrupt.
ESingleStep	Your application generated a single-step interrupt.

You should rarely encounter the fault exceptions, other than the general protection fault, because they represent serious failures in the operating environment. The breakpoint and single-step exceptions are usually handled by Delphi's integrated debugger.

ERangeError object

SysUtils

Declaration

```
ERangeError = class(EIntError);
```

Description

The *ERangeError* exception is an integer math exception. It occurs when an integer expression evaluates to a value that exceeds the bounds of the specified integer type to which it is assigned. For example, this code raises an *ERangeError* exception:

```
  var
  SmallNumber: Shortint;
  X, Y: Integer;
begin
  X := 100;
  Y := 75;
  SmallNumber := X * Y;
end;
```

Attempting to access an item in an array with an index value that is not within the defined array results in a range error exception. For example, this code attempts to assign a value to *Values[11]* when the highest index of the *Values* array is 10:

```
var
  Values: array{1..10] of Integer;
  I: Integer;
begin
  for I := 1 to 11 do
    Values[I] := I; { on the last loop a range error exception is raised }
end;
```

The *ERangeError* exception is raised only if range checking is turned on (your code includes the **$R+** directive or you set the Range-checking option using the Options | Project dialog box).

Erase procedure **System**

Declaration

```
procedure Erase(var F);
```

The *Erase* procedure deletes the external file associated with *F*.

F is a file variable of any file type.

Always close a file before erasing it.

{**$I+**} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using {**$I–**}, you must use *IOResult* to check for I/O errors.

Example

```
procedure TForm1.Button1Click(Sender: TObject);
var
  F: Textfile;
begin
  OpenDialog1.Title := 'Delete File';
  if OpenDialog1.Execute then begin
    AssignFile(F, OpenDialog1.FileName);
    try
      Reset(F);
      if MessageDlg('Erase ' + OpenDialog1.FileName + '?',
        mtConfirmation, [mbYes, mbNo], 0) = mrYes then
      begin
        CloseFile(F);
        Erase(F);
      end;
    except
      on EInOutError do
        MessageDlg('File I/O error.', mtError, [mbOk], 0);
    end;
  end;
end;
```

See also
Rename procedure

EraseSection method

Applies to
TIniFile object

Declaration

```
procedure EraseSection(const Section: string);
```

The *EraseSection* method erases an entire section of an .INI file.

The *Section* constant identifies the section of the .INI file to erase. For example, the WIN.INI for Windows contains a [Desktop] section.

Example
This examples erases the SaveSettings section in the MYAPP.INI file when the user clicks the button on the form:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyAppIni: TIniFile;
begin
  MyAppIni := TIniFile.Create('MYAPPINI.INI');
  MyAppIni.EraseSection('SaveSettings');
  MyAppIni.Free;
end;
```

See also
ReadSection method

EReadError object

Classes

Declaration

```
EReadError = class(EFilerError);
```

The *EReadError* is raised when your application attempts to read data from a stream by calling the *ReadBuffer* method, but the number of bytes specified in the *Count* parameter of the method cannot be read.

A read error exception can also occur if Delphi is unable to read a property.

EReportError object

Report

Declaration

```
EReportError = class(Exception);
```

The *EReportError* exception is raised when the *Connect* method of a report component (*TReport*) cannot connect the report to a database because the specified server is invalid.

EResNotFound object

Classes

Declaration

```
EResNotFound = class(Exception);
```

The *EResNotFound* exception is raised when the *ReadComponentRes* method cannot find the name of the specified resource in the resource file.

Error property

Applies to

TMediaPlayer component

Declaration

```
property Error: Longint;
```

Run-time and read only. The *Error* property specifies the MCI error code returned by the most recent media control method (*Back*, *Close*, *Eject*, *Next*, *Open*, *Pause*, *PauseOnly*, *Play*, *Previous*, *StartRecording*, *Resume*, *Rewind*, *Step*, or *Stop*).

The error codes returned by media control methods are the same error codes returned by the *mciSendCommand* function, which is documented in MMSYSTEM.HLP. The message describing the error code is stored in the *ErrorMessage* property.

The value of *Error* is zero if the most recent media control method didn't cause an error. If a method results in an error, a value other than zero is stored in *Error*. If the error occurs during the opening of the device, an *EMCIDeviceError* exception occurs.

Example

The following code opens, closes, then plays *MediaPlayer1*. If an error occurs, a message window displays the error number.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
var
  MyErrorString: String;
begin
  MediaPlayer1.Open;
  MediaPlayer1.Close;
  MediaPlayer1.Play;
  MyErrorString := 'ErrorCode: ' + IntToStr(Error);
  MessageDlg(MyErrorString, mtError, [mbOk], 0);
end;
```

ErrorAddr variable — System

Declaration

```
var ErrorAddr: Pointer;
```

The *ErrorAddr* variable contains the address of the statement causing a run-time error.

If a program terminates normally or stops due to a call to *Halt*, *ErrorAddr* is **nil.**

If a program ends because of a run-time error, *ErrorAddr* contains the address of the statement in error.

See also

ExitCode variable, *ExitProc* variable

ErrorMessage property

Applies to

TMediaPlayer component

Declaration

```
property ErrorMessage: String;
```

Run-time and read only. The *ErrorMessage* property specifies the error message that describes the error code returned from the most recent media control method (*Back, Close, Eject, Next, Open, Pause, PauseOnly, Play, Previous, StartRecording, Resume, Rewind, Step,* or *Stop*). The error code described by the message is stored in the *Error* property.

Example

The following code opens *MediaPlayer1*. If an exception occurs, a message window displays the error number and string.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
var
  MyErrorString: String;
begin
  try
    MediaPlayer1.Open;
  except
    MyErrorString := 'ErrorCode: ' + IntToStr(Error) + #13#10;
    MessageDlg(MyErrorString + MediaPlayer1.ErrorMessage, mtError, [mbOk], 0);
  end;
end;
```

ErrorProc typed constant — System

Declaration

```
const ErrorProc: Pointer = nil;
```

ErrorProc is a procedure variable pointing to the RTL run-time error handler. The standard RTL *ErrorProc* reports the run-time error and terminates the program. However, if you use *SysUtils* in your program, it will force *ErrorProc* to its own routine and convert the run-time error into an exception.

ESingleStep object — SysUtils

Declaration

```
ESingleStep = class(EProcessorException);
```

The *ESingleStep* exception is a hardware exception. It occurs when your application generates a single-step interrupt. Usually Delphi's integrated debugger handles single-step exceptions.

EStackFault object — SysUtils

Declaration

```
EStackFault = class(EFault);
```

Description

The *EStackFault* exception is a hardware fault exception. It occurs when an illegal attempt to access the processor's stack is made. Usually a stack fault represents a serious failure in the operating environment.

If you have stack checking turned on {**$S+**}, you are not likely to reach a point where a stack fault occurs because each procedure or function call checks to be sure there is enough stack space for local variables before it runs. If stack checking is off {**$S-**}, this checking does not occur, and the stack fault exception could be raised.

EStreamError object — Classes

Declaration

```
EStreamError = class(Exception);
```

The *EStreamError* exception is raised when an error occurs when a stream is read with the *LoadFromStream* method. It also is the parent of these two stream exceptions:

Exception	Description
EFCreateError	An error occurred while creating a file
EFOpenError	An error occurred while opening a file

EStringListError object — Classes

Declaration

```
EStringListError = class(Exception);
```

The *EStringListError* exception occurs when an error is made in a string list object. String list error exceptions commonly occur when your application refers to an item in a list that is out of the string list's range.

EUnderflow object — SysUtils

Declaration

```
EUnderflow = class(EMathError);
```

The *EUnderflow* exception is a floating-point math exception. It occurs when the result of a calculation is too small to be represented in the size register allocated for it. For example, a 16-bit precision result that has a significant digit only in the 16th bit would underflow a register that was expecting only a *Byte* value.

EWriteError object — Classes

Declaration

```
EWriteError = class(EFilerError);
```

The *EWriteError* exception is raised when the *WriteBuffer* method of a stream object is unable to write the number of bytes specified in its *Count* parameter.

Exception object SysUtils

Declaration

```
Exception = class(TObject)
  private
    FMessage: PString;
    FHelpContext: Longint;
    function GetMessage: string;
    procedure SetMessage(const Value: string);
  public
    constructor Create(const Msg: string);
    constructor CreateFmt(const Msg: string; const Args: array of const);
    constructor CreateRes(Ident: Word);
    constructor CreateResFmt(Ident: Word; const Args: array of const);
    constructor CreateHelp(const Msg: string; AHelpContext: Longint);
    constructor CreateFmtHelp(const Msg: string; const Args: array of const;
      AHelpContext: Longint);
    constructor CreateResHelp(Ident: Word; AHelpContext: Longint);
    constructor CreateResFmtHelp(Ident: Word; const Args: array of const;
      AHelpContext: Longint);
    destructor Destroy; override;
    property HelpContext: Longint
    property Message: string;
    property MessagePtr: PString;
  end;
```

The *Exception* object is the base class for all exceptions. Therefore, all exceptions inherit the methods and properties declared within *Exception*.

The *Message* property is the message displayed when the exception occurs.

The *CreateFmt* method allows you to create a formatted message as the value of *Message*. The *Msg* constant is the string you specify, and the *Args* constant is an array of format specifiers used to format the message. *CreateFmt* uses the *Format* function to format the message.

The *CreateRes* method obtains the string that becomes the value of the *Message* property from a resource file. Specify the string as the value of the *Ident* parameter.

The *CreateResFmt* method obtains the string that becomes the value of the *Message* property and formats it using the *Format* function.

The *CreateHelp* method creates an exception object with an help context ID number.

The *CreateFmtHelp* method allows you to create a formatted message as the value of *Message* with a context-sensitive ID help number. The *Msg* constant is the string you specify, and the *Args* constant is an array of format specifiers used to format the message. The *AHelpContext* parameter is the context- sensitive help ID number. *CreateFmt* uses the *Format* function to format the message.

The *CreateResHelp* method obtains the string that becomes the value of the *Message* property from a resource file. Specify the string as the value of the *Ident* parameter. The *AHelpContext* parameter is for a context-sensitive ID number.

The *CreateResFmtHelp* method obtains the string that becomes the value of the *Message* property and formats it using the *Format* function. The *AHelpContext* parameter is for a context-sensitive ID number.

Exchange method

Applies to
TList, *TStringList*, *TStrings* objects

Declaration

```
procedure Exchange(Index1, Index2: Integer);
```

The *Exchange* method exchanges the position of two items in the list of a list object, or in the list of strings of a string or string list object. The items are specified with their index values in the *Index1* and *Index2* parameters. Because the indexes are zero-based, the first item in the list has an index value of 0,the second item has an index value of 1, and so on.

If a string in a string or string list object has an associated object, *Exchange* changes the position of both the string and the object.

Example
This example uses a list box that contains several strings as the value of the *Items* property, and a button. When the user clicks the button, the second and third items in the list box switch places in the list box.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Exchange(1, 2);
end;
```

See also
Add method, *AddStrings* method, *Delete* method, *IndexOf* method, *Insert* method, *Move* method

Exclude procedure System

Declaration

```
procedure Exclude(var S: set of T;I:T);
```

The *Exclude* procedure removes element I from set *S*.

S is a set type variable, and *I* is an expression of a type compatible with the base type of *S*.

The construct *Exclude (S, I)* corresponds to *S := S – (I)* but the *Exclude* procedure generates more efficient code.

See also

Include procedure

Exclusive property

Applies to

TTable component

Declaration

```
property Exclusive: Boolean;
```

Set the *Exclusive* property to *True* to prevent any other user from accessing the table. If other users are accessing the table when you try to open it, your exception handler will have to wait for those users to release it. If you do not provide an exception handler and another user already has the table open, your application will be terminated.

Note Set the *Active* property to *False* before changing *Exclusive* to prevent an exception.

Do not set *Active* and *Exclusive* to *True* in the Object Inspector Window. Since the Object Inspector will have the table open, that will prevent your program from opening it.

Set *Exclusive* to *True* only when you must have complete control over the table.

Example

```
{ Try to open Table1 with Exclusive True }
{ First, close Table1 }
Table1.Active := False;
repeat { until successful or Cancel button is pressed }
  try
    Table1.Exclusive := True; { See if it will open }
    Table1.Active := True;
    Break; { If no error, exit the loop }
  except
    on EDatabaseError do
      { Ask if it is OK to retry }
      if MessageDlg('Could not open Table1 exclusively - OK to retry?', mtError,
        [mbOK, mbCancel], 0) <> mrOK then raise; { If not, reraise to abort }
      { Otherwise resume the repeat loop }
  end;
  until False;
```

ExecProc method

Applies to

TStoredProc component

Declaration

```
procedure ExecProc;
```

The *ExecProc* method executes the stored procedure on the server.

Example

```
{ Execute the stored procedure }
StoredProc1.ExecProc;
```

ExecSQL method

Applies to

TQuery component

Declaration

```
procedure ExecSQL;
```

Use the *ExecSQL* method to execute an SQL statement assigned to the *SQL* property of a *TQuery* if the statement does not return a result set. If the SQL statement is an INSERT, UPDATE, DELETE, or any DDL statement, then use this method.

If the SQL statement is a SELECT statement, use *Open* instead.

Example

```
Query1.Close;
Query1.SQL.Clear;
Query1.SQL.Add('Delete from Country where Name = 'Argentina');
Query1.ExecSQL;
```

Execute method

Applies to

TBatchMove, *TColorDialog*, *TFindDialog*, *TFontDialog*, *TOpenDialog*, *TPrintDialog*, *TPrinterSetupDialog*, *TReplaceDialog*, *TSaveDialog* components

For Color, Font, Open, Save, Print, Find, and Replace dialog boxes

Declaration

```
function Execute: Boolean;
```

The *Execute* method displays the dialog box in the application and returns *True* when it is displayed. This allows your code to determine whether the user has displayed and used the dialog box by choosing its OK button.

Example

This example uses a main menu component, a memo, an Open dialog box, and a Save dialog box on a form. To use it, you need to create a File menu that includes an Open command. This code is an event handler for the *OnClick* event of the Open command on the File menu. If the user has selected a file name by choosing the Open dialog box's OK button, the code sets the Save dialog box *Filename* property to the same file name, and displays the selected file name as the caption of the form.

```
procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    Memo1.Lines.LoadfromFile(OpenDialog1.FileName);
    SaveDialog1.Filename := OpenDialog1.FileName;
    Caption := OpenDialog1.FileName;
  end;
end;
```

For Printer Setup dialog boxes

Declaration

```
procedure Execute;
```

The *Execute* method displays the Printer Setup dialog box.

Example

This code displays the Printer Setup dialog box when the user clicks the button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  PrinterSetupDialog1.Execute;
end;
```

For batch move components

Declaration

```
procedure Execute;
```

The *Execute* method performs the batch move operation specified by *Mode* from the *Source* table to the *Destination* table.

Example

```
BatchMove1.Execute;
```

ExecuteMacro method

Applies to

TDDEClientConv component

Declaration

```
function ExecuteMacro(Cmd: PChar; WaitFlg: Boolean): Boolean;
```

The *ExecuteMacro* method attempts to send a macro command string to a DDE server application. *ExecuteMacro* returns *True* if the macro was successfully passed to the DDE server application. If *ExecuteMacro* was unable to send a command string, *ExecuteMacro* returns *False*.

Cmd is a null-terminated string that contains the macro to be executed by the DDE server application. The actual value of *Cmd* depends on the DDE server application. See the documentation of the DDE server application for the command strings it will accept.

WaitFlg determines if your application should wait until the DDE server application finishes executing the macro before allowing another successful call to *ExecuteMacro* or the *ExecuteMacroLines*, *PokeData*, or *PokeDataLines* methods. If *WaitFlg* is set to *True*, subsequent calls to these methods before the DDE server application completes the first macro do not send data to the DDE server and return *False*. If *WaitFlg* is set to *False*, subsequent calls to these methods before the DDE server application completes the first macro do attempt to send data to the DDE server.

If you need to send a macro command string list rather than a single string, use the *ExecuteMacroLines* method.

Note Depending on the DDE server, attempting to execute a macro or poke data before the DDE server application completes the first macro might cause the first macro to execute unsuccessfully or produce unpredictable results. See the documentation of the DDE server application for the results of sending command strings or poking data before macro execution has completed.

Example

The following code executes the macro that is specified by the *Text* of *Edit1*. The macro sets *WaitFlg* to *True* to wait until the server has completed macro execution.

```
var
  TheMacro: PChar;
begin
  StrPCopy(TheMacro, Edit1.Text);
```

```
    DDEClientConv1.ExecuteMacro(TheMacro, True);
  end;
```

See also

StrPCopy function

ExecuteMacroLines method

Applies to

TDDEClientConv component

Declaration

```
function ExecuteMacroLines(Cmd: TStrings; WaitFlg: Boolean): Boolean;
```

The *ExecuteMacroLines* method attempts to send a macro command string list to a DDE server application. *ExecuteMacroLines* returns *True* if the macro was successfully passed to the DDE server application. If *ExecuteMacroLines* was unable to send a command string list, *ExecuteMacroLines* returns *False*.

Cmd contains the macro to be executed by the DDE server application. *WaitFlg* determines if your application should wait until the DDE server application finishes executing the macro before allowing another successful call to *ExecuteMacroLines* or the *ExecuteMacro*, *PokeData*, or *PokeDataLines* methods.

Use *ExecuteMacroLines* to execute a macro command string list rather than a single macro command string (which is what the *ExecuteMacro* method passes for its *Cmd* parameter).

Example

The following code executes the macro that exists in the *Lines* of *Memo1*. *Wait* is a boolean variable that specifies whether to wait for the server to complete macro processing before sending more data to the server.

```
  DDEClientConv1.ExecuteMacroLines(Memo1.Lines, Wait);
```

ExeName property

Applies to

TApplication component

Declaration

```
property ExeName: string;
```

Run-time and read only. The *ExeName* property contains the name of the executable application including path information. The name of the application is the name you gave the project file with an .EXE extension. If you haven’t specified a name, the default name is PROJECT1.EXE.

Example

This code displays the current name of the application's .EXE file in a label control when the user clicks the button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := Application.ExeName;
end;
```

For example, if the application name is C:\DELPHI\WORK\MYAPP.EXE, that entire string appears in the label control.

See also

Title property

ExceptionClass typed constant — System

Declaration

```
const ExceptionClass: TClass = nil;
```

ExceptionClass is a class reference variable that determines what exception classes will be reported by the debugger. *ExceptionClass* is set to *Exception* by default, so only objects descended from *Exception* and raised in the *Raise* statement will be reported by the debugger during a debug session.

ExceptProc typed constant — System

Declaration

```
const ExceptProc: Pointer = nil;
```

ExceptProc is a pointer that points to the lowest-level RTL exception handler. Unhandled exceptions are handled by *ExceptProc*. You can hook into *ExceptProc* to change how unhandled exceptions are reported, much like hooking into *ExitProc*.

Exit procedure — System

Declaration

```
procedure Exit;
```

The *Exit* procedure immediately passes control away from the current block.

If the current block is the main program, *Exit* causes the program to terminate.

If the current block is nested, *Exit* causes the next outer block to continue with the statement immediately after the statement that passed control to the nested block.

If the current block is a procedure or function, *Exit* causes the calling block to continue with the statement after the point which the block was called.

Example

```
uses WinCrt;

procedure TForm1.Button1Click(Sender: TObject);
begin
  repeat
    if Keypressed then Exit;
    Write('Xx');
  until False;
end;
```

See also

Halt procedure

ExitCode variable System

Declaration

```
var ExitCode: Integer;
```

The *ExitCode* variable contains the application's exit code.

An exit procedure can learn the cause of termination by examining *ExitCode*.

If the program terminates normally, *ExitCode* is zero.

If the program terminates due to a call to *Halt*, *ExitCode* contains the value passed to *Halt*.

If the program terminates due to a run-time error, *ExitCode* contains the error code.

See also

ErrorAddr variable, *ExitProc* variable

ExitProc variable System

Declaration

```
var ExitProc: Pointer;
```

The *ExitProc* pointer variable enables you to install an exit procedure. The exit procedure always gets called as part of a program's termination.

An exit procedure takes no parameters and must be compiled with a **far** procedure directive to force it to use the **far** call model.

When implemented properly, an exit procedure actually becomes part of a chain of exit procedures. The procedures on the exit chain get executed in reverse order of installation.

To keep the exit chain intact, you need to save the current contents of *ExitProc* before changing it to the address of your own exit procedure.

The first statement in your exit procedure must reinstall the saved value of *ExitProc*.

See also
ErrorAddr variable, *ExitCode* variable

Exp function

System

Declaration

```
function Exp(X: Real): Real;
```

Return Value

The *Exp* function returns the exponential of *X*.

The return value is e raised to the power of *X*, where e is the base of the natural logarithms.

Example

```
 var
  S: string;
begin
  S := 'e = ' + IntToStr(Exp(1.0));
  TextOut(10, 10, S);
end;
```

See also
Ln function

Expand method

Applies to
TList, *TOutlineNode* objects

For lists

Declaration

```
function Expand: TList;
```

The *Expand* method increases the maximum size of the list maintained by a list object, creating more space to add new list items and incrementing the value of the *Capacity property*. If the value of the *Capacity* property is greater than 8, the *Expand* method increases the capacity of the list by 16. If the value of *Capacity* is greater than 4, but less than 9, then the capacity of the list increases by 8. Finally if the value of *Capacity* is less than 4, then the capacity of the list grows by 4.

The returned value is the expanded list.

Example

The following code expands *List1*.

```
List1.Expand;
```

See also

Capacity property

For outline nodes

Declaration

```
procedure Expand;
```

The *Expand* method expands an outline item by assigning *True* to its *Expanded* property. When an outline item is expanded, its sub-items are displayed and the minus picture or open picture might be displayed, depending on the outline style specified in the *OutlineStyle* property of the *TOutline* component.

Example

The following code expands the first child of the first outline item, if it has children.

```
with Outline1.Items[1] do
  if HasItems then
    Outline1.Items[GetFirstChild].Expand;
```

See also

Collapse method, *FullCollapse* method, *FullExpand* method, *PictureMinus* property, *PictureOpen* property

Expanded property

Applies to

TOutlineNode object

Declaration

```
property Expanded: Boolean;
```

Run-time only. The *Expanded* property specifies whether the outline item is expanded or not. When an outline item is expanded, its subitems are displayed and the minus picture or open picture might be displayed, depending on the outline style specified in the *OutlineStyle* property of the *TOutline* component.

Expanded is *True* if the item is expanded, *False* if it isn't expanded.

Example

The following code toggles the state of the selected outline item.

```
with Outline1 do
  Items[SelectedItem].Expanded := not Items[SelectedItem].Expanded;
```

See also

Collapse method, *Expand* method, *FullCollapse* method, *FullExpand* method, *PictureMinus* property, *PictureOpen* property

ExpandFileName function SysUtils

Declaration

```
function ExpandFileName(const FileName: string): string;
```

The *ExpandFileName* function returns a string containing a fully qualified path name for the file passed in the *FileName*. A fully qualified path name includes the drive letter and any directory and subdirectories in addition to the file name and extension.

Example

The following code converts a file name into a fully-expanded file name:

```
MyFileName := ExpandFileName(MyFileName);
```

See also

ExtractFileName function

Expression property

Applies to

TIndexDef object

Declaration

```
property Expression: string;
```

Run-time and read only. Read expressions in dBASE indexes.

ExtendedSelect property

Applies to

TListBox component

Declaration

```
property ExtendedSelect: Boolean;
```

The *ExtendedSelect* property determines if the user can select an range of items in the list box. *ExtendedSelect* works in conjunction with the *MultiSelect* property. If *MultiSelect* is *False*, the setting of *ExtendedSelect* has no effect as the user will not be able to select more than one item at a time in the list box.

If *MultiSelect* is *True* and *ExtendedSelect* is *True*, the user can select an item then hold down the *Shift* key and select another and all the items in between the two selected items also become selected. If the user doesn't hold down the *Shift* or *Ctrl* key while selecting a second item, the first selected item becomes unselected—in other words, the user must use the *Ctrl* key to select multiple noncontiguous items, or the *Shift* key to select a range of items. If *ExtendedSelect* is *False*, the user can select multiple items without using the *Shift* or *Ctrl* key, but they can't select a range of items in one operation.

See also

MultiSelect property

ExtractFileExt function **SysUtils**

Declaration

```
function ExtractFileExt(const FileName string): string;
```

The *ExtractFileExt* function takes a fully qualified *FileName* and returns a string containing the three-character extension.

Example

The following code returns the extension from a file name:

```
  MyFilesExtension := ExtractFileExt(MyFileName);
```

See also

ExtractFileName function

ExtractFileName function **SysUtils**

Declaration

```
function ExtractFileName(const FileName: string): string;
```

The *ExtractFileName* function takes a fully or partially qualified path name in *FileName* and returns a string containing only the file name part, including the name and extension.

Example

The following code changes the caption of *Form1* to read "Editing <FileName>".

```
Form1.Caption := 'Editing '+ ExtractFileName(FileName);
```

See also

ExpandFileName function, *ExtractFilePath function*

ExtractFilePath function SysUtils

Declaration

```
function ExtractFilePath(const FileName: string): string;
```

The *ExtractFilePath* function takes a fully or partially qualified path name in *FileName* and returns a string containing only the path part (drive letter and directories).

Example

The following code changes the current directory to the location of *FileName*.

```
ChDir(ExtractFilePath(FileName));
```

See also

ExtractFileName function

ExceptObject function SysUtils

Declaration

```
function ExceptObject: TObject;
```

The *ExceptObject* function returns a reference to the current exception object — that is, the object associated with the currently raised exception. If there is no current exception, *ExceptObject* returns **nil**. In most cases, you do not need to call *ExceptObject* explicitly; instead, you can use the language construct

```
on E: ExceptionType do
```

This constructs maps the identifier *E* onto the object instance of the current exception statement that follows if the current exception is of *ExceptionType*. However, if you create a default exception handler by using an **else** in your exception block, the only way to access the current exception object is by calling *ExceptObject*.

ExceptAddr function — SysUtils

Declaration

```
function ExceptAddr: Pointer;
```

The *ExceptAddr* function returns the address at which the current exception was raised. If there is no current exception, *ExceptAddr* returns **nil**.

EZeroDivide object — SysUtils

Declaration

```
EZeroDivide = class(EMathError);
```

Description

The *EZeroDivide* exception is a floating-point math exception. It occurs when your application attempts to divide a floating-point value by zero. For example, this code raises a *EZeroDivide* exception:

```
var
  X, Y: Double;
begin
  X := 0.0;
  Y := 10.11111;
  Y := Y / X;
end;
```

Fail procedure — System

Declaration

```
procedure Fail;
```

The *Fail* procedure called from within a constructor causes the constructor to deallocate a dynamic object it has just allocated.

Fail should be called only if one of the constructor operations fails. However, a better way to handle a failed constructor operation is to use exception handling; see the Help system for more information.

See also

New procedure

Field property

Applies to
TDBCheckBox, *TDBComboBox*, *TDBEdit*, *TDBImage*, *TDBListBox*, *TDBMemo*, *TDBRadioGroup*, *TDBText* components

Declaration

```
property Field: TField;
```

Read and run-time only. The *Field* property returns the *TField* object the data-aware control is linked to. Use the *Field* object when you want to change the value of the data in the field programmatically.

FieldByName method

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
function FieldByName(const FieldName: string): TField;
```

The *FieldByName* method returns the *TField* with the name passed as the argument in *FieldName*. Using *FieldByName* protects your application from a change in the order of the fields in the dataset. If the field can not be found, *FieldByName* raises an exception. If you are not certain whether a field with the requested name exists, use the *FindField* method.

Example

```
with Table1 do
  begin
  { This is the safe way to change 'CustNo' field }
    FieldByName('CustNo').AsString := '1234';
  { This is *not* the safe way to change 'CustNo' field }
    Fields[0].AsString := '1234';
  end;
```

FieldClass property

Applies to
TFieldDef object

Declaration

```
property FieldClass: TFieldClass;
```

Run-time and read only. Read *FieldClass* to determine the type of the *TField* component that corresponds to this *TFieldDef object*.

FieldCount property

Applies to
TDBGrid, *TDBLookupList*, *TQuery*, *TStoredProc*, *TTable* components

Declaration

```
property FieldCount: Integer;
```

Run-time and read only. The *FieldCount* property specifies the number of fields (columns) in a dataset. It may not be the same as the number of fields in the underlying database table, since you can add calculated fields and remove fields with the Fields Designer.

For the data grid and database lookup list box, the value of the *FieldCount* property is the number of fields in the dataset displayed in the control.

Example

The following code displays the number of fields in *DBGrid1* in a label.

```
   Label1.Caption := IntToStr(DBGrid1.FieldCount);
```

See also
Fields property, *SelectedField* property

FieldDefs property

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
property FieldDefs: TFieldDefs;
```

Run-time only. The *FieldDefs* property holds information about each *TFieldDef* in the *dataset*. You can use this property to determine which fields are in the dataset, their name, type, and size.

See also
Fields property, *TField* component

FieldName property

Applies to

TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
property FieldName: string;
```

FieldName is the name of the physical column in the underlying dataset to which a *TField* component is bound. *FieldName* is used as a default column heading by the data grid when the *DisplayLabel* property is null. For calculated fields, supply a *FieldName* when you define the field. For non-calculated fields, an exception occurs if a *FieldName* is not a column name in the physical table.

See also

DisplayName property

FieldNo property

Applies to

TFieldDef object; *TBCDField*, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

For TFieldDef objects

Declaration

```
property FieldNo: Integer;
```

Run-time and read only. *FieldNo* is the physical field number used by the Borland Database Engine to reference the field.

Example

```
   { Display the field name and number }
   with FieldDef1 do
     MessageDlg(Name + ' is field ' + IntToStr(FieldNo), mtInformation, [mbOK], 0);
```

See also

TField component

For fields

Declaration

```
property FieldNo: Integer;
```

Run-time and read only. *FieldNo* is the ordinal of the *TField* component in its dataset. This property is available for programs that make direct calls to the Borland Database Engine.

Fields property

Applies to
TIndexDef object; *TDBGrid*, *TDBLookupList*, *TQuery*, *TStoredProc*, *TTable* components

For grids, lookup lists, queries, stored procedures, and tables

Declaration

```
property Fields[Index: Integer]: TField;
```

Run-time and read only. The *Fields* property returns a specific field in the dataset. Specify the field using the *Index* parameter, with the first field in the dataset having an *Index* value of 0.

Example
The following code left justifies the first field in *DBGrid1*.

```
   DBGrid1.Fields[0].Alignment := taLeftJustify;
```

See also
FieldCount property, *FieldDefs property*, *SelectedField* property, *SelectedIndex* property

For index definitions

Declaration

```
property Fields: string;
```

Run-time and read only. *Fields* is a string consisting of the names or numbers of the fields comprising the index, separated by semicolons (“;”). When numbers are used, they are the physical field numbers in the table; for example, 1..N.

File mode constants SysUtils

Declaration

```
fmClosed = $D7B0;
fmInput  = $D7B1;
fmOutput = $D7B2;
fmInOut  = $D7B3;
```

Use the file mode constants when opening and closing disk files. The *Mode* field of *TFileRec* and *TTextRec* will contain one of these values.

File open mode constants SysUtils

Declaration

```
fmOpenRead       = $0000;
fmOpenWrite      = $0001;
fmOpenReadWrite  = $0002;
fmShareCompat    = $0000;
fmShareExclusive = $0010;
fmShareDenyWrite = $0020;
fmShareDenyRead  = $0030;
fmShareDenyNone  = $0040;
```

The file open mode constants are used to control the shareability of a file or stream when you open it.

TFileStream.Create has a *Mode* parameter that you can set to one of these constants:

Constant	Definition
fmOpenRead	Open for read access only.
fmOpenWrite	Open for write access only.
fmOpenReadWrite	Open for read and write access.
fmShareCompat	Compatible with the way FCBs are opened.
fmShareExclusive	Read and write access is denied.
fmShareDenyWrite	Write access is denied.
fmShareDenyRead	Read access is denied.
fmShareDenyNone	Allows full access for others.

FileAge function SysUtils

Declaration

```
function FileAge(const FileName: string): Longint;
```

The *FileAge* function returns the age of the file named by *FileName* as a *Longint*.

FileClose procedure **SysUtils**

Declaration

```
procedure FileClose(Handle: Integer);
```

The *FileClose* procedure closes the specified file.

The *FileClose* routine exists to prevent a name conflict between the standard *Close* procedure and the *Close* method of an object.

Example

The following code closes a file opened with *FileOpen*:

```
FileClose(MyFileHandle);
```

See also

FileCreate function, *FileOpen* procedure

FileCreate function **SysUtils**

Declaration

```
function FileCreate(const FileName: string): Integer;
```

FileCreate creates a new file by the specified name. If the return value is positive, the function was successful, and the value is the file handle of the new file. If the return value is negative, an error occurred, and the value is a negative DOS error code.

Example

The following example creates a new file and assigns it to the identifier *MyFileHandle*.

```
MyFileHandle := FileCreate('NEWFILE.TXT');
```

See also

FileClose procedure, *FileOpen* procedure

FileEdit property

Applies to

TFileListBox component

Declaration

```
property FileEdit: TEdit;
```

The *FileEdit* property provides a simple way to display a file selected in a file list box as the text of an edit box, as is commonly done in Open and Save dialog boxes. If no file is selected in the file list box, the text of the edit box is the current value of the file list box's *Mask* property.

Specify the edit box you want the mask or selected file to appear in as the value of the *FileEdit* property.

Example

This example uses a button, an edit box, a label, a drive combo box, a directory list box, a file list box, and a filter combo box on a form. When the user clicks the button, the rest of the controls of the form begin working together like the controls in an Open or Save dialog box.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  DriveComboBox1.DirList := DirectoryListBox1;
  DirectoryListBox1.FileList := FileListBox1;
  DirectoryListBox1.DirLabel := Label1;
  FileListBox1.FileEdit := Edit1;
  FilterComboBox1.FileList := FileListBox1;
end;
```

See also

DirLabel property, *DirList* property, *FileList* property, *Mask* property, *Text* property

FileEditStyle property

Applies to

TOpenDialog, *TSaveDialog* components

Declaration

```
property FileEditStyle: TFileEditStyle;
```

The *FileEditStyle* property determines if the Open or Save dialog box contains an edit box or combo box control for the user to enter a file name. These are the possible values:

Value	Meaning
fsEdit	Edit box to enter a file name.
fsComboBox	Drop-down combo box to enter a file name. The combo box can be used to display a list of file names.

The default value is *fsEdit*.

If the *FileEditStyle* is *fsComboBox*, you can specify which files names appear in the combo box. Use the *List* property to enter a list of file names, either during design time with the Object Inspector, or at run time.

Your application can also keep a history list for the combo box, a list of previous file names the user has entered. To implement a history list, follow these suggested steps:

1 Add a *TStringList* object to your application to keep the list of file names the user enters.

2 Before your application calls the *Execute* method to display the Open or Save dialog box, assign the *TStringList* object to the *HistoryList* property. For example,

```
var
  MyHistoryList: TStringList;
begin
  OpenDialog1.HistoryList := MyHistoryList;
  if OpenDialog1.Execute then
  ...
```

3 Use the returned *FileName* property value to update your history list. For example:

```
MyHistoryList.Insert(0, OpenDialog1.FileName);
```

Example

This examples uses a Save dialog box, an edit box, and a button on a form. When the user clicks the button, the Save dialog box appears with a combo box control to allow the user to type a file name, select a file name from the list box, or drop down a list to choose a file name from a history list. For this example, no history list exists. Once the user selects a file name, the selected name appears in the edit box on the form.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 SaveDialog1.FileEditStyle := fsComboBox;
 SaveDialog1.Filter := 'Text Files(*.TXT) | *.TXT';
 if SaveDialog1.Execute then
    Edit1.Text := SaveDialog1.FileName;
end;
```

See also

HistoryList property, *TFilterComboBox* component

FileExists function — SysUtils

Declaration

```
function FileExists(const FileName: string): Boolean;
```

The *FileExists* function returns *True* if the file specified by *FileName* exists. If the file does not exist, *FileExists* returns *False*.

Example

The following code prompts you for confirmation before deleting a file:

```
if FileExists(FileName) then
  MsgBox('Do you really want to delete ' + ExtractFileName(FileName)
  + '?'), []) = IDYes then FileDelete(FileName);
```

See also
FileSearch function

FileDateToDateTime function **SysUtils**

Declaration

```
function FileDateToDateTime(FileDate: Longint): TDateTime;
```

FileDateToDateTime converts a DOS date-and-time value to a *TDateTime* value. The *FileAge*, *FileGetDate*, and *FileSetDate* routines operate on DOS date-and-time values, and the Time field of a *TSearchRec* used by the *FindFirst* and *FindNext* functions contains a DOS date-and-time value.

See also
DateTimeToFileDate function

FileGetAttr function **SysUtils**

Declaration

```
function FileGetAttr(const FileName: string): Integer;
```

FileGetAttr returns the file attributes of the file given by *FileName*. The attributes can be examined by AND-ing with the *faXXXX* constants. If the return value is negative, an error occurred and the value is a negative DOS error code.

See also
FileSetAttr function

FileGetDate function **SysUtils**

Declaration

```
function FileGetDate(Handle: Integer): Longint;
```

The *FileGetDate* function returns the date when a file was created or last modified in DOS internal format.

See also
FileSetDate procedure

FileList property

Applies to

TDirectoryListBox, *TFilterComboBox* components

Declaration

```
property FileList: TFileListBox;
```

The *FileList* property is used for two different purposes, depending on the type of control it is a property of.

For directory list boxes, *FileList* provides a simple way to connect a directory list box with a file list box. Once the two controls are connected and new directory is selected as the current directory using a directory list box, the file list box displays the files in the current directory. Specify the file list box in which you want to display the files in the directory selected in the directory list box as the value of the *FileList* property.

For filter combo boxes, *FileList* provides a simple way to connect a filter combo box with a file list box. Once the two controls are connected and a new filter is selected using a filter combo box, the file list box displays the files that match the selected filter. Specify the file list box you want to display the files matching the selected filter as the value of the *FileList* property.

Example

This example uses a button, an edit box, a label, a drive combo box, a directory list box, a file list box, and a filter combo box on a form. When the user clicks the button, the rest of the controls on the form begin working together like the controls in an open or save dialog box.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  DriveComboBox1.DirList := DirectoryListBox1;
  DirectoryListBox1.FileList := FileListBox1;
  DirectoryListBox1.DirLabel := Label1;
  FileListBox1.FileEdit := Edit1;
  FilterComboBox1.FileList := FileListBox1;
end;
```

See also

DirLabel property, *Directory* property, *DirList* property, *FileEdit* property, *FileList* property, *Filter* property, *Mask* property

FileMode variable — System

Declaration

```
var FileMode: Byte;
```

The *FileMode* variable determines the access code to pass to DOS when typed and untyped files (not text files) are opened using the *Reset* procedure.

The default *FileMode* is 2. Assigning another value to *FileMode* causes all subsequent *Resets* to use that mode.

The range of valid *FileMode* values depends on the version of DOS in use. For all versions, these modes are defined:

0	Read only
1	Write only
2	Read/Write

DOS version 3.x and later defines additional modes, which are primarily concerned with file sharing on networks.

See also
Rewrite procedure

FileName property

Applies to
TFileListBox, *TMediaPlayer*, *TOpenDialog*, *TSaveDialog* components; *TIniFile* object

For Open and Save dialog boxes

Declaration

```
property FileName: TFileName;
```

The *FileName* property specifies the file name that appears in the File Name edit box when the dialog box opens. The user can then select that file name or specify any other. Once the user specifies a file name and chooses OK, the value of the *FileName* property becomes the name of the file the user selected.

The path name can include a path. For example, to open the file README.TXT in the directory C:\TEMP, set *FileName* to C:\TEMP\README.TXT.

The *FileName* property can be set to the name of a file that doesn't exist in the current directory. In an Open dialog box, you can use this capability to let the user open a new file, and in a Save dialog box, the user can save a file that hasn't been saved before.

Example
This example displays an Open dialog box and suggests the file name LIST.PAS to the user. Once the user selects a file name, the code displays that name in a label on the form:

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

```
    OpenDialog1.FileName := 'LIST.PAS';
    if OpenDialog1.Execute then
      Label1.Caption := OpenDialog1.FileName;
  end;
```

See also

Files property

For media player components

Declaration

```
property FileName: string;
```

The *FileName* property specifies the media file to be opened by the *Open* method, or the file to save by the *Save* method. At design time, you can use a file open dialog box to specify the *FileName* property by clicking the ellipses button (...) in the Object Inspector.

Example

The following code determines what type of media device to open from the results of an Open dialog box, then opens the file.

```
  if OpenDialog1.Execute then
  begin
    MediaPlayer1.DeviceType := dtAutoSelect;
    MediaPlayer1.FileName := OpenDialog1.FileName;
    MediaPlayer1.Open;
  end;
```

For the file list boxes

Declaration

```
property FileName: string;
```

Run-time only. The *FileName* property contains the name of the selected file in the list box, including the path name.

Example

This example uses a file list box and a label on a form. When the user selects a file in the file list box, the name of the file appears as the caption of the label.

```
  procedure TForm1.FileListBox1Click(Sender: TObject);
  begin
    Label1.Caption := FileListBox1.FileName;
  end;
```

See also

FileList property

For TIniFile objects

Declaration

```
property FileName: string;
```

Run-time and read only. The *FileName* property contains the name of the .INI file the *TIniFile* object encapsulates.

FilePos function

System

Declaration

```
function FilePos(var F): Longint;
```

The *FilePos* function returns the current file position within a file.

To use *FilePos* the file must be open and it can't be used on a text file.

F is a file variable.

Position	Result
Beginning of file	FilePos(F) = 0
Middle of file	FilePos(F) = current file position
End of file	Eof(F) = True

{**$I+**} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using {**$I–**}, you must use *IOResult* to check for I/O errors.

Example

```
var
  f: file of Byte;
  size : Longint;
  S: string;
  y: integer;
 begin
  if OpenDialog1.Execute then begin
    AssignFile(f, OpenDialog1.FileName);
    Reset(f);
    size := FileSize(f);
    S := 'File size in bytes: ' + IntToStr(size);
    y := 10;
    Canvas.TextOut(5, y, S);
    y := y + Canvas.TextHeight(S) + 5;
    S := 'Seeking halfway into file...';
    Canvas.TextOut(5, y, S);
    y := y + Canvas.TextHeight(S) + 5;
```

```
        Seek(f,size div 2);
        S := 'Position is now ' + IntToStr(FilePos(f));
        Canvas.TextOut(5, y, S);
        CloseFile(f);
      end;
    end;
```

See also
FileSize function, *Seek* procedure

FileRead function SysUtils

Declaration

```
function FileRead(Handle: Integer; var Buffer; Count: Longint): Longint;
```

The *FileRead* function reads *Count* bytes from the *Handle* into the buffer. The function result is the actual number of bytes read, which may be less than *Count*.

Example
The following code fills a buffer from a file.

```
  ActualRead := FileRead(MyFileHandle, Buffer, SizeOf(Buffer));
```

See also
FileSeek function, *FileWrite function*

Files property

Applies to
TOpenDialog, *TSaveDialog* components

Declaration

```
property Files: TStrings;
```

Run-time and read only. The *Files* property value contains a list of all the file names selected in the Open or Save dialog box including the path names.

To let users select multiple file names in the dialog box, include *ofAllowMultiSelect* in the *Options* property set (set *ofAllowMultiSelect* to *True*).

The entire list of names is returned as the value of the *FileName* property. If the list of names is long, *FileName* contains only the first 127 characters.

Example

This example uses an Open dialog box, a list box, and a button on a form. When the user clicks the button, the Open dialog box appears. When the user selects files in the dialog box and chooses the OK button, the list of selected files appears in the list box.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  OpenDialog1.Options := [ofAllowMultiSelect];
  OpenDialog1.Filter := 'All files (*.*)|*.*';
  if OpenDialog1.Execute then
    ListBox1.Items := OpenDialog1.Files;
end;
```

See also

Filter property, *Options* property

FileSearch function SysUtils

Declaration

```
function FileSearch(const Name, DirList: string): string;
```

The *FileSearch* function searches through the directories passed in *DirList* for a file named *Name*. *DirList* should be in the same format as a DOS path: directory names separated by semicolons. If *FileSearch* locates a file matching *Name*, it returns a string containing a fully-qualified path name for that file. If no matching file exists, *FileSearch* returns an empty string.

Example

The following code searches for FINDME.DLL in a series of directories:

```
FoundIt := FileSearch('FINDME.DLL', MyAppDir+’\';'+WinDir+';'+WinDir+'\SYSTEM');
```

See also

FileExists function

FileSeek function SysUtils

Declaration

```
function FileSeek(Handle: Integer; Offset: Longint; Origin: Integer): Longint;
```

The *FileSeek* function positions the current file pointer within a previously opened file. *Handle* contains the file handle. *Offset* specifies the number of bytes from *Origin* where the file pointer should be positioned. *Origin* is a code with three possible values, denoting the beginning of the file, the end of the file, and the current position of the file pointer.

Origin	Action
0	The file pointer is positioned *Offset* bytes from the beginning of the file.
1	The file pointer is positioned *Offset* bytes from its current position.
2	The file pointer is positioned *Offset* bytes from the end of the file.

If *FileSeek* is successful, it returns the new position of the file pointer; otherwise, it returns the Windows constant HFILE_ERROR.

Example

The following code positions the file pointer at the end of a file:

```
if FileSeek(MyFileHandle,0,2) = HFILE_ERROR then
  HandleFileError
else
  AppendStuff;
```

See also

FileRead function, *FileWrite function*

FileSetAttr function SysUtils

Declaration

```
function FileSetAttr(const FileName: string; Attr: Integer): Integer;
```

FileSetAttr sets the file attributes of the file given by *FileName* to the value given by *Attr*. The attribute value is formed by OR-ing the appropriate *faXXXX* constants. The return value is zero if the function was successful. Otherwise the return value is a negative DOS error code.

See also

FileGetAttr function

FileSetDate procedure SysUtils

Declaration

```
procedure FileSetDate(Handle: Integer; Age: Longint);
```

FileSetDate sets the DOS date-and-time stamp of the file given by *Handle* to the value given by *Age*. The *DateTimeToFileDate* function can be used to convert a *TDateTime* value to a DOS date-and-time stamp.

See also

FileGetDate function

FileSize function

System

Declaration

```
function FileSize(var F): Longint;
```

The *FileSize* function returns the size in bytes of file *F*. However, if *F* is a record file *FileSize* will return the number of records in the file.

To use *FileSize* the file must be open and it can't be used on a text file.

F is a file variable.

If the file is empty, *FileSize*(*F*) returns 0.

{**$I+**} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using {**$I–**}, you must use *IOResult* to check for I/O errors.

Example

```
var
   f: file of Byte;
   size : Longint;
   S: string;
   y: integer;
 begin
   if OpenDialog1.Execute then begin
     AssignFile(f, OpenDialog1.FileName);
     Reset(f);
     size := FileSize(f);
     S := 'File size in bytes: ' + IntToStr(size);
     y := 10;
     Canvas.TextOut(5, y, S);
     y := y + Canvas.TextHeight(S) + 5;
     S := 'Seeking halfway into file...';
     Canvas.TextOut(5, y, S);
     y := y + Canvas.TextHeight(S) + 5;
     Seek(f,size div 2);
     S := 'Position is now ' + IntToStr(FilePos(f));
     Canvas.TextOut(5, y, S);
     CloseFile(f);
   end;
 end;
```

See also

FilePos function

FileType property

Applies to
TFileListBox component

Declaration

```
property FileType: TFileType;
```

The *FileType* property determines which files are displayed in the file list box based on the attributes of the files. Because *FileType* is of type *TFileType*, which is a set of file attributes, *FileType* can contain multiple values. For example, if the value of *FileType* is a set containing the values *ftReadOnly* and *ftHidden*, only files that have the read-only and hidden attributes are displayed in the list box. These are the values that can occur in the *FileType* property:

Value	Meaning
ftReadOnly	When *ftReadOnly* is *True*, the list box can display files with the read-only attribute.
ftHidden	When *ftHidden* is *True*, the list box can display files with the hidden attribute.
ftSystem	When *ftSystem* is *True*, the list box can display files with the system attribute.
ftVolumeID	When *ftVolumeID* is *True*, the list box can display the volume name.
ftDirectory	When *ftDirectory* is *True*, the list box can display directories.
ftArchive	When *ftArchive* is *True*, the list box can display files with archive attribute.
ftNormal	When *ftNormal* is *True*, the list box can display files with no attributes.

If you use the Object Inspector to change the value of *FileType*, click the *FileType* property to see the attribute values. Then you can set each value to *True* or *False*, which builds the *FileType* set.

Example
This example uses a file list box on a form. When the application runs, only read-only files, directories, volume IDs, and files with no attributes appear in the list box.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FileListBox1.FileType := [ftReadOnly, ftDirectory, ftVolumeID, ftNormal];
end;
```

See also
Mask property, *TFileType* type

FileWrite function SysUtils

Declaration

```
function FileWrite(Handle: Integer; const Buffer; Count: Longint): Longint;
```

This is an internal routine, and you will not need to use it.

The *FileWrite* function writes *Count* bytes from *Buffer* to the file indicated by *Handle*. The actual number of bytes written is returned. If the return value is not equal to *Count*, it is usually because the disk is full.

Example

```
if FileWrite(MyFileHandle, Buffer, SizeOf(Buffer)) <> SizeOf(Buffer) then
  ErrorMsg('Disk full while writing to file!');
```

See also

FileRead function, FileSeek function

FillChar procedure — System

Declaration

```
procedure FillChar(var X; Count: Word; value);
```

The *FillChar* procedure fills *Count* number of contiguous bytes with a specified value (can be type *Byte* or *Char*).

This function does not perform any range checking.

Example

```
 var
  S: string[80];
begin
  { Set a string to all spaces }
  FillChar(S, SizeOf(S), ' ');
  S[0] := #80;                                                  { Set length byte }
end;
```

See also

Move procedure

FillRect method

Applies to

TCanvas object

Declaration

```
procedure FillRect(const Rect: TRect);
```

The *FillRect* method fills the specified rectangle on the canvas using the current brush.

Example

This code creates a rectangle on the form's canvas and colors it red by changing the canvas *Brush* property to *clRed*.

```
procedure TForm1.ColorRectangleClick(Sender: TObject);
var
  NewRect: TRect;
begin
  NewRect := Rect(20, 30, 50, 90);
  Form1.Canvas.Brush.Color := clRed;
  Form1.Canvas.FillRect(NewRect);
end;
```

See also

Brush property, *Rect* function

Filter property

Applies to

TFilterComboBox, *TOpenDialog*, *TSaveDialog* components

For Open and Save dialog boxes

Declaration

```
property Filter: string;
```

The *Filter* property determines the file masks available to the user for use in determining which files display in the dialog box's list box.

A file mask or file filter is a file name that usually includes wildcard characters (*.PAS, for example). Only files that match the selected file filter are displayed in the dialog box's list box, and the selected file filter appears in the File Name edit box. To specify a file filter, assign a filter string as the value of *Filter*. To create the string, follow these steps:

1. Type some meaningful text that indicates the type of file.
2. Type a | character (this is the "pipe" or "or" character).
3. Type the file filter.

Don't put in any spaces around the | character in the string.

Here's an example:

```
OpenDialog1.Filter := 'Text files|*.TXT'
```

If you entered the preceding example as the Filter of an Open or Save dialog box, the string "Text files" appears in the List Files of Type drop-down list box when the dialog box appears in your application, the file filter appears in the File Name edit box, and only .TXT files appear in the list box. You can specify multiple file filters so that a list of filters appears in the List Files of Type drop-down list box or in the filter combo box.

This allows the user to select from a number of file filters and determine which files are displayed in the list box.

To specify multiple file filters,

1 Create a file filter string as previously shown.
2 Type another file filter in the same way, but separate the second file filter from the first with the | character.
3 Continue adding as many file filters as you like, separating them with the | character. The string can be up to 255 characters.

Here's an example of three file filters specified as the value of the *Filter* property:

```
'Text files (*.TXT)|*.TXT|Pascal files (*.PAS)|*.PAS|Quattro Pro files (*.WB1)|*.WB1'
```

Now when the dialog box appears, the user can choose from three file filters that appear in the List Files of Type drop-down list box.

Note that the previous example includes the file filters in parentheses in the text parts. This isn't required, but it's a common convention that helps users understand what to expect when they select a file filter.

You can string multiple wildcard file filters together if you separate them with semicolons:

```
OpenDialog1.Filter := 'All files|*.TXT;*.PAS;*.WB1';
```

Example

This code sets the value of the *Filter* property, displays the dialog box, and assigns the file name the user selects to a variable:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NameOfFile : TFileName;
begin
  OpenDialog1.Filter := 'Text files (*.TXT)|*.TXT|Pascal files (*.PAS)' +
    '|*.PAS|Quattro Pro files (*.WB1)|*.WB1';
  if OpenDialog1.Execute then
    NameOfFile := OpenDialog1.FileName;
  ...
end;
```

See also

FileName property, *FilterIndex* property

For filter combo boxes

Declaration

```
property Filter: string;
```

The *Filter* property determines the file masks displayed in the filter combo box.

A file mask or file filter is a file name that usually includes wildcard characters (*.PAS, for example). When your application runs, the file filter the user selects in the filter combo box becomes the value of the *Mask* property. To specify a file filter, assign a filter string as the value of *Filter*. To create the string, follow these steps:

1 Type some meaningful text to indicate the type of file.
2 Type a | character (this is the "pipe" or "or" character).
3 Type the file filter.

Don't put in any spaces around the | character in the string.

Here's an example:

```
FilterComboBox1.Filter := 'Text files|*.TXT';
```

If you entered this string, the string "Text files" appears in the filter combo box.

You can specify multiple file filters so that a list of filters appears in the filter combo box from which the user can select. To specify multiple file filters,

1 Type a file filter as shown previously.

2 Type another file filter in the same way, but separate the second file filter from the first with the | character.

3 Continue adding as many file filters as you like, separating them with the | character. The string can be up to 255 characters.

Here's an example of three file filters specified as the value of the *Filter* property:

```
'Text files (*.TXT)|*.TXT|Pascal files (*.PAS)|*.PAS|Quattro Pro files (*.WB1)|*.WB1'
```

Note that the previous example includes the file filters in parentheses in the text parts. This isn't required, but it's a common convention that helps users understand what to expect when they select a file filter.

You can string multiple wildcard file filters together if you separate them with semicolons:

```
FilterComboBox1.Filter := 'All files|*.TXT;*.PAS;*.WB1';
```

Examples

This example uses a filter combo box on a form. When the application runs, three filters appear in the filter combo box:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FilterComboBox1.Filter := 'Text files (*.TXT)|*.TXT|Pascal files (*.PAS)' +
  '|*.PAS|Quattro Pro files (*.WB1)|*.WB1';
end;
```

This example uses a filter combo box, a file list box, and an edit box on a form. The code connects the three controls through the *FileList* and *FileEdit* properties. When the user selects a filter in the filter combo box, the filter is applied to the files in the list box so the list box displays only the files that match the filter. The filter in effect on the file list box appears in the edit box. When the user selects a file in the file list box, the selected file appears in the edit box.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FilterComboBox1.Filter := 'All files (*.*)|*.*|Pascal files (*.pas)|' +
    '*.PAS|DLLs (*.dll)|*.DLL';
  FilterComboBox1.FileList := FileListBox1;
  FileListBox1.FileEdit := Edit1;
end;
```

See also

FileList property, *FileName* property, *Mask* property

FilterIndex property

Applies to

TOpenDialog, *TSaveDialog* components

Declaration

```
property FilterIndex: Integer;
```

The *FilterIndex* property determines which file filter specified in the *Filter property* appears as the default file filter in the List Files of Type drop-down list box. For example, if you set the *FilterIndex* value to 2, the second file filter listed in the *Filter* property becomes the default filter when the dialog box appears. The default *FilterIndex* value is 1. If you specify a value greater than the number of file filters in the *Filter* property, the first filter is chosen.

The default value is 1.

Example

This code specifies three file filters as the value of the *Filter* property, sets the *FilterIndex* to 2 so that the second file filter is the default file filter, and displays the Open dialog box. Once the user selects a file with the dialog box and chooses OK, the file name the user selected appears in a label on the form.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  OpenDialog1.Filter := 'Text files (*.TXT)|*.TXT|Pascal files (*.PAS)' +
    ‘|*.PAS|dBASE program files (*.PRG)|*.PRG';
  OpenDialog1.FilterIndex := 2;
  if OpenDialog1.Execute then
    Label1.Caption := OpenDialog1.FileName;
end;
```

See also

Filter property

Find method

Applies to
TFieldDefs, *TStringList* objects

For string list objects

Declaration

```
function Find(const S: string; var Index: Integer): Boolean;
```

The *Find* method searches for a specified string in the list of strings kept in a string list object. If the string specified as the value of the *S* parameter is found, *Find* returns *True* and the position of the string in the string list is stored as the value of the *Index* parameter. Because the index is zero-based, the first string in the string list has an index value of 0, the second string has an index value of 1, and so on.

Find returns *False* if the specified string is not found.

Example

This example uses a list box and a label on a form. When the application runs, a string list object is created and three strings are added to it. The *Find* method searches the strings to look for a match with the string 'Flowers'. If the string is found, all the strings in the string list are added to the list box, and the index value of the 'Flowers' string appears in the caption of the label control.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  MyList: TStringList;
  Index: Integer;
begin
  MyList := TStringList.Create;
  MyList.Add('Animals');
  MyList.Add('Flowers');
  MyList.Add('Cars');
  if MyList.Find('Flowers', Index) then
  begin
    ListBox1.Items.AddStrings(MyList);
    Label1.Caption := 'Flowers has an index value of ' + IntToStr(Index);
  end;
  MyList.Free;
end;
```

See also

Add method, *Clear* method, *IndexOf* method, *Strings* property

For TFieldDefs objects

Declaration

```
function Find(const Name: string): TFieldDef;
```

The *Find* method returns a pointer to an entry in the *Items* property whose *Name* property matches the *Name* parameter. Use this method to obtain information about a particular *TFieldDef* object.

Example

```
{ Display the field name and number }
MessageDlg('CustNo is field ' + IntToStr(FieldDefs.Find('CustNo').FieldNo),
  mtInformation, [mbOK], 0);
```

See also

Name property

FindClose procedure — SysUtils

Declaration

```
procedure FindClose(var SearchRec: TSearchRec);
```

FindClose terminates a *FindFirst*/*FindNext* sequence. *FindClose* does nothing in the 16-bit version of Windows, but is required in the 32-bit version, so for maximum portability every *FindFirst*/*FindNext* sequence should end with a call to *FindClose*.

See also

FindFirst function, *FindNext* function

FindComponent method

Applies to

All components

Declaration

```
function FindComponent(const AName: string): TComponent;
```

The *FindComponent* method returns the component in the *Components* array property with the name that matches the string in the *AName* parameter. *FindComponent* is not case sensitive.

Example

To set up this example, place several components on a form, including an edit box and a button. When the user clicks the button, the code displays the value of the *ComponentIndex* of the edit box in the edit box.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TheComponent: TComponent;
begin
  TheComponent := FindComponent('Edit1');
  Edit1.Text := IntToStr(TheComponent.ComponentIndex);
end;
```

See also

ComponentCount property, *ComponentIndex* property, *Components* property

FindDatabase method

Applies to

TSession component

Declaration

```
function FindDatabase(const DatabaseName: string): TDatabase;
```

The *FindDatabase* method attempts to find a *TDatabase* component in the *Databases* collection with a *DatabaseName* property which matches the *DatabaseName* parameter. If there is no such database, *FindDatabase* returns **nil**.

Example

```
MyDatabase := Session.FindDatabase(‘MYDB’);
```

See also

Session variable

FindField method

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
function FindField(const FieldName: string): TField;
```

The *FindField* method returns the field with the name passed in *FieldName*. While calling *FindField* is slightly slower than a direct reference to the *Fields* property, using *FindField* protects your application from a change in the order of the fields in the component. If the field can not be found, *FindField* returns **nil**.

Example

```
with Table1 do
  begin
{ This is the safe way to change ‘CustNo’ field }
  FindField(‘CustNo’).AsString := ‘1234’;
{ This is *not* the safe way to change ‘CustNo’ field }
  Fields[0].AsString := ‘1234’;
  end;
```

See also

FieldByName method

FindFirst function **SysUtils**

Declaration

```
function FindFirst(const Path: string; Attr: Word; var F: TSearchRec): Integer;
```

The *FindFirst* function searches the specified directory for the first entry matching the specified file name and set of attributes.

The *Path* constant parameter is the directory and file name mask, including wildcard characters. For example, 'c:\test\*.*' specifies all files in the C:\TEST directory).

The *Attr* parameter specifies the special files to include in addition to all normal files. Choose from these file attribute constants when specifying the *Attr* parameter:

Constant	Value	Description
faReadOnly	$01	Read-only files
faHidden	$02	Hidden files
faSysFile	$04	System files
faVolumeID	$08	Volume ID files
faDirectory	$10	Directory files
faArchive	$20	Archive files
faAnyFile	$3F	Any file

You can combine attributes by adding their constants or values. For example, to search for read-only and hidden files in addition to normal files, pass (*faReadOnly* + *faHidden*) the *Attr* parameter.

FindFirst returns the results of the directory search in the search record you specify in the *F* parameter. You can then use the fields of the search record to extract the information you want.

The return value is zero if the function was successful. Otherwise the return value is a negative DOS error code; a value of -18 indicates that there are no more files matching the search criteria.

Example

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

```
var
  SearchRec: TSearchRec;
procedure TForm1.SearchClick(Sender: TObject);
begin
  FindFirst('c:\delphi\bin\*.*', faAnyFile, SearchRec);
  Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) +
    ' bytes in size';
end;
```

See also

FindNext function

FindIndexForFields method

Applies to

TIndexDefs object

Declaration

```
function FindIndexForFields(const Fields: string): TIndexDef;
```

Run-time and read only. Returns the *TIndexDef* object that is present in *Items* corresponding to a semicolon-separated list of fields.

FindItem method

Applies to

TMainMenu component

Declaration

```
function FindItem(Value: Word; Kind: TFindItemKind): TMenuItem;
```

The *FindItem* method returns the menu item owned by the menu that has either a menu handle, menu command, or menu shortcut matching the value of the *Value* parameter. The *Kind* parameter can be any of these values:

Value	Meaning
fkCommand	Menu command number used by Windows WM_COMMAND message
fkHandle	Menu handle
fkShortCut	Menu shortcut

Example

This example uses a label, a button, and a main menu component. The menu is a File menu that contains Open, Save, and Close commands. Delphi automatically names the menu items that are the commands, *Open1*, *Save1*, and *Close1*. The *Open1* menu item has a *ShortCut* value of *F3*. The code locates the menu item that has the specified shortcut and reports the name of the menu item in the caption of the label. Note that the shortcut is specified as a virtual key code. You can find a list of virtual key codes in the Help system. Search for the Virtual Key Codes topic.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ItemName: TMenuItem;
begin
  ItemName := MainMenu1.FindItem(VK_F3, fkShortCut);
  Label1.Caption := ItemName.Name;
end;
```

See also

Command property, *ShortCut* property, *TMenuItem* component

FindKey method

Applies to

TTable component

Declaration

```
function FindKey(const KeyValues: array of const): Boolean;
```

The *FindKey* method searches the *database table* to find a record whose index fields match those passed in *KeyValues*. *FindKey* takes a comma-delimited array of values as its argument, where each value corresponds to a index column in the underlying table. The values can be literals, variables, null, or **nil**. If the number of values supplied is less than the number of columns in the database table, then the remaining values are assumed to be null. *FindKey* will search for values specified in the array in the current index.

FindKey does the following:

- Puts the *TTable* in *SetKey* state.

- Finds the record in the table that matches the specified values. If a matching record is found, it moves the cursor there, and returns *True*.
- If a matching record is not found, it does not move the cursor, and returns *False*.

Example

```
{ Search for CustNo = '1234' }
if Table1.FindKey(['1234']) then
   ShowMessage('Customer Found');
```

See also

FindNearest method, *GotoKey* method

FindNearest method

Applies to

TTable component

Declaration

```
procedure FindNearest(const KeyValues: array of const);
```

The *FindNearest* method moves the cursor to the first record whose index fields' values are greater than or equal to those passed in *KeyValues*. The search begins at the first record, not at the current cursor position. This method can be used to match columns of string data type only. If you do not supply values for each field in the index key, any unassigned fields will use a null value.

FindNearest works by default on the primary index column. To search the table for values in other indexes, you must specify the field name in the table's *IndexFieldNames* property or the name of the index in the *IndexName* property.

The *KeyExclusive* property indicates whether a search will position the cursor on or after the specified record being searched for.

Note With Paradox or dBASE tables, *FindNearest* works only with indexed fields. With SQL databases, it can work with any columns specified in the *IndexFieldNames* property.

Example

```
{ Search for CustNo >= '1234' }
Table1.FindNearest(['1234']);
```

See also

FindKey method, *GotoKey* method, *GotoNearest* method, *TField* component

FindNext function

SysUtils

Declaration

```
function FindNext(var F: TSearchRec): Integer;
```

The *FindNext* function returns the next entry that matches the name and attributes specified in the previous call to the *FindFirst* function.

The search record must be the same one you passed to the *FindFirst* function.

The return value is zero if the function was successful. Otherwise the return value is a negative DOS error code; a value of -18 indicates that there are no more files matching the search criteria.

Example

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

```
var
  SearchRec: TSearchRec;
procedure TForm1.SearchClick(Sender: TObject);
begin
  FindFirst('c:\delphi\bin\*.*', faAnyFile, SearchRec);
  Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) +
    ' bytes in size';
end;
procedure TForm1.AgainClick(Sender: TObject);
begin
  FindNext(SearchRec);
  Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) +
    ' bytes in size';
end;
```

See also

FindFirst function

FindText property

Applies to

TFindDialog, *TReplaceDialog* components

Declaration

```
property FindText: string;
```

The *FindText* property contains the string your application can search for if it uses the Find dialog box.

You can specify a *FindText* value before the user displays the Find dialog box so that when it appears, the *FindText* value appears in the Find What edit box. The user can then either accept or change the *FindText* value before choosing the Find Next button in the dialog box.

Example

The following *OnFind* event handler searches a memo component for the text specified in the *FindText* property of a find dialog component. If found, the first occurrence of the text in *Memo1* is selected. The code uses the *Pos* function to compare strings, and stores the number of characters to skip when determining the selection position in the *SkipChars* variable. Because there is no handling of case, whole word, or search direction in this algorithm, it is assumed that the *Options* property of *FindDialog1* was set to [*frHideMatchCase*, *frHideWholeWord*, *frHideUpDown*].

```
procedure TForm1.FindDialog1Find(Sender: TObject);
var
  I, J, PosReturn, SkipChars: Integer;
begin
  For I := 0 to Memo1.Lines.Count do
  begin
    PosReturn := Pos(FindDialog1.FindText,Memo1.Lines[I]);
    if PosReturn <> 0 then {found!}
      begin
        Skipchars := 0;
        for J := 0 to I - 1 do
          Skipchars := Skipchars + Length(Memo1.Lines[J]);
        SkipChars := SkipChars + (I*2);
        SkipChars := SkipChars + PosReturn - 1;
        Memo1.SetFocus;
        Memo1.SelStart := SkipChars;
        Memo1.SelLength := Length(FindDialog1.FindText);
      end;
  end;
end;
```

See also

ReplaceText property

First method

Applies to

TList object; *TQuery*, *TStoredProc*, *TTable* components

For list objects

Declaration

```
function First: Pointer;
```

The *First* method returns a pointer that points to the first item referenced in the *List* property, which is indexed by *Items*[0].

Example

The following code assumes that the items in MyList are objects that have a text field named *Desc*. If the Desc of the first item in the list is 'Blue', the following code changes it to 'Green'.

```
   if MyList.First.Desc = 'Blue' then MyList.First.Desc := 'Green';
```

See also

IndexOf method, *Last* method

For tables, queries, and stored procedures

Declaration

```
procedure First;
```

The *First* method moves the cursor to the first record in the active range of records of the dataset. The active range of records is affected by the filter established with *ApplyRange*.

If the dataset is in Edit or Insert state, *First* will perform an implicit *Post* of any pending data.

See also

Last method, *MoveBy* method, *Next* method, *Prior* method, *SetRange* method, *SetRangeStart* method

FirstIndex property

Applies to

TTabSet component

Declaration

```
property FirstIndex: Integer;
```

Run-time only. The value of the *FirstIndex* property is the tab that appears in the leftmost visible position in the tab set control. Any tabs with a lower value in the *FirstIndex* property scroll to the left in the tab set control and don't appear until the user scrolls the tabs.

The default value of *FirstIndex* is 0 indicating that the tab with an index of 0 is in the leftmost position. For example, if you have three tabs labeled First, Second, and Third with *TabIndex* values of 0, 1, and 2, respectively, First appears first, by default, because it has an index value of 0. If you want to shift the tabs so the Second or Third tab appears leftmost in the tab set control, change the *FirstIndex* value to 1 or 2.

Example

This example uses a tab set control, a label, and a button on a form.

This code in an event handler creates 20 tabs labeled Tab 1 through Tab 20 when *Form1* is created:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to 19 do
   TabSet11.Tabs.Add('Tab ' + IntToStr(I));
end;
```

Users can scroll through the tabs. When they click the button, the caption of the first tab visible in the tab set control is displayed in the label control.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  Label1.Caption := IntToStr(TabSet11.FirstIndex);
end;
```

See also

TabIndex property, *Tabs* property

FixedColor property

Applies to

TDBGrid, *TDrawGrid*, *TStringGrid* components

Declaration

```
property FixedColor: TColor;
```

The value of the *FixedColor* property determines the color of nonscrolling or fixed columns and rows within the grid. Refer to the *Color* property for a list of the possible values for *FixedColor*.

The default color is *clBtnFace*, the color of the face of a button.

Example

This example uses a draw grid and a button on a form. When the user clicks the button, the color of the nonscrolling (fixed) rows and columns of the draw grid changes color.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if DrawGrid1.FixedColor = clBlue then
    DrawGrid1.FixedColor := clLime
  else
    DrawGrid1.FixedColor := clMaroon;
end;
```

See also
Color property, *FixedCols* property, *FixedRows* property

FixedCols property

Applies to
TDrawGrid, *TStringGrid* components

Declaration

```
property FixedCols: Integer;
```

The *FixedCols* property determines the number of nonscrolling columns within a grid. The default value is 1. Nonscrolling columns remain fixed at the far left of the grid, even when the user scrolls the other columns. Nonscrolling columns are useful for displaying row titles that need to remain visible in the grid at all times.

Each grid must have a least one column that isn't fixed. In other words, the value of the *FixedCols* property must always be at least one less than the value of the *ColCount* property, which contains the number of columns in the grid.

Example
This example uses a string grid and a button. When the user clicks the button, a message dialog box appears informing the user that a fixed column number of 2 is recommended. The dialog box also offers the user an opportunity to accept the recommended number if the number of fixed columns isn't already 2. If the user chooses Yes, the number of fixed columns changes to 2.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Check: Integer;
begin
  if StringGrid1.FixedCols <> 2 then
  begin
    Check := MessageDlg('2 fixed columns are recommended!  Change?',
        mtWarning, mbYesNoCancel, 0);
    if Check = IdYes then
       StringGrid1.FixedCols := 2;
  end;
end;
```

See also
FixedColor property, *FixedRows* property, *LeftCol* property

FixedRows property

Applies to
TDrawGrid, *TStringGrid* components

Declaration

```
property FixedRows: Integer;
```

The *FixedRows* property determines the number of nonscrolling rows within a grid. The default value is 1. Nonscrolling rows remain fixed at top of the grid, even when the user scrolls the other rows. Nonscrolling rows are useful for displaying column titles that need to remain visible in the grid at all times.

Each grid must have a least one row that isn't fixed. In other words, the value of the *FixedRows* property must always be at least one less than the value of the *RowCount* property, which contains the number of rows in the grid.

Example
This example uses a string grid and three radio buttons on a form. With the Object Inspector, specify the following event handler for all *OnClick* events of the three radio buttons. As the user selects different radio buttons, the number of fixed rows in the string grid changes.

```
procedure TForm1.RadioButton1Click(Sender: TObject);
begin
  if RadioButton1.Checked then
    StringGrid1.FixedRows := 1
  else if RadioButton2.Checked then
    StringGrid1.FixedRows := 2
  else if RadioButton3.Checked then
    StringGrid1.FixedRows := 3;
end;
```

See also
FixedColor property, *FixedCols* property, *TopRow* property

FloatToDecimal procedure **SysUtils**

Declaration

```
procedure FloatToDecimal(var Result: TFloatRec; Value: Extended; Precision, Decimals:
Integer);
```

FloatToDecimal converts a floating-point value to a decimal representation that is suited for further formatting.

The *Precision* parameter specifies the requested number of significant digits in the result—the allowed range is 1..18.

The *Decimals* parameter specifies the requested maximum number of digits to the left of the decimal point in the result.

Precision and *Decimals* together control how the result is rounded. To produce a result that always has a given number of significant digits regardless of the magnitude of the number, specify 9999 for the *Decimals* parameter.

The result of the conversion is stored in the specified *TFloatRec* record as follows:

Field	Value
Exponent	Contains the magnitude of the number, i.e. the number of significant digits to the right of the decimal point. The *Exponent* field is negative if the absolute value of the number is less than one. If the number is a NAN (not-a-number), *Exponent* is set to -32768. If the number is INF or -INF (positive or negative infinity), *Exponent* is set to 32767.
Negative	*True* if the number is negative, *False* if the number is zero or positive.
Digits	Contains up to 18 significant digits followed by a null terminator. The implied decimal point (if any) is not stored in *Digits*. Trailing zeros are removed, and if the resulting number is zero, NAN, or INF, *Digits* contains nothing but the null terminator.

FloatToStr function SysUtils

Declaration

```
function FloatToStr(Value: Extended): string;
```

FloatToStr converts the floating-point value given by *Value* to its string representation. The conversion uses general number format with 15 significant digits.

For further details, see the description of the *FloatToStrF* function.

FloatToStrF function SysUtils

Declaration

```
function FloatToStrF(Value: Extended; Format: TFloatFormat; Precision,
  Digits: Integer): string;
```

FloatToStrF converts the floating-point value given by *Value* to its string representation.

The *Format* parameter controls the format of the resulting string.

The *Precision* parameter specifies the precision of the given value. It should be 7 or less for values of type *Single*, 15 or less for values of type *Double*, and 18 or less for values of type *Extended*.

The meaning of the *Digits* parameter depends on the particular format selected.

The possible values of the *Format* parameter, and the meaning of each, are described below.

Value	Meaning
ffGeneral	General number format. The value is converted to the shortest possible decimal string using fixed or scientific format. Trailing zeros are removed from the resulting string, and a decimal point appears only if necessary. The resulting string uses fixed point format if the number of digits to the left of the decimal point in the value is less than or equal to the specified precision, and if the value is greater than or equal to 0.00001. Otherwise the resulting string uses scientific format, and the *Digits* parameter specifies the minimum number of digits in the exponent (between 0 and 4).
ffExponent	Scientific format. The value is converted to a string of the form "-d.ddd...E+dddd". The resulting string starts with a minus sign if the number is negative, and one digit always precedes the decimal point. The total number of digits in the resulting string (including the one before the decimal point) is given by the *Precision* parameter. The "E" exponent character in the resulting string is always followed by a plus or minus sign and up to four digits. The *Digits* parameter specifies the minimum number of digits in the exponent (between 0 and 4).
ffFixed	Fixed point format. The value is converted to a string of the form "-ddd.ddd...". The resulting string starts with a minus sign if the number is negative, and at least one digit always precedes the decimal point. The number of digits after the decimal point is given by the *Digits* parameter—it must be between 0 and 18. If the number of digits to the left of the decimal point is greater than the specified precision, the resulting value will use scientific format.
ffNumber	Number format. The value is converted to a string of the form "-d,ddd,ddd.ddd...". The *ffNumber* format corresponds to the *ffFixed* format, except that the resulting string contains thousand separators taken from WIN.INI.
ffCurrency	Currency format. The value is converted to a string that represents a currency amount. The conversion is controlled by the *CurrencyString*, *CurrencyFormat*, *NegCurrFormat*, *ThousandSeparator*, and *DecimalSeparator* global variables, all of which are initialized from the Currency Format in the International section of the Windows Control Panel and WIN.INI. The number of digits after the decimal point is given by the *Digits* parameter—it must be between 0 and 18.

For all formats, the actual characters used as decimal and thousand separators are obtained from the *DecimalSeparator* and *ThousandSeparator* global variables.

If the given value is a NAN (not-a-number), the resulting string is 'NAN'. If the given value is positive infinity, the resulting string is 'INF'. If the given value is negative infinity, the resulting string is '-INF'.

FloatToText function **SysUtils**

Declaration

```
function FloatToText(Buffer: PChar; Value: Extended; Format: TFloatFormat;
  Precision, Digits: Integer): Integer;
```

FloatToText converts the given floating-point value to its decimal representation using the specified format, precision, and digits. The resulting string of characters is stored in the given buffer, and the returned value is the number of characters stored. The resulting string is not null-terminated.

For further details, see the description of the *FloatToStrF* function.

FloatToTextFmt function **SysUtils**

Declaration

```
function FloatToTextFmt(Buffer: PChar; Value: Extended; Format: PChar): Integer;
```

FloatToTextFmt converts the given floating-point value to its decimal representation using the specified format. The resulting string of characters is stored in the given buffer, and the returned value is the number of characters stored. The resulting string is not null-terminated.

For further details, see the description of the *FormatFloat* function.

F

FloodFill method

Applies to

TCanvas object

Declaration

```
procedure FloodFill(X, Y: Integer; Color: TColor; FillStyle: TFillStyle);
```

The *FloodFill* method fills an area of the screen surface using the current brush specified by the *Brush* property. The *FloodFill* method begins at the point at coordinates (*X, Y)* and continues in all directions to the color boundary.

The way in which the area is filled is determined by the *FillStyle* parameter. If *FillStyle* is *fsBorder*, the area fills until a border of the color specified by the *Color* parameter is encountered. If *FillStyle* is *fsSurface*, the area fills as long as the color specified by the *Color* parameter is encountered. *fsSurface* fills are useful to fill an area with a multicolored border.

Example

The following code floodfills from the center point of *Form1*'s client area until the color black is encountered.

```
  Form1.Canvas.FloodFill(ClientWidth/2, ClientHeight/2, clBlack, fsBorder);
```

See also

Ellipse method, *FillRect* method, *Polygon* method, *Rectangle* method

Flush procedure **System**

Declaration

```
procedure Flush(var F: Text);
```

The *Flush* procedure clears the buffer of a text file open for output.

F is a text file variable.

When a text file is opened for output using *Rewrite* or *Append*, *Flush* empties the file's buffer. This guarantees that all characters written to the file at that time have actually been written to the external file. *Flush* has no effect on files opened for input.

{**$I+**} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using {**$I–**}, you must use *IOResult* to check for I/O errors.

Example

```
var
  f: TextFile;
begin
  if OpenDialog1.Execute then
  begin                                                  { open a text file }
    AssignFile(f, OpenDialog1.FileName);
    Append(f);
    Writeln(f, 'I am appending some stuff to the end of the file.');
    Flush(f);  { ensures that the text was actually written to file }
    CloseFile(f);
  end;
end;
```

FmtLoadStr function — SysUtils

Declaration

```
function FmtLoadStr(Ident: Word; const Args: array of const): string;
```

FmtLoadStr loads a string from a program's resource string table and uses that string, plus the *Args* array, as a parameter to *Format*. *Ident* is the string resource ID of the desired format string. *Result* is the output of *Format*.

See also

Format function

FmtStr procedure — SysUtils

Declaration

```
procedure FmtStr(var Result: string; const Format: string; const Args: array of const);
```

This function formats the series of arguments in the open array *Args*. Formatting is controlled by the Pascal format string *Format*; the results are returned in the parameter *Result*.

For information on the format strings, see Format Strings.

See also
FormatBu f function, *StrFmt function*, *StrLFmt function*

FocusControl method

Applies to
TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
function FocusControl;
```

Sets a form's focus to the first data-aware component associated with a *TField*. Use this method when doing record-oriented validation (for example, in the *BeforePost* event) since a field may be validated whether its associated data-aware components have focus.

Example

```
{ Set focus to first data-aware component associated with Field1 }
Field1.FocusControl;
```

FocusControl property

Applies to
TLabel component

Declaration

```
property FocusControl: TWinControl;
```

The *FocusControl* links the label control with another control on the form. If the *Caption* of a label includes an accelerator key, the control specified as the value of the *FocusControl* property becomes the focused control when the user uses the accelerator key.

The caption of a label often identifies the purpose of another control on the form, or directs the user to interact with it. For example, a label placed right above an edit box might have the caption 'File Name', indicating the user should type a file name in the edit box. In this case, making that edit box the value of the label's *FocusControl* property gives the edit box the focus when the user presses *Alt+F*.

Example
This code displays a line of text in a label on the form and associates the label with an edit box control. Note that the label caption includes an accelerator key. When the user presses *Alt+N*, the edit box control receives the focus:

```
Label1.Caption := '&Name';
Label1.FocusControl := Edit1;
```

For this example, you need to place the label and edit box control close together to make sure that users understand that they should enter text in the edit box.

See also

ShowAccelChar property, *TabStop* property

Focused method

Applies to

All windowed controls

Declaration

```
function Focused: Boolean;
```

The *Focused* method is used to determine whether a windowed control has the focus and is therefore is the *ActiveControl*.

Example

This example uses an edit box and a memo on a form. When the user switches the focus between the two controls, the control that currently has the focus becomes red:

```
type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Memo1: TMemo;
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    procedure ColorControl(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.ColorControl(Sender: TObject);
begin
  if Edit1.Focused then
    Edit1.Color := clRed
  else
    Edit1.Color := clWindow;
  if Memo1.Focused then
    Memo1.Color := clRed
  else
    Memo1.Color := clWindow;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Screen.OnActiveControlChange := ColorControl;
end;
```

See also
ActiveControl property, *OnActiveControlChange* event, *SetFocus* method

Font property

Applies to
TCanvas object; *TBitBtn*, *TButton*, *TCheckBox*, *TComboBox*, *TDBCheckBox*, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBLookupList*, *TDBMemo*, *TDBRadioGroup*, *TDBText*, *TDirectoryListBox*, *TDrawGrid*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TFilterComboBox*, *TFontDialog*, *TForm*, *TGroupBox*, *THeader*, *TLabel*, *TListBox*, *TMaskEdit*, *TMemo*, *TNotebook*, *TOutline*, *TPaintBox*, *TPanel*, *TRadioButton*, *TScrollBox*, *TSpeedButton*, *TStringGrid*, *TTabbedNotebook*, *TTabSet* components

Declaration

```
property Font: TFont;
```

The *Font* property is a font object that controls the attributes of text written on or in the component or object or sent to the printer. To modify a font, you change the value of the *Color*, *Name*, *Size*, or *Style* properties of the font object.

Example
This code changes color of text in a memo control to dark blue:

```
  Memo1.Font.Color := clNavy;
```

See also
ParentFont property

For Font dialog boxes

Declaration

```
property Font: TFont;
```

The *Font* property is the font the Font dialog box returns when the user uses the Font dialog box. Your application can then use this returned *Font* value for further processing.

You can also specify a default font before displaying the Font dialog box; the font name then appears selected in the Font combo box. Use the Object Inspector to specify a *Font* property, or assign a value to *Font* before using the *Execute method* to display the dialog box.

Example

This example uses a button, a Font dialog box, and a label on a form. When the user clicks the button, the Font dialog box appears. If the user uses the dialog box to change the font and chooses OK, the caption of the label changes to reflect the user's font selection.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  FontDialog1.Font.Name := 'System';
  FontDialog1.Font.Size := 10;
  if FontDialog1.Execute then
    Label1.Font := FontDialog1.Font;
end;
```

See also

Color property, *Name* property, *Size* property

Fonts property

Applies to

TPrinter object; *TScreen* component

Declaration

```
property Fonts: TStrings;
```

Run-time and read only. The *Fonts* property for the screen component returns a list of fonts supported by the screen.

The *Fonts* property for a printer object holds a list of fonts supported by the printer. The list contains TrueType fonts even if the printer doesn't support them natively because the Windows Graphics Device Interface (GDI) can draw TrueType fonts accurately when a print job uses them.

Example

This code displays the fonts supported by the screen in a *FontList* list box when the user clicks the *ListFonts* button:

```
procedure TForm1.ListFontsClick(Sender: TObject);
var
  FontIndex: Integer;
begin
  FontList.Clear;
  FontList.Sorted := True;
  FontList.Items := Screen.Fonts;
end;
```

See also
Canvas property, *Screen* variable, *Printer* variable

ForceDirectories procedure — FileCtrl

Declaration

```
procedure ForceDirectories(Dir: string);
```

Whenever you create directories using DOS and Windows, you must create one at a time. For example, if you want to create the C:\APPS\SALES\LOCAL directory, the APPS and SALES directories must exist before you can create the LOCAL directory.

The *ForceDirectories* can create all the directories specified along a directory path all at once if they don't exist. If the first directories in the path do exist, but the latter ones don't, *ForceDirectories* creates just the ones that don't exist.

Example
This example uses a label and a button on a form. When the user clicks the button, all the directories along the specified path that don't exist are created. The results are reported in the caption of the label:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Dir: string;
begin
  Dir := 'C:\APPS\SALES\LOCAL';
  ForceDirectories(Dir);
  if DirectoryExists(Dir) then
    Label1.Caption := Dir + ' was created'
end;
```

See also
DirectoryExists function, *SelectDirectory* function

Format function — SysUtils

Declaration

```
function Format(const Format: string; const Args: array of const): string;
```

This function formats the series of arguments in the open array *Args*. Formatting is controlled by the Object Pascal format string *Format*; the results are returned in the function result as a Pascal string.

For information on the format strings, see Format Strings.

Format strings

Format strings passed to the string formatting routines contain two types of objects—plain characters and format specifiers. Plain characters are copied verbatim to the resulting string. Format specifiers fetch arguments from the argument list and apply formatting to them.

Format specifiers have the following form:

```
"%" [index ":"] ["-"] [width] ["." prec] type
```

A format specifier begins with a % character. After the % come the following, in this order:

- An optional argument index specifier, [index ":"]
- An optional left justification indicator, ["-"]
- An optional width specifier, [width]
- An optional precision specifier, ["." prec]
- The conversion type character, type

The following table summarizes the possible values for type:

Value	What it specifies
d	Decimal. The argument must be an integer value. The value is converted to a string of decimal digits. If the format string contains a precision specifier, it indicates that the resulting string must contain at least the specified number of digits; if the value has less digits, the resulting string is left-padded with zeros.
e	Scientific. The argument must be a floating-point value. The value is converted to a string of the form "-d.ddd...E+ddd". The resulting string starts with a minus sign if the number is negative. One digit always precedes the decimal point. The total number of digits in the resulting string (including the one before the decimal point) is given by the precision specifer in the format string—a default precision of 15 is assumed if no precision specifer is present. The "E" exponent character in the resulting string is always followed by a plus or minus sign and at least three digits.
f	Fixed. The argument must be a floating-point value. The value is converted to a string of the form "-ddd.ddd...". The resulting string starts with a minus sign if the number is negative. The number of digits after the decimal point is given by the precision specifier in the format string—a default of 2 decimal digits is assumed if no precision specifier is present.
g	General. The argument must be a floating-point value. The value is converted to the shortest possible decimal string using fixed or scientific format. The number of significant digits in the resulting string is given by the precision specifier in the format string—a default precision of 15 is assumed if no precision specifier is present. Trailing zeros are removed from the resulting string, and a decimal point appears only if necessary. The resulting string uses fixed point format if the number of digits to the left of the decimal point in the value is less than or equal to the specified precision, and if the value is greater than or equal to 0.00001. Otherwise the resulting string uses scientific format.
n	Number. The argument must be a floating-point value. The value is converted to a string of the form "-d,ddd,ddd.ddd...". The "n" format corresponds to the "f" format, except that the resulting string contains thousand separators.

Value	What it specifies
m	Money. The argument must be a floating-point value. The value is converted to a string that represents a currency amount. The conversion is controlled by the *CurrencyString*, *CurrencyFormat*, *NegCurrFormat*, *ThousandSeparator*, *DecimalSeparator*, and *CurrencyDecimals* global variables, all of which are initialized from the Currency Format in the International section of the Windows Control Panel. If the format string contains a precision specifier, it overrides the value given by the *CurrencyDecimals* global variable.
p	Pointer. The argument must be a pointer value. The value is converted to a string of the form "XXXX:YYYY" where XXXX and YYYY are the segment and offset parts of the pointer expressed as four hexadecimal digits.
s	String. The argument must be a character, a string, or a *PChar* value. The string or character is inserted in place of the format specifier. The precision specifier, if present in the format string, specifies the maximum length of the resulting string. If the argument is a string that is longer than this maximum, the string is truncated.
x	Hexadecimal. The argument must be an integer value. The value is converted to a string of hexadecimal digits. If the format string contains a precision specifier, it indicates that the resulting string must contain at least the specified number of digits; if the value has fewer digits, the resulting string is left-padded with zeros.

Conversion characters may be specified in upper case as well as in lower case—both produce the same results.

For all floating-point formats, the actual characters used as decimal and thousand separators are obtained from the *DecimalSeparator* and *ThousandSeparator* global variables.

Index, width, and precision specifiers can be specified directly using decimal digit string (for example "%10d"), or indirectly using an asterisk character (for example "%*.*f"). When using an asterisk, the next argument in the argument list (which must be an integer value) becomes the value that is actually used. For example,

```
Format('%*.*f', [8, 2, 123.456])
```

is the same as

```
Format('%8.2f', [123.456]).
```

A width specifier sets the minimum field width for a conversion. If the resulting string is shorter than the minimum field width, it is padded with blanks to increase the field width. The default is to right-justify the result by adding blanks in front of the value, but if the format specifier contains a left-justification indicator (a "-" character preceding the width specifier), the result is left-justified by adding blanks after the value.

An index specifier sets the current argument list index to the specified value. The index of the first argument in the argument list is 0. Using index specifiers, it is possible to format the same argument multiple times. For example "Format('%d %d %0:d %d', [10, 20])" produces the string '10 20 10 20'.

The format strings are used by the following routines:

Format function

FormatBuf function

FmtStr procedure

StrFmt function

StrLFmt function

FormatBuf function SysUtils

Declaration

```
function FormatBuf(var Buffer; BufLen: Word; const Format; FmtLen: Word; const Args: array
of const): Word;
```

This function formats the series of arguments in the open array *Args*. *Formatting* is controlled by the format string *Format* (whose length is given by *FmtLen*); the results are returned in *Buffer* (whose length is given by *BufLen*). The function result contains the number of bytes in the *Result* buffer.

For information on the format strings, see Format Strings.

FormatChars property

Applies to
TDDEClientConv component

Declaration

```
property FormatChars: Boolean;
```

The *FormatChars* property determines if certain characters are filtered out of text data transferred from a DDE server application. Some DDE server applications transfer backspaces, linefeeds, carriage returns, and tabs with the text data. Sometimes, this can cause incorrect spacing, line breaks, or characters in the DDE client data. If this is the case, the characters should be filtered. The default value of *FormatChars* is *False*.

If *False*, all text characters of the linked data from the DDE server appear in the linked data in the DDE client. If *True*, ASCII characters 8 (backspace), 9 (tab), 10 (linefeed), and 13 (carriage return) are filtered out and won't appear in the DDE client data.

Example
The following code formats characters if the DDE service name is "SuperWrd".

```
if DDEClientConv.DDEService = 'SuperWrd' then
  DDEClientConv.FormatChars := True;
```

FormatCount property

Applies to
TClipboard object

Declaration

```
property FormatCount: Integer;
```

Run-time and read only. The *FormatCount* property value is the number of formats contained in the *Formats* array property of a Clipboard object.

Example

The following code adds each format on the Clipboard to *ListBox1* when *Button1* is clicked:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to Clipboard.FormatCount-1 do
    ListBox1.Items.Add(IntToStr(Clipboard.Formats[I]));
end;
```

See also

Assign method, *AsText* property, *Clipboard* variable, *GetComponent* method, *HasFormat* method, *SetComponent* method

FormatDateTime function — SysUtils

Declaration

```
function FormatDateTime(const Format: string; DateTime: TDateTime): string;
```

FormatDateTime formats the date-and-time value given by *DateTime* using the format given by *Format*. The following format specifiers are supported:

Specifier	Displays
c	Displays the date using the format given by the *ShortDateFormat* global variable, followed by the time using the format given by the *LongTimeFormat* global variable. The time is not displayed if the fractional part of the *DateTime* value is zero.
d	Displays the day as a number without a leading zero (1-31).
dd	Displays the day as a number with a leading zero (01-31).
ddd	Displays the day as an abbreviation (Sun-Sat) using the strings given by the *ShortDayNames* global variable.
dddd	Displays the day as a full name (Sunday-Saturday) using the strings given by the *LongDayNames* global variable.
ddddd	Displays the date using the format given by the *ShortDateFormat* global variable.
dddddd	Displays the date using the format given by the *LongDateFormat* global variable.
m	Displays the month as a number without a leading zero (1-12). If the *m* specifier immediately follows an *h* or *hh* specifier, the minute rather than the month is displayed.
mm	Displays the month as a number with a leading zero (01-12). If the *mm* specifier immediately follows an *h* or *hh* specifier, the minute rather than the month is displayed.
mmm	Displays the month as an abbreviation (Jan-Dec) using the strings given by the *ShortMonthNames* global variable.

Specifier	Displays
mmmm	Displays the month as a full name (January-December) using the strings given by the *LongMonthNames* global variable.
yy	Displays the year as a two-digit number (00-99).
yyyy	Displays the year as a four-digit number (0000-9999).
h	Displays the hour without a leading zero (0-23).
hh	Displays the hour with a leading zero (00-23).
n	Displays the minute without a leading zero (0-59).
nn	Displays the minute with a leading zero (00-59).
s	Displays the second without a leading zero (0-59).
ss	Displays the second with a leading zero (00-59).
t	Displays the time using the format given by the *ShortTimeFormat* global variable.
tt	Displays the time using the format given by the *LongTimeFormat* global variable.
am/pm	Uses the 12-hour clock for the preceding *h* or *hh* specifier, and displays 'am' for any hour before noon, and 'pm' for any hour after noon. The *am/pm* specifier can use lower, upper, or mixed case, and the result is displayed accordingly.
a/p	Uses the 12-hour clock for the preceding *h* or *hh* specifier, and displays 'a' for any hour before noon, and 'p' for any hour after noon. The *a/p* specifier can use lower, upper, or mixed case, and the result is displayed accordingly.
ampm	Uses the 12-hour clock for the preceding *h* or *hh* specifier, and displays the contents of the *TimeAMString* global variable for any hour before noon, and the contents of the *TimePMString* global variable for any hour after noon.
/	Displays the date separator character given by the *DateSeparator* global variable.
:	Displays the time separator character given by the *TimeSeparator* global variable.
'xx'/"xx"	Characters enclosed in single or double quotes are displayed as-is, and do not affect formatting.

Format specifiers may be written in upper case as well as in lower case letters—both produce the same result.

If the string given by the *Format* parameter is empty, the date and time value is formatted as if a '*c*' format specifier had been given.

Example

The following example assigns 'The meeting is on Wednesday, February 15, 1995 at 10:30 AM' to the string variable *S*.

```
S := FormatDateTime('"The meeting is on" dddd, mmmm d, yyyy, ' +
  '"at" hh:mm AM/PM', StrToDateTime('2/15/95 10:30am'));
```

FormatFloat function — SysUtils

Declaration

```
function FormatFloat(const Format: string; Value: Extended): string;
```

FormatFloat formats the floating-point value given by *Value* using the format string given by *Format*. The following format specifiers are supported in the format string:

Specifier	Represents
0	Digit placeholder. If the value being formatted has a digit in the position where the '0' appears in the format string, then that digit is copied to the output string. Otherwise, a '0' is stored in that position in the output string.
#	Digit placeholder. If the value being formatted has a digit in the position where the '#' appears in the format string, then that digit is copied to the output string. Otherwise, nothing is stored in that position in the output string.
.	Decimal point. The first '.' character in the format string determines the location of the decimal separator in the formatted value; any additional '.' characters are ignored. The actual character used as a the decimal separator in the output string is determined by the *DecimalSeparator* global variable. The default value of *DecimalSeparator* is specified in the Number Format of the International section in the Windows Control Panel.
,	Thousand separator. If the format string contains one or more ',' characters, the output will have thousand separators inserted between each group of three digits to the left of the decimal point. The placement and number of ',' characters in the format string does not affect the output, except to indicate that thousand separators are wanted. The actual character used as a the thousand separator in the output is determined by the *ThousandSeparator* global variable. The default value of *ThousandSeparator* is specified in the Number Format of the International section in the Windows Control Panel.
E+	Scientific notation. If any of the strings 'E+', 'E-', 'e+', or 'e-' are contained in the format string, the number is formatted using scientific notation. A group of up to four '0' characters can immediately follow the 'E+', 'E-', 'e+', or 'e-' to determine the minimum number of digits in the exponent. The 'E+' and 'e+' formats cause a plus sign to be output for positive exponents and a minus sign to be output for negative exponents. The 'E-' and 'e-' formats output a sign character only for negative exponents.
'xx'/"xx"	Characters enclosed in single or double quotes are output as-is, and do not affect formatting.
;	Separates sections for positive, negative, and zero numbers in the format string.

The locations of the leftmost '0' before the decimal point in the format string and the rightmost '0' after the decimal point in the format string determine the range of digits that are always present in the output string.

The number being formatted is always rounded to as many decimal places as there are digit placeholders ('0' or '#') to the right of the decimal point. If the format string contains no decimal point, the value being formatted is rounded to the nearest whole number.

If the number being formatted has more digits to the left of the decimal separator than there are digit placeholders to the left of the '.' character in the format string, the extra digits are output before the first digit placeholder.

To allow different formats for positive, negative, and zero values, the format string can contain between one and three sections separated by semicolons.

- One section: The format string applies to all values.
- Two sections: The first section applies to positive values and zeros, and the second section applies to negative values.
- Three sections: The first section applies to positive values, the second applies to negative values, and the third applies to zeros.

If the section for negative values or the section for zero values is empty, that is if there is nothing between the semicolons that delimit the section, the section for positive values is used instead.

If the section for positive values is empty, or if the entire format string is empty, the value is formatted using general floating-point formatting with 15 significant digits, corresponding to a call to *FloatToStrF* with the *ffGeneral* format. General floating-point formatting is also used if the value has more than 18 digits to the left of the decimal point and the format string does not specify scientific notation.

Example

The following table shows some sample formats and the results produced when the formats are applied to different values:

Format string–	1234	–1234	0.5	0
	1234	–1234	0.5	0
0	1234	–1234	1	0
0.00	1234.00	–1234.00	0.50	0.00
#.##	1234	–1234	.5	
#,##0.00	1,234.00	–1,234.00	0.50	0.00
#,##0.00;(#,##0.00)	1,234.00	(1,234.00)	0.50	0.00
#,##0.00;;Zero	1,234.00	–1,234.00	0.50	Zero
0.000E+00	1.234E+03	–1.234E+03	5.000E–01	0.000E+00
#.###E–0	1.234E3	–1.234E3	5E–1	0E0

Formats property

Applies to

TClipboard object

Declaration

```
property Formats[Index: Integer]: Word;
```

Run-time and read only. The *Formats* property array contains a list of all the formats the Clipboard contains. Usually when an application copies or cuts something to the Clipboard, it places it there in multiple formats.

Your application can place items of a particular format on the Clipboard and retrieve items with a particular format from the Clipboard if the format is in the *Formats* array. You can find out if a particular format is available on the Clipboard with the *HasFormat* method.

The *Index* parameter of the *Formats* property lets you access a format by its position in the array.

Example

The following code adds each format on the Clipboard to *ListBox1* when *Button1* is clicked:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to Clipboard.FormatCount-1 do
    ListBox1.Items.Add(IntToStr(Clipboard.Formats[I]));
end;
```

See also

Assign method, *AsText* property, *Clipboard* variable, *FormatCount* property, *HasFormat* method

FormCount property

Applies to

TScreen component

Declaration

```
property FormCount: Integer;
```

Run-time and read only. The *FormCount* property value contains the number of forms displayed on the screen.

Example

The following code adds the name of all forms on the screen to *ListBox1* when *Button1* is clicked.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  For I := 0 to Screen.FormCount-1 do
    ListBox1.Items.Add(Screen.Forms[I].Name);
end;
```

See also

Forms property, *Screen* variable

Forms property

Applies to

TScreen component

Declaration

```
property Forms[Index: Integer]: TForm;
```

Description

Run-time and read only. The *Forms* property lets you access a form on the screen by specifying its position in the list of forms kept by the *TScreen* component using its *Index* value. The first form has an index value of 0, the second has an index value of 1, and so on.

Example

The following code adds the name of all forms on the screen to *ListBox1* when *Button1* is clicked.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  For I := 0 to Screen.FormCount-1 do
    ListBox1.Items.Add(Screen.Forms[I].Name);
end;
```

See also

FormCount property, *Screen* variable

FormStyle property

Applies to

TForm component

Declaration

```
property FormStyle: TFormStyle;
```

The *FormStyle* property determines the style of the form. These are the possible values and their meanings:

Value	Meaning
fsNormal	The form is neither an MDI parent window nor an MDI child window.
fsMDIChild	The form is an MDI child window.
fsMDIForm	The form is an MDI parent window.
fsStayOnTop	This form remains on top of other forms in the project, except any others that also have *FormStyle* set to *fsStayOnTop*.

The default value is *fsNormal*.

All MDI (Multiple Document Interface) applications must have the *FormStyle* property of the main form set to *fsMDIForm*. All forms specified as MDI child forms display as forms contained within the MDI parent form. You must use the Object Inspector to set the child form's *Visible* property to *True* or your child form won't appear. You can have as many child forms as you like.

Example

This example ensures the main form of the application is an MDI parent form:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if FormStyle <> fsMDIForm then
    FormStyle := fsMDIForm;
  if FormStyle = fsMDIForm then
    Edit1.Text := 'MDI form'
  else
    Edit1.Text := 'Not an MDI form';   {This line never runs}
end;
```

See also

CreateForm method, *MainForm* property, *Visible* property

Frac function **System**

Declaration

```
function Frac(X: Real): Real;
```

The *Frac* function returns the fractional part of the argument *X*.

X is a real-type expression. The result is the fractional part of *X*; that is, `Frac(X) = X - Int(X).`

Example

```
 var
  R: Real;
begin
  R := Frac(123.456);    { 0.456 }
  R := Frac(-123.456);   { -0.456 }
end;
```

See also

Int function

FrameRect method

Applies to

TCanvas object

Declaration

```
procedure FrameRect(const Rect: TRect);
```

The *FrameRect* method draws a rectangle using the *Brush* of the canvas to draw the border. *FrameRect* does not fill the interior of the rectangle with the *Brush* pattern.

Example

The following code displays the text "Hello, world!" in a rectangle defined by the coordinates (10, 10) and (100, 100). After displaying the text with the *TextRect* method, the code draws a black, vertical line frame around the rectangle.

```
var
  TheRect: TRect;
begin
  Form1.Canvas.Brush.Color := clBlack;
  Form1.Canvas.Brush.Style := bsVertical;
  TheRect.Top := 10;
  TheRect.Left := 10;
  TheRect.Bottom := 100;
  TheRect.Right := 100;
  Form1.Canvas.TextRect(TheRect,10,10,'Hello, world!');
  Form1.Canvas.FrameRect(TheRect);
end;
```

See also

Brush property, *Rect* function, *TextRect* method

Frames property

Applies to

TMediaPlayer component

Declaration

```
property Frames: Longint;
```

Run-time-only. The *Frames* property specifies the number of frames the *Step* method steps forward or the *Back* method steps backward.

Frames defaults to ten percent of the length of the currently loaded medium, which is specified by the *Length* property.

Note The definition of frame varies by multimedia device. For display media, a frame is one still image.

Free method

Applies to

All objects and components

Declaration

```
procedure Free;
```

The *Free* method destroys the object and frees its associated memory. If you created the object yourself using the *Create* method, you should use *Free* to destroy and release memory. *Free* is successful even if the object is **nil**, so if the object was never initialized, for example, calling *Free* won't result in an error.

Delphi automatically destroys Visual Component Library objects and frees memory allocated to them.

You should never explicitly free a component within one of its own event handlers, nor should you free a component from an event handler of a component the component owns or contains. For example, you should avoid freeing a button in its *OnClick* event handler. Nor should you free the form that owns the button from the button's *OnClick* event.

If you want to free the form, call the *Release* method, which destroys the form and releases the memory allocated for it after all its event handlers and those of the components it contains are through executing.

Example

The following code frees an object called *MyObject*:

```
MyObject.Free;
```

See also

Destroy method, *Release* method

Free procedure — System

Declaration

```
procedure Free;
```

The *Free* procedure tests whether or not the instance of the caller is **nil**.

If it isn't **nil**, *Free* calls *Destroy*.

If it is **nil**, the *Free* call is ignored.

FreeBookmark method

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure FreeBookmark(Bookmark: TBookmark);
```

Use the *FreeBookmark* method in combination with the *GetBookmark* and *GotoBookmark* methods. *FreeBookmark* releases the system resources reserved during a call to *GetBookmark*.

Example

```
var MyBookmark: TBookmark;
...
with Table1 do
  begin
{ Save the current record position in MyBookmark }
  MyBookmark := GetBookmark;
  ... { Other code here }
{ Return to the record associated with MyBookmark }
  GotoBookmark(MyBookmark);
{ Release the resources for MyBookmark }
  FreeBookmark(MyBookmark);
  end;
```

FreeMem procedure — System

Declaration

```
procedure FreeMem(var P: Pointer; Size: Word);
```

The *FreeMem* procedure disposes of a dynamic variable of a given size.

P is a variable of any pointer type previously assigned by the *GetMem* procedure or assigned a meaningful value using an assignment statement.

Size specifies the size in bytes of the dynamic variable to dispose of; it must be exactly the number of bytes previously allocated to that variable by *GetMem*.

FreeMem destroys the variable referenced by *P* and returns its memory to the heap. If *P* does not point to memory in the heap, a run-time error occurs.

After calling *FreeMem*, the value of *P* is undefined, and an error occurs if you subsequently reference *P*^. You can use the exceptions to handle this error. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

Example

```
type
   TFriendRec = record
     Name: string[30];
     Age : Byte;
   end;

 var
   p: pointer;
 begin
   if MaxAvail < SizeOf(TFriendRec) then
     MessageDlg('Not enough memory', mtWarning, [mbOk], 0);
   else
   begin
     { Allocate memory on heap }
     GetMem(p, SizeOf(TFriendRec));
     { ...}
     { ...Use the memory... }
     { ...}
     { then free it when done }
     FreeMem(p, SizeOf(TFriendRec));
   end;
 end;
```

See also
Dispose procedure, *GetMem* procedure, *New* procedure

FromPage property

Applies to
TPrintDialog component

Declaration

```
property FromPage: Integer;
```

The value of the *FromPage* property determines on which page the print job begins. The default value is 0.

Example
This example uses a Print dialog box on a form. These lines set up the Print dialog box so that when it appears, the default values of 1 and 1 are the default starting and ending values for the Pages From and To edit boxes.

```
PrintDialog1.Options := [poPageNums];
PrintDialog1.FromPage := 1;
PrintDialog1.ToPage := 1;
```

See also

ToPage property

FullCollapse method

Applies to

TOutline component

Declaration

```
procedure FullCollapse;
```

FullCollapse collapses all the items within an outline. When an item is collapsed, its *Expanded* property is set to *False*, its subitems are hidden, and the closed or plus pictures might be displayed, depending on the outline style specified in the *OutlineStyle* property.

Example

The following code collapses the outline if the selected item is visible.

```
if Outline[Outline1.SelectedItem].IsVisible then
  Outline1.FullCollapse;
```

See also

Collapse method, *Expand* method, *FullExpand* method, *OnCollapse* event, *PictureClosed* property, *PicturePlus* property

FullExpand method

Applies to

TOutlineNode object; *TOutline* component

Declaration

```
procedure FullExpand;
```

FullExpand expands the items within an outline. If the *FullExpand* method belongs to a *TOutline* component, all items in the outline are expanded. If the *FullExpand* method belongs to a *TOutlineNode* object, only the items on the same branch as the outline node are expanded. This means that all subitems are expanded, and all parents up to the top item on level 1 (specified by the *TopItem* property) are expanded. No items on other branches (with different level 1 parents) are expanded.

When an item is expanded, its *Expanded* property is set to *True*, its subitems are displayed, and the open or minus pictures might be displayed, depending on the outline style specified in the *OutlineStyle* property.

Example

The following code expands the outline if the selected item is not visible:

```
if not Outline1.Items[Outline1.SelectedItem].IsVisible then
  Outline1.FullExpand;
```

See also

Collapse method, *Expand* method, *FullCollapse* method, *OnExpand* event, *PictureMinus* property, *PictureOpen* property

FullPath property

Applies to

TOutlineNode object

Declaration

```
property FullPath: string;
```

Run-time and read only. The *FullPath* property specifies the path of outline items from the top item on level 1 to the item contained by the *TOutlineNode*. The path consists of the values of the *Text* properties of the outline items separated by the string specified in the *ItemSeparator* property of the *TOutline* component.

Example

The following code displays the full path of the selected outline item in *Label1*:

```
Label1.Caption := Outline1.Items[Outline1.SelectedItem].FullPath;
```

See also

Items property, *SelectedItem* property

GetAliasNames method

Applies to

TSession component

Declaration

```
procedure GetAliasNames(List: TStrings);
```

The *GetAliasNames* method clears the parameter *List* and adds to it the names of all defined BDE aliases. Application-specific aliases are not included.

Example

```
Session.GetAliasNames(MyStringList);
```

See also

GetDataBaseNames method, *Session* variable

GetAliasParams method

Applies to

TSession component

Declaration

```
procedure GetAliasParams(const AliasName: string; List: TStrings);
```

The *GetAliasParams* method clears *List* and adds to it the parameters associated with the BDE alias passed in *AliasName*.

Example

```
Session.GetAliasParams(MyStringList);
```

See also

Session variable

GetAsHandle method

Applies to

TClipboard object

Declaration

```
function GetAsHandle (Format: Word): THandle;
```

The *GetAsHandle* method returns the data from the Clipboard in a Windows handle for the format specified in the *Format* parameter. See the Windows API Help file for information about the available formats.

Your application doesn’t own the handle, so it should copy the data before using it.

Example

The following code locks the memory for text on the Clipboard, then converts the text to a Pascal-style string.

```
var
  TheClipboard: TClipboard;
  MyHandle: THandle;
  TextPtr: PChar;
  MyString: string;

begin
  MyHandle := TheClipboard.GetAsHandle(CF_TEXT);
```

```
    TextPtr := GlobalLock(MyHandle);
    MyString := StrPas(TextPtr);
    GlobalUnlock(MyHandle);
  end;
```

See also

FormatCount property, *Formats* property, *HasFormat* method, *SetAsHandle* method

GetBookmark method

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
function GetBookmark: TBookmark;
```

The *GetBookmark* method saves the current record information of the *dataset* to allow you to return to that record with a later call to the *GotoBookmark* method. The bookmark should be eventually be passed to the *FreeBookmark* method to release the resources reserved during the call to *GetBookmark*. If the dataset is empty or not in Browse state, *GetBookmark* will return **nil**.

Note All bookmarks are invalidated when a dataset is closed and when a table's index is changed.

Example

```
  var MyBookmark: TBookmark;
  ...
  with Table1 do
    begin
  { Save the current record position in MyBookmark }
    MyBookmark := GetBookmark;
    ... { Other code here }
  { Return to the record associated with MyBookmark }
    GotoBookmark(MyBookmark);
  { Release the resources for MyBookmark }
    FreeBookmark(MyBookmark);
    end;
```

GetComponent method

Applies to

TClipboard object

Declaration

```
function GetComponent(Owner, Parent: TComponent): TComponent;
```

The *GetComponent* method retrieves a component from the Clipboard and places it according to the value of the *Owner* and *Parent* parameters. With *Owner*, specify the component that becomes the owner of the retrieved component—usually this is a form. With *Parent*, specify the component that becomes the parent of the component. Both *Owner* and *Parent* can be **nil**.

Example

This example uses a button and a group box on a form. When the user clicks the button, the button is copied to the Clipboard and then retrieved from the Clipboard and placed in the new parent of the button, the group box. The name of the original button is changed to an empty string to avoid having two components with the same name at the same time.

```
implementation

uses Clipbrd;

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  Clipboard.SetComponent(Button1);                { copies button to the Clipboard }
  Button1.Name := '';          { prevents having two components with the same name }
  Clipboard.GetComponent(Self, GroupBox1);   { retrieves button from Clipboard and }
end;                                                    { places it in the group box }

initialization
  RegisterClasses([TButton]);                           { registers the TButton class }
end.
```

See also

AsText property, *Owner* property, *Parent* property, *SetComponent* method

GetData method

Applies to

TParam object; *TBCDField*, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

For fields

Declaration

```
function GetData(Buffer: Pointer): Boolean;
```

GetData is the method used to obtain "raw" data from the field. Unlike the *AsString*, *DisplayText*, and *Text* properties, *GetData* performs no translation or interpretation of the data. *Buffer* must have sufficient space allocated for the data. Use the *DataSize* property to determine the space required. If the data is NULL, *GetData* returns *False* and no data is transferred to *Buffer*. Otherwise, it returns *True*.

Example

```
{ Retrieve the "raw" data from Field1 }
with Field1 do
  begin
{ Allocate space }
  GetMem(Buffer, DataSize);
  if not Field1.GetData(Buffer) then
    MessageDlg(FieldName + ' is NULL', mtInformation, [mbOK], 0)
  else { Do something with the data };
{ Free the space }
  FreeMem(Buffer, DataSize);
  end;
```

For Tparam objects

Declaration

```
procedure GetData(Buffer: Pointer);
```

The *GetData* method copies the current value of the parameter in native format to *Buffer*. *Buffer* must have enough space to hold the information; use the *GetDataSize* method to determine the requirement.

Example

```
var Buffer: Pointer;
{ Allocate enough space to hold the CustNo data }
GetMem(Buffer, Query1.ParamByName('CustNo').GetDataSize);
{ Retrieve the data }
Query1.ParamByName('CustNo').GetData(Buffer);
```

See also
SetData method

GetDatabaseNames method

Applies to
TSession component

Declaration

```
procedure GetDatabaseNames(List: TStrings);
```

The *GetDatabaseNames* method clears *List* and adds to it the names of all BDE aliases and application-specific aliases.

Example

```
Session.GetDatabaseNames(MyStringList);
```

See also

GetAliasNames method, *Session* variable

GetDataItem method

Applies to

TOutline component

Declaration

```
function GetDataItem(Value: Pointer): Longint;
```

The *GetDataItem* method returns the *Index* value of the first outline item that contains the data specified in the *Value* parameter in its *Data* property. Use *GetDataItem* when you have a pointer to data and you want to know which outline item contains the data.

Example

The following code displays the *Text* of the outline item that points to the variable *P3* in its *Data* property. The text is displayed in a label.

```
Label1.Caption := Outline1.Items[GetDataItem(p3)].Text;
```

See also

GetItem method, *GetTextItem* method

GetDataSize method

Applies to

TParam object

Declaration

```
function GetDataSize: Word;
```

The *GetDataSize* method returns the number of bytes required to hold the parameter's value. Use *GetDataSize* in conjunction with the *GetData* method to allocate memory for the parameter's data.

Example

```
var Buffer: Pointer;
```

```
{ Allocate enough space to hold the CustNo data }
GetMem(Buffer, Query1.ParamByName('CustNo').GetDataSize);
{ Retrieve the data }
Query1.ParamByName('CustNo').GetData(Buffer);
```

GetDir procedure **System**

Declaration

```
procedure GetDir(D: Byte; var S: string);
```

The *GetDir* procedure returns the current directory of a specified drive.

D can be set to any of the following values:

Value	Drive
0	Default
1	A
2	B
3	C

Performs no error checking. If the drive specified by *D* is invalid, *S* returns X:\ as if it were the root directory of the invalid drive.

Example

```
var
   s : string;
 begin
   GetDir(0,s); { 0 = Current drive }
   MessageDlg('Current drive and directory: ' + s, mtInformation, [mbOk] , 0);
 end;
```

See also

ChDir function, *MkDir* procedure, *RmDir* procedure

GetDriverNames method

Applies to

TSession component

Declaration

```
procedure GetDriverNames(List: TStrings);
```

The *GetDriverNames* method clears *List* and adds to it the names of all BDE drivers currently installed. This will not include 'PARADOX' or 'DBASE', since these databases are handled by the driver named 'STANDARD'.

Example

```
Session.GetDriverNames(MyStringList);
```

See also

Session variable

GetDriverParams method

Applies to

TSession component

Declaration

```
procedure GetDriverParams(const DriverName: string; List: TStrings);
```

The *GetDriverParams* method clears *List* and adds to it the default parameters for the driver named in *DriverName* parameter. The driver named 'STANDARD' (used for Paradox and dBASE tables) has only one parameter, 'PATH='. SQL drivers will have varying parameters.

Example

```
Session.GetDriverParams(MyStringList);
```

See also

Session variable

GetFieldNames method

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure GetFieldNames(List: TStrings);
```

The *GetFieldNames* method clears the *TStrings* argument, *List,* and then adds the name of each field in the *dataset* to it.

Example

```
var FieldNames: TStringList;
...
{ Initialize FieldNames to hold the names }
  FieldNames := TStringList.Create;
{ Get the names }
  Table1.GetFieldNames(FieldNames);
{ Do something with them }
...
{ Free the TStringList }
  FieldNames.Free;
```

GetFirstChild method

Applies to

TOutlineNode object

Declaration

```
function GetFirstChild: Longint;
```

The *GetFirstChild* method returns the *Index* value of the first subitem in an outline item. If the item has no subitems, *GetFirstChild* returns -1.

Example

The following code expands the selected outline item if it has children and then selects the first child.

```
  with Outline1 do
  if Items[SelectedItem].HasItems then
  begin
    Items[SelectedItem].Expanded := True;
    SelectedItem := Items[Items[SelectedItem].GetFirstChild];
  end;
```

See also

GetLastChild method, *GetNextChild* method, *GetPrevChild* method

GetFormatSettings procedure SysUtils

Declaration

```
procedure GetFormatSettings;
```

GetFormatSettings reloads all the date and number format preferences stored in the WIN.INI file's International section. When a program, such as Control Panel, modifies the WIN.INI file, it should notify other running applications by broadcasting a WM_WININIFILEChanged message. Your application should call *GetFormatSettings* when you receive this message.

GetFormImage method

Applies to

TForm component

Declaration

```
function GetFormImage: TBitmap;
```

The *GetFormImage* returns a bitmap of the form as it appears when printed.

Example

This example uses an image, a button, and a shape component on a form. When the user clicks the button, an image of the form is stored in the *FormImage* variable and copied to the Clipboard. Then image of the form in then copied back to the image component, producing an interesting result, especially if the button is clicked multiple times.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  FormImage: TBitmap;
begin
  FormImage := GetFormImage;
  Clipboard.Assign(FormImage);
  Image1.Picture.Assign(Clipboard);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Shape1.Shape := stEllipse;
  Shape1.Brush.Color := clLime;
  Image1.Stretch := True;
end;
```

See also

PrintScale property

GetHelpContext method

Applies to

TMainMenu component

Declaration

```
function GetHelpContext(Value: Word; ByCommand: Boolean): THelpContext;
```

The *GetHelpContext* method returns a help context number.

See also

HelpContext property, *HelpContext* method, *HelpJump* method, *OnHelp* event

GetIndexForPage method

Applies to

TTabbedNotebook component

Declaration

```
function GetIndexForPage(const PageName: string): Integer;
```

The *GetIndexForPage* method returns the *PageIndex* value of the specified page. The *PageIndex* property value is determined by the page's position in the *Pages* property array. Specify the name of the page as the value of the *PageName* parameter. The name you specify must be one of the strings in the *Pages* property.

Example

This example uses a tabbed notebook and a label on a form. When the form is created, pages are added to the tabbed notebook. The *PageIndex* value of the Preferences page appears in the caption of the label.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  with TabbedNotebook1 do
  begin
    Pages.Clear;
    Pages.Add('Styles');
    Pages.Add('Fonts');
    Pages.Add('Preferences');
  end;
  Label1.Caption := 'The Preferences page has an index of ' +
    IntToStr(TabbedNotebook1.GetIndexForPage('Preferences'));
end;
```

See also

SetTabFocus method

GetIndexNames method

Applies to

TTable component

Declaration

```
procedure GetIndexNames(List: TStrings);
```

The *GetIndexNames* method adds the names of all available indexes for the *TTable* to the *List* parameter.

Example

```
var
  MyList: TStringList;
...
MyList := TStringList.Create;
Table1.GetIndexNames(MyList);
{ Do something with the names }
```

```
MyList.Free;
```

See also
IndexName property

GetItem method

Applies to
TOutline component

Declaration

```
function GetItem(X, Y: Integer): Longint;
```

The *GetItem* method returns the *Index* value of the outline item that resides at the pixel coordinates (*X*, *Y*). Use *GetItem* when you want to know which outline item is in a specific screen location.

Example
The following code makes the item at screen coordinates (34,100) the selected item.

```
Outline1.SelectedItem := Outline1.GetItem(34,100);
```

See also
GetDataItem method, *GetTextItem* method

GetItemPath method

Applies to
TDirectoryListBox component

Declaration

```
function GetItemPath(Index : Integer): string;
```

The *GetItemPath* method returns as a string the path of a directory in a directory list box. Specify the directory with the *Index* value using the first directory in the list that has an index value of 0.

Example
This example uses a directory list box, a button, and a label on a form. When the user selects a directory in the directory list box and clicks the button, the selected directory opens, and the path of the second directory displayed in the list box appears as the caption of the label.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  DirectoryListBox1.OpenCurrent;
```

```
    Label1.Caption := DirectoryListBox1.GetItemPath(1);
  end;
```

See also
Directory property, *Drive* property, *OpenCurrent* method

GetLastChild method

Applies to
TOutlineNode object

Declaration

```
function GetLastChild: Longint;
```

The *GetLastChild* method returns the *Index* value of the last subitem in an outline item. If the item has no subitems, *GetLastChild* returns -1.

Example
The following code expands the selected outline item if it has children and then selects the last child.

```
with Outline1 do
if Items[SelectedItem].HasItems then
begin
  Items[SelectedItem].Expanded := True;
  SelectedItem := Items[Items[SelectedItem].GetLastChild];
end;
```

See also
GetFirstChild method, *GetNextChild* method, *GetPrevChild* method

GetLongHint function **Controls**

Declaration

```
function GetLongHint(const Hint: string): string;
```

The *GetLongHint* function returns the second part of the two-part string specified as the value of the *Hint* property. The second part of the string is the text following the | character. If the *Hint* string value is not separated into two parts, *GetLongHint* returns the entire Hint string.

Example
This code assigns a two-part string as to the *Hint* property of an edit box and then displays the “long” or second part of the string as the text of the edit box:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
```

```
begin
  Edit1.Hint := 'Name|Enter full name';
  Edit1.Text := GetLongHint(Edit1.Hint);
end;
```

See also

GetShortHint function, *OnHint* event, *ShowHint* property

GetMem procedure System

Declaration

```
procedure GetMem(var P: Pointer; Size: Word);
```

The *GetMem* procedure creates a dynamic variable of the specified size and puts the address of the block in a pointer variable.

P is a variable of any pointer type. *Size* is an expression specifying the size in bytes of the dynamic variable to allocate. You should reference the newly created variable as *P^*.

If there isn't enough free space in the heap to allocate the new variable, a run-time error occurs. When {**$I+**}, you can use the exceptions to handle the error. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

The largest single block that can be safely allocated on the heap at one time is 65,528 bytes.

Example

```
type
  TFriendRec = record
    Name: string[30];
    Age : Byte;
  end;

 var
  p: pointer;
 begin
  if MaxAvail < SizeOf(TFriendRec) then
    MessageDlg('Not enough memory', mtWarning, [mbOk], 0);
  else
  begin
    { Allocate memory on heap }
    GetMem(p, SizeOf(TFriendRec));
    { ...}
    { ...Use the memory... }
    { ...}
    { then free it when done }
    FreeMem(p, SizeOf(TFriendRec));
  end;
 end;
```

See also
Dispose procedure, *FreeMem* procedure, *New* procedure

GetNextChild method

Applies to
TOutlineNode object

Declaration

```
function GetNextChild(Value: Longint): Longint;
```

The *GetNextChild* method returns the *Index* value of the next outline item that shares the same parent item as the item that has an *Index* value equal to the *Value* parameter. This is useful when the item indexed by *Value* has subitems, thus the index of its next sibling is not simply one more than *Value*. If the item indexed by *Value* has no next sibling, *GetNextChild* returns -1.

Example
The following code selects the next sibling of the selected item.

```
with Outline1 do
  SelectedItem := Items[SelectedItem].GetNextChild(SelectedItem);
```

See also
GetFirstChild method, *GetLastChild* method, *GetPrevChild* method

GetParentForm function — Forms

Declaration

```
function GetParentForm(Control: TControl): TForm;
```

The *GetParentForm* function returns the form that contains the control specified in the *Control* parameter. If the specified control is not on a form, *GetParentForm* returns **nil**.

If you'd rather have the function return an exception when the specified control is not on a form, use the *ValidParentForm* function.

Example
The following code shows the form that contains *Button2*:

```
GetParentForm(Button2).Show;
```

See also
ValidParentForm function

GetPassword method

Applies to
TSession component

Declaration

```
function GetPassword: Boolean;
```

The *GetPassword* method invokes the *OnPassword* event (if any) or displays the default password dialog box. It then returns *True* if the user chose the OK button and *False* if the user chose the Cancel button.

Example

```
  Session.GetPassword;
```

See also
Session variable

GetPrevChild method

Applies to
TOutlineNode object

Declaration

```
function GetPrevChild(Value: Longint): Longint;
```

The *GetPrevChild* method returns the *Index* value of the previous outline item that shares the same parent item as the item that has an *Index* value equal to the *Value* parameter. This is useful when the previous sibling has subitems, thus its index is not simply one less than *Value*. If the item indexed by *Value* has no previous sibling, *GetPrevChild* returns -1.

Example
The following code tests to determine if the selected item has a previous sibling. The results are displayed in a label.

```
  with Outline1 do
    if (Items[SelectedItem].GetPrevChild > -1) then
      Label1.Caption := 'Has a prior sibling'
    else
      Label1.Caption := 'Has no prior sibling';
```

See also
GetFirstChild method, *GetLastChild* method, *GetNextChild* method

GetPrinter method

Applies to
TPrinter object

Declaration

```
procedure GetPrinter (ADevice, ADriver, APort: PChar; var ADeviceMode: THandle);
```

The *GetPrinter* method retrieves the current printer. You should rarely need to call this method and should instead access the printer you want in the *Printers* property array. For more information, see the Windows API *CreateDC* function.

See also
SetPrinter method

GetProfileChar function — SysUtils

Declaration

```
function GetProfileChar(Section, Entry: PChar; Default: Char): Char;
```

GetProfileChar loads a single character from the given section and item of WIN.INI.

GetProfileStr function — SysUtils

Declaration

```
function GetProfileStr(Section, Entry: PChar; const Default: string): string;
```

GetProfileStr loads a string value from the given section and item of WIN.INI. This function is used by *GetFormatSettings*.

GetResults method

Applies to
TParam object

Declaration

```
procedure GetResults;
```

You only need to call this method with a Sybase stored procedure that returns a result set. *GetResults* returns the output parameter values from the stored procedure. Usually, *TStoredProc* does this automatically, but Sybase stored procedures do not return the values until the cursor reaches the end of the result set, so you must call *GetResults* explicitly.

Example

```
StoredProc1.Open
while not EOF do
begin
  StoredProc1.Next;
  {Do Something}
end;
StoredProc1.GetResults;
Edit1.Text := StoredProc1.ParamByName('Output');
```

GetSelTextBuf method

Applies to

TDBEdit, *TDBMemo*, *TEdit*, *TMaskEdit*, *TMemo* components

Declaration

```
function GetSelTextBuf(Buffer: PChar; BufSize: Integer): Integer;
```

The *GetSelTextBuf* method copies the selected text from the edit box or memo control into the buffer pointed to by *Buffer*, up to a maximum of *BufSize* characters, and returns the number of characters copied.

You should need to use the *GetSelTextBuf* method only if you are working with strings longer than 255 characters. Because an Object Pascal style string has a limit of 255 characters, such properties as *Text* for an edit box, *Items* for a list box, and *Lines* for a memo control do not allow you to work with strings longer than 255 characters. *GetSelTextBuf* and the corresponding *SetSelTextBuf* methods use null-terminated strings that can be up to 64K in length.

Example

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Buffer: PChar;
  Size: Integer;
begin
  Size := Edit1.SelLength;          {Get length of selected text in Edit1}
  Inc(Size);                        {Add room for null character}
  GetMem(Buffer, Size);             {Creates Buffer dynamic variable}
  Edit1.GetSelTextBuf(Buffer,Size); {Puts Edit1.Text into Buffer}
  Edit2.Text := StrPas(Buffer);     {Converts string in Buffer into Pascal-style string}
  FreeMem(Buffer, Size);            {Frees memory allocated to Buffer}
end;
```

See also

GetTextBuf method, *SelText* property, *SetSelTextBuf* method

GetShortHint function **Controls**

Declaration

```
function GetShortHint(const Hint: string): string;
```

The *GetShortHint* function returns the first part of the two-part string specified as the value of the *Hint* property. The first part of the string is the text following the | character. If the *Hint* string value is not separated into two parts, *GetShortHint* returns the entire *Hint* string.

Example

This code assigns a two-part string as the *Hint* property of an edit box and then displays the “short” or first part of the string as the text of the edit box:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  Edit1.Hint := 'Name|Enter full name';
  Edit1.Text := GetShortHint(Edit1.Hint);
end;
```

See also

GetLongHint function, *OnHint* event, *ShowHint* property

GetStoredProcNames method

Applies to

TSession component

Declaration

```
procedure GetStoredProcNames(const DatabaseName: string; List: TStrings);
```

GetStoredProcNames returns a list of all stored procedures defined for the specified SQL database. This method is not valid for Paradox or dBASE databases.

Example

```
Session.GetStoredProcNames(‘IB_EMPLOYEE’, MyStringList);
```

See also

Session variable

GetTableNames method

Applies to
TSession component

Declaration

```
procedure GetTableNames(const DatabaseName, Pattern: string;
  Extensions, SystemTables: Boolean; List: TStrings);
```

The *GetTableNames* method clears *List* and then adds to it the names of all the tables in the database referenced by *DatabaseName*. The *Pattern* parameter will limit the table names to those matching *Pattern*.

For SQL servers, set *SystemTables* to *True* to obtain system tables in addition to user tables. For desktop (non-SQL) databases, set *Extensions* to *True* to include file-name extensions in the table names.

Example

```
  Session.GetTableNames('DBDEMOS', False, False, MyStringList);
```

See also
Session variable

GetText method

Applies to
TStrings, *TStringList* objects
Declaration

```
function GetText: PChar;
```

The *GetText* method returns a string list as a null-terminated string. *GetText* is useful when working with components that contain blocks of text made up of more than one string. For example, a memo component (*TMemo*) can contain multiple strings. When you want to return the entire list of strings in a memo component all at once, use the *GetText* method.

Example
The following code returns the text in the items of an outline to one variable called *MyVar*.

```
  MyVar := Outline1.Lines.GetText;
```

See also
SetText method

GetTextBuf method

Applies to

All controls; *TClipboard* object

Declaration

```
function GetTextBuf(Buffer: PChar; BufSize: Integer): Integer;
```

The *GetTextBuf* method retrieves the control's text and copies it into the buffer pointed to by *Buffer*, up to the number of characters given by *BufSize*, and returns the number of characters copied.

The resulting text in *Buffer* is a null-terminated string.

To find out how many characters the buffer needs to hold the entire text, you can call the *GetTextLen* method before calling *GetTextBuf*.

Usually you need to use *GetTextBuf* and the corresponding *SetTextBuf* only when working with strings longer than 255 characters. Because Object Pascal strings have a limit of 255 characters, such properties as *Text* for an edit box, *Items* for a list box, and *Lines* for a memo control only allow you to work with strings up to 255 characters. *GetTextBuf* and *SetTextBuf* use null-terminated strings that can be up to 64K in length.

Example

This example copies the text in an edit box into a null-terminated string, and puts this string in another edit box when the user clicks the button on the form.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Buffer: PChar;
  Size: Byte;
begin
  Size := Edit1.GetTextLen;       {Get length of string in Edit1}
  Inc(Size);                      {Add room for null character}
  GetMem(Buffer, Size);           {Creates Buffer dynamic variable}
  Edit1.GetTextBuf(Buffer,Size);  {Puts Edit1.Text into Buffer}
  Edit2.Text := StrPas(Buffer);   {Converts Buffer to a Pascal-style string]
  FreeMem(Buffer, Size);          {Frees memory allocated to Buffer}
end;
```

See also

GetSelTextBuf method, *SetTextBuf* method

GetTextItem method

Applies to

TOutline component

Declaration

```
function GetTextItem(Value: string): Longint;
```

The *GetTextItem* method returns the *Index* value of the first outline item that contains the string specified in the *Value* parameter in its *Text* property. Use *GetTextItem* when you want to know which outline item is identified by a string.

Example

The following code returns the index of the outline item that contains the text 'Perry' to a variable called *PerryIndex*.

```
PerryIndex := Outline1.GetTextItem('Perry');
```

See also

GetDataItem method, *GetItem* method

GetTextLen method

Applies to

All controls

Declaration

```
function GetTextLen: Integer;
```

The *GetTextLen* method returns the length of the control's text. The most common use of *GetTextLen* is to find the size needed for a text buffer in the *GetTextBuf method*.

Example

This example uses two edit boxes and a button on a form. When the user clicks the button, the length of the text in the *Edit1* is displayed in *Edit2*.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Size: Integer;
begin
  Size := Edit1.GetTextLen;
  Edit2.Text := ('Edit1 has ' + IntToStr(Size) + 'characters in it');
end;
```

See also

GetTextBuf method

Glyph property

Applies to

TBitBtn, *TSpeedButton* controls

Declaration

```
property Glyph: TBitmap;
```

The *Glyph* property specifies the bitmap that appears on the selected bitmap button or on a speed button. Use the Open dialog box that appears as an editor in the Object Inspector to choose a bitmap file (with a .BMP extension) to use on the button, or specify a bitmap file at run time.

You can provide up to four images on a bitmap button or speed button with a single bitmap. Delphi then displays one of these images depending on the state of the button.

Image position in bitmap	Button state	Description
First	Up	This image appears when the button is unselected. If no other images exist in the bitmap, Delphi also uses this image for all other images.
Second	Disabled	This image usually appears dimmed to indicate that the button can't be selected.
Third	Down	This image appears when a button is clicked. The up state image reappears when the user releases the mouse button.
Fourth	Stay down	This image appears when a button stays down indicating that it remains selected. (This fourth state applies only to speed buttons.)

If only one image is present, Delphi attempts to represent the other states by altering the image slightly for the different states, although the stay down state is always the same as the up state. If you aren't satisfied with the results, you can provide one or more additional images in the bitmap.

If you have multiple images in a bitmap, you must specify the number of images that are in the bitmap with the *NumGlyphs* property. All images must be the same size and next to each other in a horizontal row.

Example

This example uses a bitmap button on a form. When the application runs and the form is created, a bitmap is placed on the bitmap button.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  BitBtn1.Glyph.LoadFromFile('TARTAN.BMP');
end;
```

These lines of code load a four-image bitmap into the *Glyph* property of a speed button, and specify the appropriate value for the *NumGlyphs* property:

```
SpeedButton1.Glyph.LoadFromFile('MYBITMAP.BMP');
SpeedButton1.NumGlyphs := 4;
```

See also

Kind property, *Layout* property, *Margin* property, *ModalResult* property, *NumGlyphs* property, *Spacing* property, *TBitmap* object

GotoBookmark method

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure GotoBookmark(Bookmark: TBookmark);
```

The *GotoBookmark* method moves the cursor to the record corresponding to the bookmark obtained through a call to *GetBookmark*. While you must eventually call the *FreeBookmark* method to release the resources reserved during the call to *GetBookmark*, you are free to make as many calls to *GotoBookmark* as you wish before calling *FreeBookmark*. If the *Bookmark* parameter is **nil**, *GotoBookmark* does nothing.

Example

```
var MyBookmark: TBookmark;
...
with Table1 do
  begin
{ Save the current record position in MyBookmark }
  MyBookmark := GetBookMark;
  ... { Other code here }
{ Return to the record associated with MyBookmark }
  GotoBookMark(MyBookMark);
{ Release the resources for MyBookmark }
  FreeBookmark(MyBookmark);
  end;
```

GotoCurrent method

Applies to

TTable component

Declaration

```
procedure GotoCurrent(Table: TTable);
```

Use the *GotoCurrent* method to synchronize the positions of two *TTable* components that use the same database table. *GotoCurrent* changes the position of the table to match that of the *Table* parameter.

Note Both tables must have the same *DatabaseName* and *TableName* or a "table mismatch" exception will be raised.

Example

```
Table1.GotoCurrent(Table2);
```

GotoKey method

Applies to

TTable component

Declaration

```
function GotoKey: Boolean;
```

The *GotoKey* method is used with the *SetKey* or *EditKey* method to move to a specific record in a *TTable*. Call *SetKey* to put the table in *SetKey* state. In *SetKey* state, assignments to fields indicate values to search for in indexed fields. *GoToKey* then moves the cursor to the first row in the table that matches those field values.

GoToKey is a *Boolean* function that moves the cursor and returns *True* if the search is successful. If the search is unsuccessful, it returns *False* and does not change the position of the cursor.

Note If you want to search on a subset of fields in a multiple-field key, you must set the *KeyFieldCount* property to the number of fields on which you want to search.

Example

```
with Table1 do
  begin
  EditKey;
  FieldByName('CustNo').AsFloat := 610;
  GotoKey;
  end;
```

See also

FindKey method

GotoNearest method

Applies to

TTable component

Declaration

```
procedure GotoNearest;
```

The *GotoNearest* method is used with the *EditKey* or *SetKey* method to move to a record in the *dataset* whose index fields are greater than or equal to the *IndexFields* property. Call *SetKey* first to put the *TTable* in *SetKey* state, modify the fields of the key, and finally call *GotoNearest* to perform the move.

The *KeyExclusive* property indicates whether a search will position the cursor on or after the specified record being searched for.

Note You do not have to assign a value for each field in the index key. Any unassigned field will use a NULL value.

The search begins at the first record in the table, not at the current cursor position.

Example

```
with Table1 do
  begin
  SetKey;
  FieldByName('State').AsString := 'CA';
  FieldByName('City').AsString := 'Santa';
  GotoNearest;
  end;
```

See also

GotoKey method, *KeyFieldCount* property, *SetKey* method

GotoXY procedure **WinCrt**

Declaration

```
procedure GotoXY(X, Y: Byte);
```

The *GotoXY* procedure moves the cursor to specified coordinates (X,Y) within the virtual screen.

The upper left corner of the virtual screen corresponds to (1, 1).

Use *CursorTo* instead of *GotoXY* when developing new applications.

Example

```
uses WinCrt;

var
  C: PChar;

begin
  GotoXY(10,10);
  Writeln('Hello');
end;
```

See also

CursorTo procedure, *WhereX* function, *WhereY* function

Graphic property

Applies to

TPicture object

Declaration

```
property Graphic: TGraphic;
```

The *Graphic* property specifies the graphic that the picture contains. The graphic can be a bitmap, icon, or metafile.

Example

The following code draws the graphic in *Picture1* in the top-left corner of the canvas of *Form1*.

```
   Form1.Canvas.Draw(0,0 Picture1.Graphic);
```

See also

Bitmap property, *Icon* property, *Metafile* property

GraphicExtension function — Graphics

Declaration

```
function GraphicExtension(GraphicClass: TGraphicClass): string;
```

The *GraphicExtension* function returns the file-name extension of the graphics object specified by the *GraphicClass* parameter. The *TGraphicClass* type is simply a container class for the *TBitmap*, *TGraphic*, *TIcon*, and *TMetafile* objects. These are the file extensions returned for each graphics class:

Graphic class	File extension returned
TGraphic	.BMP, .ICO, or .WFM
TBitmap	.BMP
TIcon	.ICO
TMetafile	.WFM

Example

The following code tests to determine if the graphic in *Picture1* is an icon. If so, the minimized icon of *Form1* is set to the graphic.

```
   if GraphicExtension(Picture1.Graphic)='.ICO' then
     Form1.Icon := Picture1.Graphic;
```

See also

GraphicFilter function

GraphicFilter function — Graphics

Declaration

```
function GraphicFilter(GraphicClass: TGraphicClass): string;
```

The *GraphicFilter* function returns a filter string compatible with the *Filter* property value of an Open or Save dialog box. The *GraphicClass* parameter can be one of these values: *TBitmap*, *TGraphic*, *TIcon*, or *TMetafile*. These are the strings that are returned for each class:

Graphic class	Filter string returned
TBitmap	Bitmaps (*.BMP) | *.BMP
TIcon	Icons (*.ICO) | *.ICO
TMetafile	Metafiles (*.WMF) | *.WMF
TGraphic	All (*.BMP; *.WMF; *.ICO) | *.BMP; *.WMF; *.ICO | Bitmaps (*.BMP | *.BMP | Metafiles (*.WFM | *.WMF | Icons (*.ICO) | *.ICO

Example

This code displays an Open dialog box with the *TBitmap* filter string in the List Files of Type combo box:

```
OpenDialog1.DefaultExt := GraphicExtension(TBitmap);
OpenDialog1.Filter := GraphicFilter(TBitmap);
if OpenDialog1.Execute then
  ...
```

See also

GraphicExtension function, *TOpenDialog* component, *TSaveDialog* component

GridHeight property

Applies to

TDrawGrid, *TStringGrid* components

Declaration

```
property GridHeight: Integer;
```

Run-time and read only. The *GridHeight* property is the height of the grid in pixels. If the grid is too tall to be fully displayed causing the user to scroll to see its entire contents, the value of *GridHeight* is the same as the *ClientHeight* property value for the grid.

Example

This example uses a string grid and a label on a form. The height of the grid appears in the caption of the label.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  ARow, ACol: Integer;
begin
  with StringGrid1 do
  begin
    for ARow := 1 to RowCount - 1 do
```

```
        for ACol := 1 to ColCount - 1 do
          Cells[ARow, ACol] := 'Delphi';
    end;
    Label1.Caption := IntToStr(StringGrid1.GridHeight) + ' pixels';
  end;
```

See also

GridWidth property

GridLineWidth property

Applies to

TDrawGrid, *TStringGrid* components

Declaration

```
property GridLineWidth: Integer;
```

The *GridLineWidth* property determines the width of the lines between the cells in the grid. The default value is 1 pixel. Larger values create heavier lines.

Example

This example includes a draw grid on a form. When the application runs and the form is created, the width of the lines on the draw grid changes if the default column width of the grid is over 90 pixels wide:

```
  procedure TForm1.FormCreate(Sender: TObject);
  begin
    with DrawGrid1 do
    begin
      if DefaultColWidth > 90 then
        GridLineWidth := 2
      else
        GridLineWidth := 1;
     end;
  end;
```

See also

ColWidths property, *DefaultColWidth* property, *GridHeight* property, *GridWidth* property, *RowHeights* property

GridWidth property

Applies to

TDrawGrid, *TStringGrid* components

Declaration

```
property GridWidth: Integer;
```

Run-time and read only. The *GridWidth* property is the width of the grid in pixels. If the grid is too wide to be fully displayed causing the user to scroll it to see its entire contents, the value of *GridWidth* is the same as the *ClientWidth* property value for the grid.

Example

This example uses a string grid and a label on a form. The label reports the width of the grid.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  ARow, ACol: Integer;
begin
  with StringGrid1 do
  begin
    for ARow := 1 to RowCount - 1 do
      for ACol := 1 to ColCount - 1 do
        Cells[ARow, ACol] := 'Pascal';
  end;
  Label1.Caption := IntToStr(StringGrid1.GridWidth) + ' pixels';
end;
```

See also

GridHeight property

GroupIndex property

Applies to

TMenuItem, *TSpeedButton* components

For speed button controls

Declaration

```
property GroupIndex: Integer;
```

The *GroupIndex* property determines which speed buttons work together as a group.

By default, speed buttons have a *GroupIndex* property value of 0, indicating that they do not belong to a group. When the user clicks such a speed button, the button appears “pressed,” or in its down state, then the button returns to its normal up state when the user releases the mouse button.

Speed buttons with the same *GroupIndex* property value (other than 0), work together as a group. When the user clicks one of these speed buttons, it remains “pressed,” or in its down state, until the user clicks another speed button belonging to the same group. Speed buttons used in this way can present mutually exclusive choices to the user.

Example

This code assures that the three speed buttons work together as a group:

```
SpeedButton1.GroupIndex := 1;
SpeedButton2.GroupIndex := 1;
SpeedButton3.GroupIndex := 1;
```

See also

AllowAllUp property, *Down* property

For menu items

Declaration

```
property GroupIndex: Byte;
```

If your application has multiple forms, you'll probably want your application's main menu to change as different forms become active. The alternative is for each form to display its own menu within itself. MDI applications always merge the menus of child windows with the main menu of the parent window. By using the *GroupIndex* property for menu items, you can determine how menus are merged. You can choose to replace or insert menu items in a menu bar.

Each menu item has a *GroupIndex* property value. By default, all menu items in a menu bar have the same *GroupIndex* value, unless you explicitly change them. Each successive menu item in a menu bar must have a *GroupIndex* value equal to or greater than the previous menu item.

Replacing menu items in a menu bar

If a menu item in a menu bar on a form other than the main form has the same *GroupIndex* value as a menu item in a menu bar on the main form, the menu item replaces the menu item in the menu bar of the main form when that form becomes active.

If multiple menu items in the menu bar on the main form have the same *GroupIndex* value, and all menu items of another form also have the same *GroupIndex* value, then the other form's menu items replace all menu items on the menu bar on the main form.

For example, imagine that the menu bar on *Form1* has three items: *One*, *Two*, and *Three*, and all have a *GroupIndex* value of 0. If *Form2* has a menu bar with one menu item, *Four*, with a *GroupIndex* value of 0, when *Form2* becomes active, only the menu item *Four* appears in the menu bar on *Form1*.

Inserting menu items in a menu bar

If one or more menu items in a menu bar on a form that isn't the main form have a *GroupIndex* value greater than a menu item in the menu bar on the main form, those menu items are inserted into the menu bar on the main form when the menus merge. If the item's *GroupIndex* value is greater than all other *GroupIndex* values in the main form's menu bar, the item appears at the end of the menu. If the *GroupIndex* value is between other *GroupIndex* values in the menu bar on the main form, the menu item appears between other menu items, depending on the value.

For example, an item with a *GroupIndex* value of 2 would be inserted between items with *GroupIndex* values of 1 and 3. An item with a *GroupIndex* value of 4 would appear after all the other items.

Note The *GroupIndex* value must be different from all others in the menu bar on the main form, or else the new menu item will replace one or more menu items with the same *GroupIndex* value, which you may or may not want to do.

OLE application menus

When you activate an object created by an OLE 2.0 server application, the server might try to merge its menus with the menus of your container application, depending on the OLE server application. The *GroupIndex* property of each of the container application's menus determines where the merging menu items appear in the container's menu bar. Merged menu items from the OLE server might replace those on the main menu bar, or they might be inserted beside existing container application menu items.

Note See the documentation for the OLE server for information about whether it attempts menu merge during in-place activation.

The OLE server can merge up to three groups of menu items. Each group is distinguished by a unique group index and can contain any number of menu commands. The following table summarizes the menu item groups that the OLE server application can merge:

Group	Index	Description
Edit	1	Menu item(s) from the server for editing the active OLE object
View	3	Menu item(s) from the server for modifying the view of the OLE object.
Help	5	Menu item(s) from the server for accessing the server's online Help

Any menu items in your container application with values of 1, 3, or 5 for their *GroupIndex* properties are replaced by menu items with corresponding index values from the OLE server application. The menu items from your OLE container with a *GroupIndex* value other than 1, 3, or 5 won't be replaced by menus from the server.

See also

AutoMerge property, *FormStyle* property

Halt procedure — System

Declaration

```
procedure Halt [ ( Exitcode: Word ) ];
```

The *Halt* procedure stops the program and returns to the operating system. *Exitcode* is an optional expression that specifies the exit code of your program.

Example

```
begin
   if 1 = 1 then
     begin
       if 2 = 2 then
         begin
           if 3 = 3 then
             begin
               Halt(1); { Halt right here! }
             end;
         end;
     end;
   Canvas.TextOut(10, 10, 'This will not be executed');
 end;
```

See also

Exit procedure, *RunError* procedure

Handle property

Applies to

All windowed controls; *TApplication*, *TBitmap*, *TBrush*, *TCanvas*, *TFont*, *TIcon*, *TMetafile*, *TPen*, *TPrinter* objects; *TDatabase*, *TFindDialog*, *TMainMenu*, *TMenuItem*, *TPopupMenu*, *TQuery*, *TSession*, *TStoredProc*, *TTable* components

For graphics objects

Declaration

```
property Handle: HBitmap;    {for TBitmap objects}

property Handle: HBrush;     {for TBrush objects}

property Handle: HDC;        {for TCanvas objects}

property Handle: HFont;      {for TFont objects}

property Handle: HIcon;      {for TIcon objects}

property Handle: HMetafile;  {for TMetafile objects}

property Handle: HPen;       {for TPen objects}
```

The *Handle* property lets you access the Windows GDI object handle, so you can access the GDI object. If you need to use a Windows API function that requires the handle of a pen object, you could pass the handle from the *Handle* property of a *TPen* object.

For applications, Find and Replace dialog boxes, windowed controls

Declaration

```
property Handle: HWND;
```

Read and run-time only. The *Handle* property gives you access to window handle of the application, the Find and Replace dialog boxes, and all controls in case you need to call a Windows API function that requires a handle.

Example

The following code uses the Windows API function *ShowWindow* to display *Form2* as an icon, but does not activate it.

```
ShowWindow(Form2.Handle, SW_SHOWWINMINNOACTIVE);
```

See also

HandleAllocated method, *HandleNeeded* method

For menu items, main menus, and pop-up menus

Declaration

```
property Handle: HMENU;
```

Read and run-time only. The *Handle* property lets you access the menu or menu item's window handle, so you can call a Windows API function that requires a menu handle.

Example

The following code uses the Windows API function *HiliteMenuItem* to highlight the first menu item in *MainMenu1* on *Form1*.

```
HiliteMenuItem(Form1.Handle, MainMenu1.Handle, 0, MF_BYPOSITION+MF_HILITE);
```

For printer objects

Declaration

```
property Handle: HDC;
```

Read and run-time only. The *Handle* property give you access to the handle of the printer object.

For sessions

Declaration

```
property Handle: HDBISES;
```

Run-time and read only. The *Handle* property allows you to make direct calls to the Borland Database Engine using this handle to the session (*TSession*). Under most circumstances you should not need to use this property, unless your application requires some functionality not encapsulated in the VCL.

For tables, queries, and stored procedures

Declaration

```
property Handle: HDBICur;
```

Run-time and read only. The *Handle* property enables an application to make direct calls to the Borland Database Engine API using this handle of a dataset component.

Under most circumstances you should not need to use this property, unless your application requires some functionality not encapsulated in the VCL.

For databases

Declaration

```
property Handle: HDBIDB;
```

Run-time and read only. Use the *Handle* property to make direct calls to the Borland Database Engine (BDE) API that require a database handle. Under most circumstances you should not need to use this property, unless your application requires some functionality not encapsulated in the VCL.

HandleAllocated method

Applies to
All controls

Declaration

```
function HandleAllocated: Boolean;
```

The *HandleAllocated* method returns *True* if a window handle for the control exists. If no window handle exists, *HandleAllocated* returns *False*. If you query the *Handle* property of a control directly, a handle is automatically created if it didn't previously exist. Therefore, you should call the *HandleAllocated* method if you don't want a handle created automatically for the control, but simply want to know if one exists.

Example
The following code displays the value of the handle of *GroupBox1* if it exists. If not, it displays a message.

```
var
  TheValue: string;
```

```
begin
  if GroupBox1.HandleAllocated then
    TheValue := IntToStr(GroupBox1.Handle)
  else TheValue := 'Handle not allocated.';
  Label1.Caption := TheValue;
end;
```

See also
HandleNeeded method

HandleException method

Applies to
TApplication component

Declaration

```
procedure HandleException(Sender: TObject);
```

The *HandleException* method handles the exceptions for the application. If an exception passes through all the **try** blocks in your application code, your application automatically calls the *HandleException* method, which displays a dialog box indicating an error occurred. To assign other exception handling code for the application, use the *OnException* event handler.

Example
The following code uses the default error handling:

```
try
  { Some code that may produce an exception goes here }
except
  Application.HandleException(Self);
end;
```

See also
Application variable, *OnException* event

HandleNeeded method

Applies to
All controls

Declaration

```
procedure HandleNeeded;
```

The *HandleNeeded* method creates a window handle for the control if one doesn't already exist.

Example

The following code creates a window handle for *Button1*:

```
Button1.HandleNeeded;
```

See also

Handle property, *HandleAllocated* method

HasFormat method

Applies to

TClipboard object

Declaration

```
procedure HasFormat(Format: Word): Boolean;
```

The *HasFormat* method determines if the Clipboard object contains a particular format. If *HasFormat* is *True*, the format is present; if *False*, the format is absent. The Clipboard object keeps a list of available formats in the *Formats* array property.

These are the possible values of the *Format* parameter:

Value	Meaning
CF_TEXT	Text with each line ending with a CR-LF combination. A null character identifies the end of the text.
CF_BITMAP	A Windows bitmap graphic.
CF_METAFILE	A Windows metafile graphic.
CF_PICTURE	An object of type *TPicture*.
CF_OBJECT	Any persistent object.

Example

This example uses a button on a form. When the user clicks the button, a message box appears if there is no text on the Clipboard; otherwise, you don't see anything happen.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if not Clipboard.HasFormat(CF_TEXT) then
      MessageDlg('There is no text on the Clipboard', mtInformation,
        [mbOK],0);
end;
```

See also

Assign method, *FormatCount* property, *Formats* property, *GetComponent* method, *SetComponent* method

HasItems property

Applies to

TOutlineNode object

Declaration

```
property HasItems: Boolean;
```

Run-time and read only. The *HasItems* property determines if an outline item has any subitems. Subitems appear below and indented from their parent item when the parent is expanded. The *Index* value of the parent item is one less than the *Index* value of its first subitem. *HasItems* is *True* if the item has subitems, or *False* if the item has no subitems.

Example

The following code expands the selected item of *Outline1* if it has subitems:

```
  with Outline1[Outline1.SelectedItem] do
    if HasItems then Expand;
```

See also

GetFirstChild method, *GetLastChild* method, *GetNextChild* method, *GetPrevChild* method

Heap variables — System

Declaration

```
var HeapAllocFlags: Word;

var HeapBlock: Word;

var HearLimit: Word;

var HeapError: Pointer;

var HeapCheck: Pointer;
```

The heap manager uses the variables *HeapList*, *HeapLimit*, *HeapBlock*, *HeapError*, *HeapCheck* to implement dynamic memory allocation routines.

Heap variable	Description
HeapAllocFlags	Defines the attribute flags passed to *GlobalAlloc* when the heap manager allocates global blocks. Used with *gmem_Moveable*.
HeapError	Contains the address of a heap-error function that is called whenever the heap manager cannot complete an allocation request.
HeapLimit	Defines the threshold between small and large heap blocks. The default value is 1024.

Heap variable	Description
HeapBlock	Defines the size the heap manager uses when allocating blocks assigned to the sub-allocator. The default value is 8192.
HeapCheck	Contains the address of the heap integrity checking hook. If this pointer is non-**nil**, the allocator/deallocator will call this each time a block is allocated and a block is freed. It is called before the actual allocation or deallocation occurs.

You should have no reason to change the values of *HeapLimit* and *HeapBlock*, but should you decide to do so, make sure that *HeapBlock* is at least 4 times the size of *HeapLimit*.

HeapError is a pointer that points to a function with this header:

```
function HeapFunc (Size: Word): Integer; far;
```

Install the heap-error function by assigning its address to the *HeapError* variable as follows:

```
HeapError := @HeapFunc;
```

The heap-error function gets called whenever a call to *New* or *GetMem* cannot complete the request.

The *Size* parameter contains the size of the block that could not be allocated, and the heap error function should attempt to free a block of a least that size.

Before calling the heap-error function, the heap manager attempts to allocate the block within its sub-allocation free space as well as through a direct call to the Windows *GlobalAlloc* function.

The *HeapError* function returns

- 0 to indicate failure, and causes a run-time error to occur immediately
- 1 to indicate failure, and causes *New* or *GetMem* to return a **nil** pointer
- 2 to indicate success, and causes a retry (which could also cause another call to the heap error function)

See also

GlobalAlloc function, *GlobalLock* function

Height property

Applies to

All controls; *TBitmap*, *TFont*, *TGraphic*, *TIcon*, *TMetafile*, *TPicture* objects; *TForm*, *TScreen* components

Declaration

```
property Height: Integer;
```

For controls, forms, and graphics

The *Height* property of a control is the vertical size of the control, form, or graphic in pixels.

Example

The following code doubles the height of a list box control:

```
ListBox1.Height := ListBox1.Height * 2;
```

See also

ClientHeight property, *SetBounds* method, *Width* property

For the screen

Read and run-time only. The *Height* property of a screen component contains the vertical size of the screen device in pixels.

Example

To following code sets a form's height to half the height of the screen:

```
Form1.Height := Screen.Height div 2;
```

See also

Screen variable, *Width* property

For fonts

The *Height* property is the height of the font in pixels. It is the size of the font plus the font's internal leading. If you are concerned with the size of the font on the screen—the number of pixels the font needs—use the *Height* property. If you want to specify a font's size using points, use the *Size* property instead.

Delphi calculates *Height* using this formula:

```
Font.Height = -Font.Size * Font.PixelsPerInch / 72
```

Therefore, whenever you enter a point size in the *Height* property, you'll notice the *Size* property changes to a negative value. Conversely, if you enter a positive *Size* value, the *Height* property value changes to a negative value.

Example

This example uses button and a label on a form. When the user clicks the button, the height of the font changes to 36 pixels on the screen:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Font.Height := 36;
end;
```

See also
Font property, *PixelsPerInch* property, *Size* property

HelpCommand method

Applies to
TApplication component

Declaration

```
function HelpCommand(Command: Word; Data: Longint): Boolean;
```

The *HelpCommand* method gives you quick access to any of the Help commands in the WinHelp API (application programming interface). For information about the commands you can call and the data passed to them, see the WinHelp topic in the Help system.

Example
This example uses a bitmap button on a form. When the user clicks the button, the Help contents screen of the specified Help file appears.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  Application.HelpFile := 'MYHELP.HLP';
  Application.HelpCommand(HELP_CONTENTS, 0);
end;
```

See also
Application variable, *HelpContext* method, *HelpContext* property, *HelpFile* property, *HelpJump* method

HelpContext method

Applies to
TApplication component

Declaration

```
function HelpContext(Context: THelpContext): Boolean;
```

The *HelpContext* method calls WinHelp, the Windows Help system program, if the *HelpFile property* is assigned a file to use for Help. *HelpContext* passes the file name contained in *HelpFile* and the context number passed in *Context* parameter. For example, if you specify the *Context* value as 714, the *HelpContext* method displays the screen with the context help ID of 714 in the Help file.

HelpContext returns *False* if *HelpFile* is an empty string, meaning the application has no Help file assigned. In all other cases, *HelpContext* returns *True*.

Example

This example uses a bitmap button on a form. When the user clicks the button, the screen with the context number of 714 in the DATA.HLP Help file appears:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  Application.HelpFile := 'DATA.HLP';
  Application.HelpContext(714);
end;
```

See also

Application variable, *HelpFile* property, *OnHelp* event

HelpContext property

Applies to

All controls; *Exception*, *TColorDialog*, *TFindDialog*, *TFontDialog*, *TMenuItem*, *TPopupMenu*, *TOpenDialog*, *TPrintDialog*, *TPrinterSetupDialog*, *TReplaceDialog*, *TSaveDialog* components

Declaration

```
property HelpContext: THelpContext;
```

The *HelpContext* property provides a context number for use in calling context-sensitive online Help. Each screen in the Help system should have a unique context number. When a component is selected in the application, pressing *F1* displays a Help screen. Which Help screen appears depends on the value of the *HelpContext* property.

Example

The following code associates a Help file with the application, and makes the screen with a context number of 7 the context-sensitive Help screen for the *Edit1* edit box:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  Application.HelpFile := 'MYHELP.HLP';
  Edit1.HelpContext := 7;
end;
```

See also

HelpContext method, *HelpFile* property, *HelpJump* method, *OnHelp* event

HelpFile property

Applies to

TApplication component

Declaration

```
property HelpFile: string;
```

Run-time only. The *HelpFile* property holds the name of the file the application uses to display online Help. By default, *HelpFile* is a null string, and the application's *Help* method ignores attempts to display Help. If *HelpFile* contains anything, the *HelpContext* method passes it to the Windows Help system as the name of the file to use for Help.

Example

To specify the MYHELP.HLP file as the Help file for your application, use this line of code:

```
Application.HelpFile := 'MYHELP.HLP';
```

See also

HelpContext method

HelpJump method

Applies to

TApplication component

Declaration

```
function HelpJump(const JumpID: string): Boolean;
```

The *HelpJump* method calls WinHelp, the Windows Help system program, if the *HelpFile property* is assigned a file to use for Help. *HelpJump* passes the file name contained in *HelpFile* and the context string specified in the *JumpID* parameter. For example, if you specify the *JumpID* value as 'vclDefaultProperty', the *HelpJump* method displays the screen in the Help file that has the context string 'vclDefaultProperty'.

HelpJump returns *False* if *HelpFile* is an empty string, meaning the application has no Help file assigned. In all other cases, *HelpJump* returns *True*.

Example

This example uses a bitmap button on a form. When the user clicks the button, the Help screen describing the *Default* property in the DELPHI.HLP file appears, because the *Default* property screen has the a *JumpID* string of 'vclDefaultProperty'.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  Application.HelpFile := 'DELPHI.HLP';
  Application.HelpJump('vclDefaultProperty');
end;
```

See also

Application variable, *HelpCommand* method, *HelpContext* method, *HelpContext* property, *HelpFile* property, *OnHelp* event

Hi function

System

Declaration

```
function Hi(X): Byte;
```

The *Hi* function returns the high-order byte of *X* as an unsigned value. *X* is an expression of type *Integer* or *Word*.

Example

```
var B: Byte;
begin
  B := Hi($1234);   { $12 }
end;
```

See also

Lo function, *Swap* function

Hide method

Applies to

TForm component, All controls

Declaration

```
procedure Hide;
```

The *Hide* method makes a form or control invisible by setting the *Visible* property of the form or control to *False*. Although a form or control that is hidden is not visible, you can still set the properties of the form or control, or call its methods.

Example

This code uses a button and a timer on a form. When the user clicks the button, the form disappears for the period of time specified in the *Interval* property of the timer control, then the form reappears:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Timer1.Enabled := True;
  Hide;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Visible := True;
  Timer1.Enabled := False;
end;
```

See also
Close method, *Show* method, *ShowModal* method

HideSelection property

Applies to
TEdit, *TMemo* components

Declaration

```
property HideSelection: Boolean;
```

The *HideSelection* property determines whether text that is selected in an edit or memo remains selected when the focus shifts to another control. If True, the text is no longer selected until the focus returns to the control. If False, the text remains selected. The default value is *True*.

Example
This example uses an edit box and a memo on a form. When the user jumps from one control to the other, selected text remains selected in the memo, but not in the edit box.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Edit1.HideSelection := True;
  Memo1.HideSelection := False;
end;
```

See also
AutoSelect property

High function — System

Declaration

```
function High(X);
```

The *High* function returns the highest value in the range of the argument.

The result type is *X*, or the index type of *X*.

X is either a type identifier or a variable reference. The type denoted by *X*, or the type of the variable denoted by *X*, must be one of the following types.

For this type	High returns
Ordinal type	The highest value in the range of the type
Array type	The highest value within the range of the index type of the array
String type	The declared size of the string

For this type	High returns
Open array	The value, of type *Word*, giving the number of elements in the actual parameter minus one
String parameter	The value, of type *Word*, giving the number of elements in the actual parameter minus one

Example

```
function Sum( var X: array of Double): Double;
var
  I: Word;
  S: Double;
begin
  S := 0;  { Note that open array index range is always zero-based. }
  for I := 0 to High(X) do S := S + X[I];
  Sum := S;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  List1: array[0..3] of Double;
  List2: array[5..17] of Double;
  X: Word;
  S, TempStr: string;
begin
  for X := Low(List1) to High(List1) do
    List1[X] := X * 3.4;
    for X := Low(List2) to High(List2) do
    List2[X] := X * 0.0123;
  Str(Sum(List1):4:2, S);
  S := 'Sum of List1: ' + S + #13#10;
  S := S + 'Sum of List2: ';
  Str(Sum(List2):4:2, TempStr);
  S := S + TempStr;
  MessageDlg(S, mtInformation, [mbOk], 0);
end;
```

See also

Low function

HInstance and HPrevInst variables — System

Declaration

```
var HInstance: Word;
```

The *HInstance* variable contains the instance handle of the application or library as provided by the Windows environment.

Hint property

Applies to

All controls; *TApplication*, *TMenuItem* components

Declaration

```
property Hint: string;
```

Description

The *Hint* property is the text string that can appear when the *OnHint* event occurs, which happens when the user moves the mouse pointer over a control or menu item. The code within the *OnHint* event handler determines how the string is displayed. A common use of an *OnHint* event handler is to display the hint as the caption of a panel component that is being used as a status bar.

H

You can have a Help Hint, a box containing help text, appear for a control when the user moves the mouse pointer over the control and pauses momentarily. This is how:

1. Specify a *Hint* value for each control you want a Help Hint to appear for.
2. Set the *ShowHint* property of each control to *True*.
3. At run time, set the value of application's *ShowHint* property to *True*.

You can specify a hint to be used for both for a Help Hint box and in an *OnHint* handler (as the application's *Hint* property value) by specifying two values separated by a | character (the "or" or "pipe" symbol). For example,

```
Edit1.Hint := 'Name|Enter Name in the edit box';
```

The 'Name' string appears in the Help Hint box, and the 'Enter full name' string appears as specified in the *OnHint* event handler.

If you specify just one value, it can be used both as a Help Hint and as the *Hint* property of the application. If the application's *ShowHint* property is *False*, the Help Hint won't appear, but the other hint still will.

If a control has no *Hint* value specified, but its parent control does, the control uses the *Hint* value of the parent control as long as the control's *ShowHint* property is *True*.

Example

This example uses an edit box and a list box on a form. Items are added to the list box and a Help Hint is assigned to both controls. The last statement enables the Help Hints for the entire application.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  Edit1.Hint := 'Enter your name';
  Edit1.ShowHint := True;
  with ListBox1 do
  begin
```

```
      for I := 1 to 10 do
        Items.Add('Item ' + IntToStr(I));
      Hint := 'Select an item';
      ShowHint := True;
    end;
    Application.ShowHint := True;
  end;
```

To see a example that displays the hints of controls in some place other than in a Help Hint box, see the *OnHint* example.

See also

GetLongHint function, *GetShortHint* function, *HintColor* property, *HintPause* property, *ShowHint* property for controls, *ShowHint* property for the application

HintColor property

Applies to

TApplication component

Declaration

```
property HintColor: TColor
```

Description

Run time only. The *HintColor* property determines the color of the hint boxes for the Help Hints of the controls in the application. For a table of possible color values, see the *Color* property.

Example

This example includes an control that has a *Hint* property value and has its ShowHint property value set to *True*. When the application runs and the user places the mouse cursor over the control, a Help Hint appears for the control in an aqua hint box after a delay of 1000 milliseconds:

```
  procedure TForm1.FormCreate(Sender: TObject);
  begin
    Application.ShowHint := True;
    Application.HintColor := clAqua;
    Application.HintPause := 1000;
  end;
```

See also

Application variable, *Hint* property, *HintPause* property, *ShowHint* property, *TColor* type

HintPause property

Applies to

TApplication component

Declaration

```
property HintPause: Integer;
```

The *HintPause* property determines the time interval that passes when the user places the mouse pointer on a control before the control's Help Hint specified in its *Hint* property appears. The interval is in milliseconds. The default value is 800 milliseconds.

Example

This example includes an control that has a *Hint* property value and has its ShowHint property value set to *True*. When the application runs and the user places the mouse cursor over the control, a Help Hint appears for the control in an aqua hint box after a delay of 1000 milliseconds:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.ShowHint := True;
  Application.HintColor := clAqua;
  Application.HintPause := 1000;
end;
```

See also

Application variable, *Hint* property, *HintColor* property, *ShowHint* property

Hints property

Applies to

TDBNavigator component

Declaration

```
property Hints: TStrings;
```

The *Hints* property allows you to customize the Help Hints for the buttons on the database navigator. Each hint is a string. The first string in the string object becomes the Help Hint for the first button on the navigator. The seventh hint becomes the Help Hint for the seventh button (the Edit button).

If you don't want to change the Help Hint for every button, enter an empty string ('') for the Help Hint you want to stay the same, or simply leave the line blank if you are using the string list property editor of the Object Inspector for the *Hints* property.

Example

This example uses a database navigator and a button on a form. When the user clicks the button, the Help Hints for the navigator are modified.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NewStrings: TStringList;
begin
  NewStrings := TStringList.Create;
  with NewStrings do
  begin
    Add('Beginning of dataset');
    Add('Previous record');
    Add('');
    Add('End of dataset');
  end;
  DBNavigator1.Hints := NewStrings;
  DBNavigator1.ShowHint := True;
end;
```

See also

Hint property, *HintColor* property, *HintPause* property

HistoryList property

Applies to

TOpenDialog, *TSaveDialog* components

Declaration

```
property HistoryList: TStrings ;
```

The *HistoryList* property contains strings that appear in the File Name drop-down combo box of an Open or Save dialog box when the user opens it. Because only a File Name combo box can have a value for the *HistoryList* property, the *FileEditStyle* property value of the dialog box must be *fsComboBox*. If the *FileEditStyle* property value is *fsEdit*, the strings in the *HistoryList* property aren't used by the dialog box.

Your application can use the *HistoryList* property to create a list of previous files names opened or saved with the dialog box. Use a *TStringList* object to keep a list of file names, and assign this object to the *HistoryList* property.

Note When an Open or Save dialog box is open, your application won't be able to access the *HistoryList* property. Therefore, your application must work with *HistoryList* before the dialog box opens or after it closes.

Example

This example uses an Open dialog box and a button. The code creates a string list object and stores each file the user selects in the Open dialog box in it. Each time the clicks the button to open the dialog box, the string list is assigned to the *HistoryList* property.

```
var
  OldFiles: TStringList;

procedure TForm1.Button1Click(Sender: TObject);
var
  SelectedFile: string;
begin
  if OpenDialog1.Execute then
    SelectedFile := OpenDialog1.FileName;
  OldFiles.Add(SelectedFile);
  OpenDialog1.HistoryList := OldFiles;
end;

initialization
  OldFiles := TStringList.Create;
end.
```

HMetafile type — Graphics

Declaration

```
HMETAFILE = THandle;
```

HMetafile is the handle of a *TMetafile* object.

HorzScrollBar property

Applies to

TForm, TScrollBox components

Declaration

```
property HorzScrollBar: TControlScrollBar;
```

The *HorzScrollBar* property is the form's or scroll box's horizontal scroll bar. The values of *HorzScrollBar's* nested properties determines how the horizontal scroll bar behaves.

To make a horizontal scroll bar appear on a form or scroll box, the nested properties of *HorzScrollBar* must be set like this:

- *Visible* must be *True*.
- The value of the *Range* property must be greater than the value of the *ClientWidth* property of the form or the *Width* property of the scroll box.

Example

This example implements a horizontal scroll bar on the form. The scroll bar scrolls the form 100 pixels more than the form width:

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

```
    ClientWidth := 300;
    with HorzScrollBar do
    begin
      Increment := 4;
      Position := 0;
      Range := ClientWidth + 100;     {Range must be greater than the form's client width}
      Visible := True;
    end;
  end;
```

See also

AutoScroll property, *Increment* property, *Position* property, *Range* property, *ScrollPos* property, *VertScrollBar* property, *Visible* property

HPrevInst variable — System

Declaration

```
var HPrevInst: Word;
```

In a program, the *HPrevInst* variable contains the handle of the previous instance of the application, or 0 if there are no previous instances. In a library,*HPrevInst* is always zero.

Icon property

Applies to

TPicture object; *TApplication*, *TForm* components

For forms

Declaration

```
property Icon: TIcon
```

The *Icon* property determines the icon that is displayed when the window or form is minimized. If you don't assign a specific icon to *Icon*, the form uses the application's icon.

Example

This code assigns an icon to a form when the form is created:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Icon.LoadFromFile('MYICON.ICO');
end;
```

See also
LoadFromFile method, *SaveToFile* method

For picture objects

Declaration

```
property Icon: TIcon
```

The *Icon* property specifies the contents of the *TPicture* object as an icon graphic (.ICO file format). If *Icon* is referenced when the *TPicture* contains a *Bitmap* or *Metafile* graphic, the graphic won't be converted. Instead, the original contents of the *TPicture* are discarded and *Icon* returns a new, blank icon.

Example
The following code allows the user to use a dialog box to redefine the icon for the application at run time. When the user clicks *Button1*, *OpenDialog1* executes and the user specifies an icon file name. The file is loaded into the *Icon* property of the *TheIcon* picture object. Then, the *Icon* of *TheIcon* is assigned to the *Icon* of *Form1*.

```
procedure TForm1.Button1Click(Sender: TObject);
var TheIcon: TPicture;
begin
  OpenDialog1.FileName := '*.ICO';
  if OpenDialog1.Execute then
  begin
    TheIcon := TPicture.Create;
    TheIcon.LoadFromFile(OpenDialog1.FileName);
    Form1.Icon := TheIcon.Icon;
  end;
end;
```

See also
Graphic property

For an application

Declaration

```
property Icon: TIcon;
```

The value of the *Icon* property determines which icon represents the application when it is minimized or displayed in the Program Manager.

Example
This line of code uses the icon in the MYAPP.ICO files for the application's icon:

```
Application.Icon.LoadFromFile('MYAPP.ICO');
```

See also
Application variable, *LoadFromFile* method, *Minimize* method, *SaveToFile* method

InactiveTitle typed constant — WinCrt

Declaration

```
const InactiveTitle: PChar = '(Inactive %s)';
```

The *InactiveTitle* typed constant points to a null-terminated string to use when constructing the title of an inactive CRT window.

The string is used as the format-control parameter of a call to the Windows *WVSPrintF* function. The *%s* specifier, if present, indicates where to insert the existing window title.

Inc procedure — System

Declaration

```
procedure Inc(var X [ ; N: Longint ] );
```

The *Inc* procedure adds one or *N* to the variable *X*.

X is an ordinal-type variable or a variable of type *PChar* if the extended syntax is enabled and *N* is an integer-type expression.

X increments by 1, or by *N* if *N* is specified; that is, `Inc(X)` corresponds to the statement `X := X + 1`, and `Inc(X, N)` corresponds to the statement `X := X + N`.

Inc generates optimized code and is especially useful in tight loops.

Example

```
var
  IntVar: Integer;
  LongintVar: Longint;
begin
  Inc(IntVar);                                          { IntVar := IntVar + 1 }
  Inc(LongintVar, 5);                           { LongintVar := LongintVar + 5 }
end;
```

See also
Dec procedure, *Pred* function, *Succ* function

Inch property

Applies to
TMetafile object

Declaration

```
property Inch: Word;
```

The *Inch* property value is the number of pixels per inch that are used for the metafile's coordinate mapping. For example, if the metafile was created in a Twips coordinate system (using *MM_TWIPS* mapping), the value of *Inch* is 1440.

Include procedure **System**

Declaration

```
procedure Include(var S: set of T; I:T);
```

The *Include* procedure adds the element *I* to the set *S*.

S is a set type variable, and *I* is an expression of a type compatible with the base type of *S*.

The construct `Include(S,I)` corresponds to `S := S + (I)` but the *Include* procedure generates more efficient code.

See also

Exclude procedure

Increment property

Applies to

TControlScrollBar component

Declaration

```
property Increment: Integer;
```

The *Increment* property determines how many positions the scroll box in a form scroll bar moves when the user clicks one of the small end arrows. The default value is 8.

Example

This example implements a horizontal scroll bar on the form. The scroll bar scrolls the form 100 pixels more than the form width. Each time the user clicks a scroll arrow on the scroll bar, the form scrolls 7 pixels:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ClientWidth := 300;
  with HorzScrollBar do
  begin
    Position := 0;
    Increment := 7;
    Range := ClientWidth + 100;     {Range must be greater than the form's client width}
```

```
      Visible := True;
    end;
  end;
```

See also
HorzScrollBar property, *Position* property, *Range* property, *ScrollPos* property, *VertScrollBar* property

Index property

Applies to
TOutlineNode object; *TBCDField*, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

For outline nodes

Declaration

```
property Index: Longint;
```

The *Index* property uniquely identifies each item of an outline. The first outline item has an *Index* value of 1, and subsequent items are indexed sequentially. If an item has subitems, the *Index* value of the parent item is one less than the *Index* value of its first subitem.

The value of the *Index* property corresponds to the index in the *Items* property array of the *TOutline* component. When an item is added, inserted, or deleted from the outline, the values of the *Index* properties of all subsequent items must be recalculated to be valid. This happens automatically, unless the *BeginUpdate* method has been called.

Example
The following code tests to determine if the selected item is the top item in the outline.

```
  with Outline1 do
    if Items[SelectedItem].Index = 1 then
      { The selected item is the top item }
    else
      { The selected item is not the top item };
```

See also
Add method, *Delete* method, *EndUpdate* method, *Insert* method

For fields

Declaration

```
property Index: Integer;
```

Index is a field's index number into the *Fields* property of the dataset. It corresponds to the order of the field in the dataset. You can change the order of a field's position in the dataset by changing its *Index* property. A better way to change field order, however, is by dragging and dropping fields in the Fields Editor at design time.

IndexDefs property

Applies to

TTable component

Declaration

```
property IndexDefs: TIndexDefs;
```

Run-time and read only. The *IndexDefs* property holds information about all the indexes for the *TTable*.

Note The *IndexDefs* property may not always reflect the current set of indexes. Before examining any property of *IndexDefs*, call its *Update* method to ensure that it has the most recent set of information.

Example

```
{ Get the current available indicies }
Table1.IndexDefs.Update;
{ Find one which combines Customer Number ('CustNo') and Order Number ('OrderNo') }
for I := 0 to Table1.IndexDefs.Count - 1 do
  if Table1.IndexDefs.Items[I].Fields = 'CustNo;OrderNo' then
```

IndexFieldCount property

Applies to

TTable component

Declaration

```
property IndexFieldCount: Integer;
```

Run-time only. The *IndexFieldCount* property is the number of actual fields for the current index. If you are using the primary index for the component, this value will be one. If the component is not *Active*, the value of *IndexFieldCount* will be zero.

Example

```
TotalLen := 0;
with Table1 do
{ Calculate the total length of the index }
  for I := 0 to IndexFieldCount - 1 do
    Inc(TotalLen, IndexFields[I].FieldDef.DataSize);
```

IndexFieldNames property

Applies to

TTable component

Declaration

```
property IndexFieldNames: string;
```

The *IndexFieldNames* property is used with an SQL server to identify the columns to be used as an index for the *TTable*. Separate the column names with semicolon characters (";"). If you have too many column names or the names are too long to fit within the 255 character limit, use column numbers instead of names.

Note *IndexFieldNames* and *IndexName* are mutually exclusive. Setting one will clear the other.

Example

```
Query1.IndexFieldNames := 'CustNo;OrderNo';
```

IndexFields property

Applies to

TTable component

Declaration

```
property IndexFields[Index: Integer]: TField;
```

Run-time only. The *IndexFields* property gives you access to information about each field of the current index for the *dataset*. The *Active* property must be *True* or the information will not be valid.

Example

```
S := '';
with Table1 do
{ Create a composite string with the index's names separated by "@" }
for I := 0 to IndexFieldCount - 1 do
    S := S + '@' + IndexFields[I].FieldName;
```

See also
IndexFieldCount property

IndexName property

Applies to
TTable component

Declaration

```
property IndexName: string;
```

The *IndexName* property identifies a secondary index for the *TTable*. If no value is assigned to *IndexName*, the table's primary index will be used to order the records.

For dBASE tables, the index must reside in the table's master index file. The master index file is determined by taking the *TableName* property and replacing any file extension with "MDX". Non-maintained indexes are not supported.

Note *IndexFieldNames* and *IndexName* are mutually exclusive. Setting one will clear the other.

Example

```
  Table1.IndexName := 'CustNoIndex';
```

See also
MasterFields property, *MasterSource* property

IndexOf method

Applies to
TList, *TStringList*, *TStrings* objects; *TFieldDefs*, *TIndexDefs*, *TMenuItem* components

For menu items

Declaration

```
function IndexOf(Item: TMenuItem): Integer;
```

The *IndexOf* method returns the position of a menu item within a menu. The first position in a menu is 0. If a menu item is not in the menu, *IndexOf* returns -1.

Example
This example uses a main menu named *File1*, a button, and a label on a form. The *File1* menu contains three menu commands, Open, Save, and Close. Delphi automatically names these menu items *Open1*, *Save1*, and *Close1*. This code returns the position of the Close command in the File menu and reports it as the caption of the label.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := IntToStr(File1.IndexOf(Close1));
end;
```

The label displays the number 2, indicating that Close is the third menu command in the File menu. The first menu item, Open, has an index value of 0.

See also

FindItem method

For list objects

Declaration

```
function IndexOf(Item: Pointer): Integer;
```

The *IndexOf* method returns the position of an item in a list kept by the *List* property. The first position in a list is 0. If an item is not in the list, *IndexOf* returns -1.

Example

The following code adds an object to *MyList* if it isn't already in the list.

```
if MyList.IndexOf(MyObject)=-1 then MyList.Add(MyObject);
```

See also

Add method, *Count* property

For string objects

Declaration

```
function IndexOf(const S: string): Integer;
```

The *IndexOf* method returns the position of a string in a list of strings in a string or string list object. Specify the string you want to locate as the value of the *S* parameter. The first position in the list of strings is 0. If the string is not in the string list, *IndexOf* returns -1.

Example

This example uses a combo box that contains five strings (enter them as the value of the *Items* property with the Object Inspector) and a label. When the user selects a string in the combo box, the index of the selected string appears as the caption of the label.

```
procedure TForm1.ComboBox1Click(Sender: TObject);
begin
  Label1.Caption := IntToStr(ComboBox1.Items.IndexOf(ComboBox1.SelText));
end;
```

This example uses a file list box, a directory list box, and a label on a form. When the user uses the directory list box to change directories, a message appears and the color of the form changes if the file AUTOEXEC.BAT is in the new directory. The code is written in the *OnChange* event of the directory list box:

```
procedure TForm1.DirectoryListBox1Change(Sender: TObject);
begin
  FileListBox1.Directory := DirectoryListBox1.Directory;
  if FileListBox1.Items.IndexOf('AUTOEXEC.BAT') > -1 then
  begin
    Color := clYellow;
    Label1.Caption := 'You are in the root directory!';
  end;
end;
```

See also

IndexOfObject method, *Strings* property

For TIndexDefs objects

Declaration

```
function IndexOf(const Name: string): Integer;
```

The *IndexOf* method returns the index of the entry in *Items* whose *Name* property matches the *Name* parameter.

For TFieldDefs objects

Declaration

```
function IndexOf(const Name: string): Integer;
```

The *IndexOf* method returns the index number of the entry in *Items* whose *Name* property matches the *Name* parameter.

IndexOfObject method

Applies to

TStringList, *TStrings* objects

Declaration

```
function IndexOfObject(AObject: TObject): Integer;
```

The *IndexOfObject* method returns the position of an object stored in the *Objects* property of a string object. Specify the object you want to locate as the value of the *AObject* parameter. The first position in the list of objects is 0. If the object is not in the list of objects, *IndexOfObject* returns -1.

Example

The following code determines if *MyObject* is the first object in *MyStringList*.

```
if MyStringList.IndexOfObject(MyObject)=0 then
  { MyObject is the first object in the list };
```

See also

AddObject method, *IndexOf* method, *InsertObject* method

InitialDir property

Applies to

TOpenDialog, *TSaveDialog* components

Declaration

```
property InitialDir: string;
```

The *InitialDir* property determines the current directory when the dialog box first appears and value of the *InitialDir* property is shown as the current directory in the directory tree. Only files in the current directory appear in the dialog box's list box of file names. After the dialog box appears, users can then use the directory tree to change to another directory if they want.

When specifying the initial directory, include the full path name. For example,

```
C:\WINDOWS\SYSTEM
```

If no initial directory is specified, the directory that is current when the dialog box appears remains the current directory. The same is true if you specify a directory that does not exist.

Example

This code specifies C:\WINDOWS as the initial directory when the dialog box appears, displays the dialog box, and displays the name of the file the user selects with the dialog box in a label on the form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  OpenDialog1.InitialDir := 'C:\WINDOWS';
  if OpenDialog1.Execute then
    Label1.Caption := OpenDialog1.FileName;
end;
```

See also

Filter property

InitialValues property

Applies to
TReport component

Declaration

```
property InitialValues: TStrings;
```

The *InitialValues* property is a list of report variable strings the specified report uses to run. By specifying these initial values, your application can bypass the dialog boxes that prompt you for these values when the report runs.

Example
The following code adds two report variable values and runs the report.

```
   Report1.InitialValues.Add('@Report1=<35>');
   Report1.InitialValues.Add('@Report2=<test>');
   Report1.Run;
```

See also
SetVariable method, *SetVariableLines* method

InitWinCrt procedure **WinCrt**

Declaration

```
procedure InitWinCrt;
```

The *InitWinCrt* procedure creates a WinCRT window.

If you do not explicitly call *InitWinCrt*, it is automatically called when you use *Read*, *Readln*, *Write*, or *Writeln* on a file assigned to the CRT.

InitWinCrt uses the *WindowOrg*, *WindowSize*, and *ScreenSize* constants, and the *WindowTitle* variable to determine the characteristics of the CRT window.

See also
ScreenSize typed constant, *WindowOrg* typed constant, *WindowSize* typed constant, *WindowTitle* variable

InOutRes variable **System**

Declaration

```
var InOutRes: Integer;
```

The built-in I/O routines use the *InOutRes* variable to store the value that the next call to the *IOResult* standard function will return.

InOutRes is used by the built-in I/O functions.

InPlaceActive property

Applies to

TOLEContainer component

Declaration

```
property InPlaceActive: Boolean;
```

Run-time and read only. The *InPlaceActive* property specifies whether the OLE object in an OLE container is active in-place. If so, the value of *InPlaceActive* is *True*. If the object is deactivated, or activated in its own window (not in place), the value of *InPlaceActive* is *False*.

When an OLE object is active in-place, the OLE server application controls the editing of the OLE object from within the OLE container application. The OLE server might replace menu items and the status bar of the OLE container.

Example

The following code waits until an OLE object is activated in place before unlocking *Panel1*. Attach this code to the *OnActivate* event handler of *OLEContainer1*.

```
procedure TForm1.OleContainer1Activate(Sender: TObject, Activating:Boolean);
begin
  if OLEContainer1.InPlaceActive then
    Panel1.Locked := False;
end;
```

See also

Active property, *GroupIndex* property

Input variable System

Declaration

```
var Input: Text;
```

The *Input* variable is a read-only file associated with the operating system's standard input device, which is usually the keyboard.

In many of Delphi's standard file-handling routines, the file variable parameter can be omitted. Instead the routine operates on the *Input* or *Output* file variable. The following standard file-handling routines operate on the *Input* file when no file parameter is specified:

- *Eof*
- *Eoln*

- *Read*
- *Readln*
- *SeekEof*
- *SeekEoln*

Since Windows does not support text-oriented input and output, *Input* and *Output* files are unassigned by default in a Windows application. Any attempt to read or write to them will produce an I/O error.

If the application uses the *WinCrt* unit, *Input* and *Output* will refer to a scrollable text window.

See also

Output variable, *TextFile* type

InputBox function **Dialogs**

Declaration

```
function InputBox(const ACaption, APrompt, ADefault: string): string;
```

The *InputBox* function displays an input dialog box ready for the user to enter a string in its edit box. The *ACaption* parameter is the caption of the dialog box, the *APrompt* parameter is the text that prompts the user to enter input in the edit box, and the *ADefault* parameter is the string that appears in the edit box when the dialog box first appears.

If the user chooses the Cancel button, the default string is the value returned. If the user chooses the OK button, the string in the edit box is the value returned.

Use the *InputBox* function when it doesn't matter if the user chooses either the OK button or the Cancel button (or presses *Esc*) to exit the dialog box. When your application needs to know if the user chooses OK or Cancel (or presses *Esc*), use the *InputQuery* function.

Example

This example displays an input dialog box when the user clicks the button on the form. The input dialog box includes a prompt string and a default string. The string the user enters in the dialog box is stored in the *InputString* variable.

```
uses Dialogs;

procedure TForm1.Button1Click(Sender: TObject);
var
  InputString: string;
begin
  InputString:= InputBox('Input Box', 'Prompt', 'Default string');
end;
```

See also

MessageDlg function, *MessageDlgPos* function

InputQuery function

Dialogs

Declaration

```
function InputQuery(const ACaption, APrompt: string; var Value: string): Boolean;
```

The *InputQuery* function displays an input dialog box ready for the user to enter a string in its edit box. The *ACaption* parameter is the caption of the dialog box, the *APrompt* parameter is the text that prompts the user to enter input in the edit box, and the *Value* parameter is the string that appears in the edit box when the dialog box first appears. If the user enters a string in the edit box and chooses OK, the *Value* parameter changes to the new value.

The *InputQuery* function returns *True* if the user chooses OK, and *False* if the user chooses Cancel or presses the *Esc* key.

If your application doesn't need to know whether the user chooses OK or Cancel, use the *InputBox* function.

Example

This example uses a button and a label on the form. When the user clicks the button, a the input box displays. If the user chooses OK, the string that appears in the edit box of the dialog box displays as the caption of the label on the form. If the user chooses Cancel, the dialog box closes and the caption of the label remains unchanged.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NewString: string;
  ClickedOK: Boolean;
begin
  NewString := 'Default String';
  Label1.Caption := NewString;
  ClickedOK := InputQuery('Input Box', 'Prompt', NewString);
  if ClickedOK then                            { NewString contains new input string }
    Label1.Caption := 'The new string is ''' + NewString + '''';
end;
```

See also

MessageDlg function, *MessageDlgPos* function

Insert method

Applies to

TList, *TMenuItem*, *TStringList*, *TStrings* objects; *TOutline*, *TTable*, *TQuery* components

For list objects

Declaration

```
procedure Insert(Index: Integer; Item: Pointer);
```

The *Insert* method inserts an item into the list of items stored in the *List* property of a list object. Specify the item to insert as the value of the *Item* parameter. Specify the position in the list where you want the item inserted as the value of the *Index* parameter. The index is zero-based, so the first position in the list has an index value of 0.

If your application calls *Insert* when the list of items is sorted, an *EListError* exception is raised.

Example

The following code inserts *MyObject* into *MyList* at the position immediately following the position of *MyOtherObject*.

```
MyList.Insert(IndexOf(MyOtherObject)+1, MyObject);
```

See also

Add method, *Clear* method, *Count* property, *Delete* method, *First* method, *IndexOf* method, *Last* method

For string objects

Declaration

```
procedure Insert(Index: Integer; const S: string);
```

The *Insert* method inserts a string into the list of strings in a string or string list object. The string *S* is inserted into the position in the list indicated by the value of *Index*. The index is zero-based, so the first position in the list has an index value of 0.

Example

This example uses a list box and a button on a form. When the form appears, it contains five items. When the user clicks the button, another string is inserted at the top of the list of items:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 5 do
    ListBox1.Items.Add('Item ' + IntToStr(I));
end;

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

```
    ListBox1.Items.Insert(0, 'Inserted here');
  end;
```

See also

Add method, *AddStrings* method, *Clear* method, *Delete* method, *IndexOf* method, *InsertObject* method

For menu items

Declaration

```
procedure Insert(Index: Integer; Item: TMenuItem);
```

The *Insert* method inserts a menu item in a menu at the position indicated by the value of *Index*.

Example

This example inserts a new menu item after the first item in a menu named *FileMenu*:

```
  procedure TForm1.Button1Click(Sender: TObject);
  var
    NewItem: TMenuItem;
  begin
    NewItem := TMenuItem.Create(FileMenu);
    NewItem.Caption := 'Do this';
    FileMenu.Insert(1, NewItem);
  end;
```

See also

Add method, *Count* property, *Delete* method

For outlines

Applies to

TOutline component

Declaration

```
function Insert(Index: Longint; const Text: string): Longint;
```

Description

The *Insert* method inserts an outline item (*TOutlineNode* object) into an outline. The value of the *Index* and *Text* parameters are stored in the *Index* and *Text* properties of the inserted item. *Insert* returns the *Index* property value of the inserted item.

The inserted item appears in the outline position determined by the *Index* parameter. It is inserted at the same level as the item that previously resided at this position. Therefore, the inserted item and the original item are siblings and share the same parent.

The original item and all other outline items that appear after the inserted item are moved down one row and are reindexed with valid *Index* values. This happens automatically unless the *BeginUpdate* method was called.

Note To insert an item as the last top-level item in an outline, pass zero (0) in the *Index* parameter.

Example

The following code inserts an item as a sibling of the selected item.

```
begin
  Outline1.Insert(Outline1.SelectedItem, 'New item');
end;
```

See also

Add method, *AddChild* method, *AddChildObject* method, *InsertObject* method, *MoveTo* method

For tables and queries

Declaration

```
procedure Insert;
```

The *Insert* method puts the *dataset* into Insert state and opens a new, empty record at the current cursor location. When an application calls *Post*, the new record will be inserted in the dataset in a position based on its index, if defined. To discard the new record, use *Cancel*.

This method is valid only for datasets that return a live result set.

Note For indexed tables, the *Append* and *Insert* methods will both put the new record in the correct location in the table, based on the table's index. If no index is defined on the underlying table, then the record will maintain its position—*Append* will add the record to the end of the table, and *Insert* will insert it at the current cursor position. In either case, posting a new record may cause rows displayed in a data grid to change as the dataset follows the new row to its indexed position and then fetches data to fill the data grid around it.

Example

```
with Table1 do
  begin
{ Move to the end ot the component }
  Last;
  Insert;
  FieldByName('CustNo').AsString := '9999';
  { Fill in other fields here }
  if { you are sure you want to do this} then Post
  else { if you changed your mind } Cancel;
  end.
```

Insert procedure **System**

Declaration

```
procedure Insert(Source: string; var S: string; Index: Integer);
```

The *Insert* procedure merges a substring into a string beginning at a specified point.

Source is a string-type expression. *S* is a string-type variable of any length. *Index* is an integer-type expression.

Insert merges *Source* into *S* at the position *S*[*index*]. If the resulting string is longer than 255 characters, it is truncated after the 255th character.

Example

```
 var
  S: string;
begin
  S := 'Honest Lincoln';
  Insert('Abe ', S, 8);                    { 'Honest Abe Lincoln' }
end;
```

See also

Concat function, *Copy* function, *Delete* procedure, *Length* function, *Pos* function

InsertComponent method

Applies to

All components

Declaration

```
procedure InsertComponent(AComponent: TComponent);
```

The *InsertComponent* method makes the component own the component passed in the *AComponent* parameter. The component is added to the end of the *Components* array property. The inserted component must have no name (no specified *Name* property value), or the name must be unique among all others in the *Components* list.

When the owning component is destroyed, *AComponent* is destroyed also.

Example

The following code inserts *NewButton* into the *Components* array of *Form1*.

```
Form1.InsertComponent(NewButton);
```

See also

RemoveComponent method

InsertControl method

Applies to

All controls

Declaration

```
procedure InsertControl(AControl: TControl);
```

The *InsertControl* method inserts a control within the *Controls* property of a windowed control, making the inserted control a child, and the containing control the parent. The inserted control is the value of the *AControl* parameter.

Example

This example uses a button placed next to a group box. When the user clicks the button, the group box becomes the parent of the button, so the button moves inside the group box.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  RemoveControl(Button1);
  GroupBox1.InsertControl(Button1);
end;
```

Note that it was necessary to remove the button from the *Controls* property of the form before the button actually moves into the group box.

This code accomplishes the same thing:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Parent := GroupBox1;
end;
```

See also

Parent property, *RemoveControl* method

InsertObject method

Applies to

TStringList, *TStrings* objects, *TOutline* component

For string and string list objects

Declaration

```
procedure InsertObject(Index: Integer; const S: string; AObject: TObject);
```

The *InsertObject* method inserts a string into the list of strings and an object into the list of objects in a string or string list object. Specify the string you want to insert as the value of the *S* parameter, and the object you want to insert as the value of the *AObject* parameter. The *Index* parameter identifies the position of the string and object in their respective string and object lists. Because the index is zero-based, the first position in each list has an *Index* value of 0.

If your application calls *InsertObject* when the list of items is sorted, an *EListError* exception is raised.

Example

The following code inserts the components of *Form1* into the first position of the *Lines* list of *Memo1*.

```
var
  I: Integer;
begin
  for I := 0 to Form1.ComponentCount-1 do
  begin
    with Form1.Components[i] as TComponent do
    Memo1.lines.InsertObject(0, Name, Self);
  end;
end;
```

See also

AddObject method, *IndexOfObject* method, *Insert* method, *Objects* property, *Strings* property

For outlines

Applies to

TOutline component

Declaration

```
function InsertObject(Index: Longint; const Text: string; const Data: Pointer): Longint;
```

Description

The *InsertObject* method inserts an outline item (*TOutlineNode* object) containing data into an outline. The value of the *Index* and *Text* parameters are stored in the *Index* and *Text* properties of the inserted item. The *Data* parameter specifies the *Data* property value of the new item. *Insert* returns the *Index* property value of the inserted item.

The inserted item appears in the outline position determined by the *Index* parameter. It will be inserted at the same level as the item that previously resided at this position. Therefore, the inserted item and the original item will be siblings and share the same parent. The original item and all other outline items that appear after the inserted item are moved down one row and are reindexed with valid *Index* values. This is done automatically unless the *BeginUpdate* method was called.

Note To insert an item as the last top-level item in an outline, pass zero (0) in the *Index* parameter.

Example

The following code creates a bitmap object named *Bitmap1* and inserts an outline item containing *Bitmap1* into the first level of *Outline1*.

```
var
  Bitmap1: TBitmap;
begin
  Bitmap1 := TBitmap.Create;
  Outline1.InsertObject(1, 'New item', Bitmap1);
end;
```

See Also

Add method, *AddChild* method, *AddChildObject* method, *Insert* method, *MoveTo* method

InsertOLEObjectDlg function **Toctrl**

Declaration

```
function InsertOleObjectDlg(Form: TForm; HelpContext: THelpContext;
  var PInitInfo: Pointer): Boolean;
```

InsertOLEObjectDlg displays the Insert Object dialog box. Use this function to allow the user to specify the OLE object initialization information by using the Insert Object dialog box.

InsertOLEObjectDlg returns *True* if the user specifies an OLE object and chooses OK from the Insert Object dialog box. *InsertOLEObjectDlg* returns *False* if the user doesn't specify an OLE object or chooses Cancel in the dialog box.

These are the parameters of *InsertOLEObjectDlg*:

Field	Description
Form	The form that owns the Insert Object dialog box.
HelpContext	A Help context identification number that is used if the user chooses Help from within the Insert Object dialog box. If you pass 0 for *HelpContext*, no Help button appears in the Insert Object dialog box. Pass a number other than 0 if you want to provide context-sensitive online Help.
PInitInfo	If *InsertOLEObject* returns *True*, *InsertOLEObjectDlg* modifies the *PInitInfo* pointer parameter to point to OLE initialization information. Initialize the OLE object by assigning this pointer to the *PInitInfo* property. When your application is finished with the *PInitInfo* pointer, it should be released with *ReleaseOLEInitInfo*.

Example

The following code displays the Insert Object dialog box. If the user specifies an object and chooses OK, *OLEContainer1* is initialized. After initialization, the OLE information is released.

```
var
  Info: Pointer;
begin
  if InsertOLEObjectDlg(Form1, 0, Info) then
  begin
    OLEContainer1.PInitInfo := Info;
    ReleaseOLEInitInfo(Info);
  end;
end;
```

See also

LinksDlg function, *PasteSpecialDlg* function

InsertRecord method

Applies to

TTable, *TQuery* components

Declaration

```
procedure InsertRecord(const Values: array of const);
```

The *InsertRecord* method inserts a new record into the *dataset* using the field values passed in the *Values* parameter. The assignment of the elements of *Values* to fields in the record is sequential; the first element is assigned to the first field, the second to the second, etc. The number of field values passed in *Values* may be fewer than the number of actual fields in the record; any remaining fields are left unassigned and are NULL. The type of each element of *Values* must be compatible with the type of the field in that the field must be able to perform the assignment using *AsString*, *AsInteger*, etc., according the type of the *Values* element.

This method is valid only for datasets that return a live result set.

Note For indexed tables, the *AppendRecord* and *InsertRecord* methods will both put the new record in the correct location in the table, based on the table's index. If no index is defined on the underlying table, then the record will maintain its position—*AppendRecord* will add the record to the end of the table, and *InsertRecord* will insert it at the current cursor position. In either case, posting a new record in a data grid may cause all the rows before and after the new record to change as the dataset follows the new row to its indexed position and then fetches data to fill the grid around it.

Example

```
Table1.InsertRecord([9998, 'Lesh', 'Phil']);
```

See also

TField component

Int function

System

Declaration

```
function Int(X: Real): Real;
```

The *Int* function returns the integer part of the argument.

X is a real-type expression. The result is the integer part of *X*; that is, *X* rounded toward zero.

Example

```
var
  R: Real;
begin
  R := Int(123.456);    { 123.0 }
  R := Int(-123.456);   { -123.0 }
end;
```

See also

Frac function, *Round* function, *Trunc* function

IntegralHeight property

Applies to

TDBListBox, *TDirectoryListBox*, *TFileListBox*, *TListBox* component

Declaration

```
property IntegralHeight: Boolean;
```

The *IntegralHeight* property controls the way the list box represents itself on the form. If *IntegralHeight* is *True*, the list box shows only entries that fit completely in the vertical space, and the bottom of the list box moves up to the bottom of the last completely drawn item in the list. If *IntegralHeight* is *False*, the bottom of the list box is at the location determined by its *ItemHeight* property, and the bottom item visible in the list might not be complete.

If the list box has a *Style* property value of *lbOwerDrawVariable*, setting the *IntegralHeight* property to *True* has no effect.

If the *Style* property value of the list box is *lsOwnerDrawFixed*, the height of the list box at design time is always an increment of the *ItemHeight* value.

Example

This example uses a list box on a form. To try it, enter as many strings in the *Items* property as you like using the Object Inspector. When the application runs, the list box displays only entries that fit completely in the vertical space, and the bottom of the list box moves up to the bottom of the last string in the list box if the form is less than 300 pixels in height:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if Height < 300 then
    ListBox1.IntegralHeight := True
  else
    ListBox1.IntegralHeight := False;
end;
```

See also

ItemHeight property, *Items* property

Interval property

Applies to

TTimer component

Declaration

```
property Interval: Word;
```

The *Interval* property determines in milliseconds the amount of time that passes before the timer component initiates another *OnTimer event.*

You can specify any value between 0 and 65,535 as the interval value, but the timer component won't call an *OnTimer* event if the value is 0. The default value is 1000 (one second).

Example

The code in this *OnTimer* event handler moves a ball, the shape component (*TShape*) slowly across a form.

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Timer1.Interval := 100;
  Shape1.Shape := stCircle;
  Shape1.Left := Shape1.Left + 1;
end;
```

IntToHex function **SysUtils**

Declaration

```
function IntToHex(Value: Longint; Digits: Integer): string;
```

The *IntToHex* function converts a number into a string containing the number's hexadecimal (base 16) representation with a specific number of digits.

Example

When the user clicks the button on the form, this code converts the number entered in Edit1 to a hexadecimal string. The string displays in Edit2.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit2.Text := IntToHex(StrToInt(Edit1.Text), 6);
end;
```

See also

IntToStr function

IntToStr function — SysUtils

Declaration

```
function IntToStr(Value: Longint): string;
```

The *IntToStr* function converts an integer into a string containing the decimal representation of that number.

Example

This example uses a button and an edit box on a form. The code assigns a value to the *Value* variable and displays the string representation of the *Value* variable in the edit box.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Value: Integer;
begin
  Value := 1234;
  Edit1.Text := IntToStr(Value);
end;
```

See also

IntToHex function, *StrToInt* function

Invalidate method

Applies to

All controls; *TForm* component

Declaration

```
procedure Invalidate;
```

The *Invalidate* method forces a control to repaint as soon as possible.

Example

The following code invalidates *Form1*.

```
Form1.Invalidate;
```

See also

Refresh method, *Update* method

IOResult function **System**

Declaration

```
function IOResult: Integer;
```

The *IOResult* function returns the status of the last I/O operation performed.

I/O-checking must be off—{**$I–**}—to trap I/O errors using *IOResult*.

If an I/O error occurs and I/O-checking is off, all subsequent I/O operations are ignored until a call is made to *IOResult*. Calling *IOResult* clears the internal error flag.

An alternative way to handle I/O errors is to use exception handling. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

Example

```
var
  F: file of Byte;
begin
  if OpenDialog1.Execute then begin
    AssignFile(F, OpenDialog1.FileName);
    {$I-}
    Reset(F);
    {$I+}
    if IOResult = 0 then
      MessageDlg('File size in bytes: ' + IntToStr(FileSize(F)),
        mtInformation, [mbOk], 0);
    else
      MessageDlg('File access error', mtWarning, [mbOk], 0);
  end;
end;
```

IsIndexField property

Applies to

TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
property IsIndexField: Boolean;
```

Run-time and read only. The *IsIndexField* property specifies whether or not a field is indexed. If *True*, a field is indexed.

IsMasked property

Applies to

TDBEdit, *TMaskEdit* components

Declaration

```
property IsMasked: Boolean;
```

The *IsMasked* property determines if a mask exists (the *EditMask* property has a value) for the data displayed in the database edit box or mask edit box. If *IsMasked* is *True*, a mask exists. If *IsMasked* is *False*, no mask exists.

Example

This example tests the masked edit box to determine if it has an edit mask. If it doesn't an edit mask is assigned. The edit mask is one for dates in the MM/DD/YY format:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if not MaskEdit1.IsMasked then
    MaskEdit1.EditMask := '!99/99/00;1;_';
end;
```

See also

EditMask property

IsNull property

Applies to

TParam object; *TBCDField*, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

For TParam objects

Declaration

```
property IsNull: Boolean;
```

IsNull is a read only property that returns *True* if the parameter has no data assigned to it. This should only occur if an application has called:

- *Assign* with another parameter that has no data assigned.
- *AssignField* with a *TField* whose data is null.
- The *Clear* method.

Example

```
{ Set the CustNo parameter to 999 if it is null }
with Params.ParamByName('CustNo') do
  if IsNull then AsInteger := 999;
```

For fields

Declaration

```
property IsNull: Boolean;
```

Run-time and read only. *IsNull* returns *True* if the value of the field is NULL.

See also

Required property

IsSQLBased property

Applies to

TDataBase component

Declaration

```
property IsSQLBased: Boolean;
```

Run-time and read only. *IsSQLBased* is *True* if the *TDatabase* component uses any driver other than 'STANDARD'. If you are accessing a dBASE or Paradox database or ASCII file, *IsSQLBased* will be *False*.

IsValidChar method

Applies to

TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
function IsValidChar(InputChar: Char): Boolean; virtual;
```

IsValidChar is used by data-aware controls to determine if a particular character entered in the field is valid for the field. *TIntegerField*, *TSmallintField* and *TWordField* allow '+', '-'

and ‘0’ to ‘9’. *TBCDField* and *TFloatField* also allow ‘E’, ‘e’, and the *DecimalSeparator* character. All other fields accept all characters.

See also

DecimalSeparator variable

IsValidIdent function

SysUtils

Declaration

```
function IsValidIdent(const Ident: string): Boolean;
```

IsValidIdent returns *True* if the given string is a valid identifier. An identifier is defined as a character from the set ['A'..'Z', 'a'..'z', '_'] followed by zero or more characters from the set ['A'..'Z', 'a'..'z', '0..'9', '_'].

Note All component names must be valid Object Pascal identifiers.

IsVisible property

Applies to

TOutlineNode object

Declaration

```
property IsVisible: Boolean;
```

Run-time and read only. The *IsVisible* property indicates whether the outline item is visible within the *TOutline* component. An item is visible if it is on level 1 or if all its parents are expanded.

Example

The following code expands the branch of the selected outline item if it isn’t visible.

```
with Outline1.Items[Outline1.SelectedItem] do
  if not IsVisible then FullExpand;
```

See also

Expanded property, *Level* property

ItemAtPos method

Applies to

TDBListBox, *TDirectoryListBox*, *TFileListBox*, *TListBox*, *TTabSet* components

For list boxes

Applies to
TDBListBox, *TDirectoryListBox*, *TFileListBox*, *TListBox* components

Declaration

```
function ItemAtPos(Pos: TPoint; Existing: Boolean): Integer;
```

The *ItemAtPos* method returns the index of the list box indicated by the coordinates of a point on the control. The *Pos* parameter is the point in the control in window coordinates.

If *Pos* is beyond the last item in the list box, the value of the *Existing* variable determines the returned value. If you set *Existing* to *True*, *ItemAtPos* returns -1, indicating that no item exists at that point. If you set *Existing* to *False*, *ItemAtPos* returns the position of the last item in the list box.

ItemAtPos is useful for detecting if an item exists at a particular point in the control.

Example
This example uses a list box, and edit box, and a button on a form. When the user clicks the button, the index value of the item in the list box which contains the point specified in the code appears in the edit box:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Value: Integer;
  APoint: TPoint;
begin
  APoint.X := 30;
  APoint.Y := 50;
  Value := ListBox1.ItemAtPos(APoint, False);
  Edit1.Text := IntToStr(Value);
end;
```

See also
ItemIndex property, *Items* property

For tab sets

Applies to
TTabSet component

Declaration

```
function ItemAtPos(Pos: TPoint): Integer;
```

The *ItemAtPos* method returns the index of the tab indicated by the coordinates of a point on the control. The *Pos* parameter is the point in the control in window coordinates. If the returned index is 0, the tab is the first tab in the tab set, if the index is 1, the tab is the second tab, and so on.

ItemAtPos is useful for determining which tab is located at a particular position in the tab set control.

Example

The following code selects the tab that is at client coordinates (100, 10) in *TabSet1*.

```
TabSet1.TabIndex := TabSet1.ItemAtPos(Point(100, 10);
```

See also

TabIndex property, *Tabs* property

ItemCount property

Applies to

TOutline component

Declaration

```
property ItemCount: Longint;
```

Run-time and read only. The *ItemCount* property specifies the total number of items in an outline.

Example

The following code turns off automatic reindexing before inserting a new item into the index if the index includes more than 100 items. Otherwise, automatic reindexing remains active.

```
Outline1.SetUpdateState(Outline1.ItemCount > 100)
Outline1.Insert(1, 'NewItem', MyData);
Outline1.EndUpdate
```

See also

Items property

ItemHeight property

Applies to

TComboBox, *TDBComboBox*, *TDBListBox*, *TDirectoryListBox*, *TFileListBox*, *TListBox*, *TOutline* components

Declaration

```
property ItemHeight: Integer;
```

For list boxes, the *ItemHeight* property is the height of an item in the list box in pixels when the list box's *Style* property is *lsOwnerDrawFixed*. If the *Style* property is *lsStandard* or *lsOwnerDrawVariable*, the value of *ItemHeight* is ignored. You can control the height of an item in a fixed owner-draw list box by changing the height of *ItemHeight*.

For combo boxes, the *ItemHeight* property is the height of an item in the combo box list in pixels when the combo box's *Style* property is *csOwnerDrawFixed*. If the *Style* property is any other setting, the value of *ItemHeight* is ignored. You can control the height of an item in a fixed owner-draw combo box by changing the height of *ItemHeight*.

For outlines, the *ItemHeight* property is the height of an item in the outline in pixels when the outline's *Style* property is *osOwnerDraw*. If the *Style* property is *osStandard*, the value of *ItemHeight* is ignored. You can control the height of an item in an owner-draw outline by changing the height of *ItemHeight*.

Example

This example uses a list box and a button on a form. Enter as many strings in the list box as you like using the property editor of the *Items* property in the Object Inspector. When the user clicks the button on the form, the amount of vertical space allotted to each item in the list box changes.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Style := lbOwnerDrawFixed;
  ListBox1.ItemHeight := 30;
end;
```

See also

Items property, *IntegralHeight* property, *OnDrawItem* event

ItemIndex property

Applies to

TComboBox, *TDBComboBox*, *TDBRadioGroup*, *TDirectoryListBox*, *TDriveComboBox*, *TFileListBox*, *TFilterComboBox*, *TListBox*, *TRadioGroup* components

Declaration

```
property ItemIndex: Integer;
```

Run-time only. The value of the *ItemIndex* property is the ordinal number of the selected item in the control's item list. If no item is selected, the value is -1, which is the default value unless *MultiSelect* is *True*. To select an item at run time, set the value of *ItemIndex* to the index of the item in the list you want selected, with 0 being the first item in the list.

For list boxes and combo boxes, if the value of the *MultiSelect* property is *True* and the user selects more than one item in the list box or combo box, the *ItemIndex* value is the index of the selected item that has focus. If *MultiSelect* is *True*, *ItemIndex* defaults to 0.

Example

This example uses a drive combo box on a form. When the user selects a drive in the combo box, the index value of the selected item appears in the caption of the label:

```
procedure TForm1.DriveComboBox1Change(Sender: TObject);
begin
  Label1.Caption := 'Index value ' + IntToStr(DriveComboBox1.ItemIndex);
end;
```

See also

Items property

ItemRect method

Applies to

TDBListBox, *TDirectoryListBox*, *TDrawGrid*, *TFileListBox*, *TListBox*, *TStringGrid*, *TTabSet* components

Declaration

```
function ItemRect(Item: Integer): TRect;
```

The *ItemRect* method returns the rectangle that surrounds the item specified in the *Item* parameter.

Example

This example uses a list box and four labels on a form. When the application runs, three strings are added to the list box. When the user selects one of the strings in the list box, the coordinates of the rectangle taken up by the selected string appear in the four labels:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  with ListBox1 do
  begin
    Items.Add('Hello');
    Items.Add('New');
    Items.Add('World');
  end;
end;

procedure TForm1.ListBox1Click(Sender: TObject);
var
  ListBoxItem: TRect;
begin
  ListBoxItem := ListBox1.ItemRect(ListBox1.ItemIndex);
  Label1.Caption := 'Left ' + IntToStr(ListBoxItem.Left);
  Label2.Caption := 'Top ' + IntToStr(ListBoxItem.Top);
  Label3.Caption := 'Right ' + IntToStr(ListBoxItem.Right);
  Label4.Caption := 'Bottom ' + IntToStr(ListBoxItem.Bottom);
end;
```

See also
TRect type

Items property

Applies to
TFieldDefs, *TIndexDefs*, *TList*, *TParams* objects; *TComboBox*, *TDBComboBox*, *TDBListBox*, *TDBRadioGroup*, *TDirectoryListBox*, *TDriveComboBox*, *TFileListBox*, *TFilterComboBox*, *TListBox*, *TMainMenu*, *TMenuItem*, *TOutline*, *TPopupMenu*, *TRadioGroup* components

For list boxes, combo boxes, and radio group boxes

Declaration

```
property Items: TStrings;
```

The *Items* property contains the strings that appear in the list box or combo box, or as radio buttons in a radio group box. Because *Items* is an object of type *TStrings*, you can add, delete, insert, and move items using the *Add*, *Delete*, *Insert*, *Exchange*, and *Move* methods of the *TStrings* object.

The *ItemIndex* property determines which item is selected, if any.

To determine if a particular item in the list of strings that makes up the *Items* property for a list box or combo box is selected, use the *Selected* property.

Example
This example uses an edit box, a list box, and a button on a form. When the user clicks the button, the text in the edit box is added to the list box:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add(Edit1.Text);
end;
```

See also
Add method, *Delete*, method, *Exchange* method, *Insert* method, *ItemIndex* property, *Move* method, *Selected* property

For menu items, main menus, and pop-up menus

Declaration

```
property Items[Index: Integer]: TMenuItem;
```

Read-only property.

For menu items, the *Items* array property provides access to a subitem of a menu item (*TMenuItem*) by its position in the list of subitems. The value of *Index* is the position of the subitem within the *Items* array. For example, if an application has a File drop-down menu that contains the menu items New, Open, and Save, in that order, FileMenu.Items[2] refers to the Save command. For menu items, *Items* is run-time only property.

For main menus, the *Items* property provides access to a menu item on the main menu bar, and is available at both design time and run time.

For pop-up menus, the *Items* property provides access to a menu item on the pop-up menu, and is available at both design time and run time.

Example

The following code disables all the subitems of *MenuItem1*.

```
var
  I: Integer;
begin
  for I := 0 to MenuItem1.ItemCount-1 do
    MenuItem1.Items[I].Enabled := False;
end;
```

See also

Count property

For outlines

Declaration

```
property Items[Index: Longint]: TOutlineNode;
```

Run-time and read only. For outlines, the *Items* array property provides access to a outline node by its row position. The value of the *Index* parameter corresponds to the *Index* property and represents the position of the item within the *Items* array. For example, if an outline has three items with *Index* property values of 1, 2, and 3 and *Text* property values of 'Orange', 'Apple', and 'Banana', respectively, *Items[2]* refers to the 'Apple' item.

Example

The following code collapses the selected item of *Outline1*.

```
Outline1.Items[Outline1.SelectedItem].Expanded := False;
```

See also

SelectedItem property

For list objects

Declaration

```
property Items[Index: Integer]: Pointer;
```

Run-time only. The *Items* array property lets you access a specific pointer kept in the *List* property of a list object. Using the *Index* parameter of *Items* you can access a list item by its position in the list.

Example

This example creates a list object and inserts two records into it. The value of the record fields are written on the form:

```
procedure TForm1.FormActivate(Sender: TObject);
type
  PMyList = ^AList;
  AList = record
    I: Integer;
    C: Char;
  end;

var
  MyList: TList;
  ARecord: PMyList;
  B: Byte;
  Y: Word;

begin
  MyList := TList.Create;
  New(ARecord);
  ARecord^.I := 100;
  ARecord^.C := 'Z';
  MyList.Add(ARecord);         {Add integer 100 and character Z to list}
  New(ARecord);
  ARecord^.I := 200;
  ARecord^.C := 'X';
  MyList.Add(ARecord);         {Add integer 200 and character X to list}
  Y := 10;                     {Variable used in TextOut function}

{Go through the list until the end is reached}
  for B := 0 to (MyList.Count - 1) do
  begin
    Y := Y + 30;               {Increment Y Value}
    ARecord := MyList.Items[B];
    Canvas.TextOut(10, Y, IntToStr(ARecord^.I)); {Display I}
    Y := Y + 30;                {Increment Y Value again}
    Canvas.TextOut(10, Y, ARecord^.C);  {Display C}
  end;
  MyList.Free;
end;
```

See also

Add method, *Expand* method, *First* method, *IndexOf* method, *Last* method, *Remove* method

For TIndexDefs objects

Declaration

```
property Items[Index: Integer]: TIndexDef;
```

Run-time and read only. *Items* holds the *TIndexDef* objects that describe each index of the dataset. The number of entries is given by the *Count* property; there will be one entry for each index of the dataset.

For TParams objects

Declaration

```
property Items[Index: Word]: TParam;
```

Read and run-time only. The *Items* array property holds the parameters (*TParam* objects). Use this property when you want to work with the entire set. While you can use *Items* to reference a particular parameter by its index, the *ParamByName* method is recommended to avoid depending on the order of the parameters.

Example

```
{ Assign 99999 to any integer parameter which does not have a value }
for I := 0 to Params.Count - 1 do
  if (Params.Items[I].IsNull) and (Params.Items[I].DataType = ftInteger) then
{ Items is the default property, so you can omit its name }
    Params[I].AsInteger := 99999;
```

For TFieldDefs objects

Declaration

```
property Items[Index: Integer]: TFieldDef;
```

Items is an array of pointers to the *TFieldDef* objects that describe each field in the dataset. There is one pointer for each component in the dataset.

See also
Count property

ItemSeparator property

Applies to
TOutline component

Declaration

```
property ItemSeparator: string;
```

The *ItemSeparator* property determines the separator string used between the outline item *Text* values in the *FullPath* property of the *TOutlineNode* object. The default value of *ItemSeparator* is '\'.

For example, if the top-level outline item has a *Text* value of 'Animals' and a child item with the *Text* value of 'Dogs', the *FullPath* property of the 'Dogs' item would have the value 'Animals\Dogs' by default. If the string '->' were assigned to the *ItemSeparator* property, the *FullPath* property of the 'Dogs' item would be 'Animals->Dogs'.

Example

The following code changes the item separator to ':'.

```
Outline1.ItemSeparator := ':';
```

See also

FullPath property, *Text* property, *TOutlineNode* object

KeepConnection property

Applies to

TDataBase component

Declaration

```
property KeepConnection: Boolean;
```

The *KeepConnection* property specifies whether an application remains connected to a database server even when no tables are open. If an application needs to open and close several tables in a single database, it will be more efficient to set *KeepConnection* to *True*. That way, the application will remain connected to the database even when it does not have any tables open. It can then open and close tables repeatedly without incurring the overhead of connecting to the database each time. If *KeepConnection* is *False*, the database must repeat the login process to the server each time the *Connected* property is set to *True*.

The *TSession* component has an application-wide *KeepConnections* property that determines the initial state of the *KeepConnection* property for temporary (automatically-created) *TDatabase* components.

Example

```
Database1.KeepConnection := False;
```

KeepConnections property

Applies to

TSession component

Declaration

```
property KeepConnections: Boolean;
```

Run-time only. *KeepConnections* specifies whether virtual *TDatabase* components will maintain database connections even if no tables in the database are open. Databases that have an explicit *TDatabase* component will use *TDatabase*'s *KeepConnection* property instead to determine if connections are persistent.

If *KeepConnections* is *True* (the default), the application will maintain database connections until the application exits or calls the *DropConnections* method. If *KeepConnections* is *False*, then the application will disconnect from the database when all datasets connected to tables in the database are closed.

Note *KeepConnections* has no effect on connections to databases for which an application has an explicit *TDatabase* component.

Example

```
   Session.KeepConnections := False;
```

See also

Session variable

KeyExclusive property

Applies to

TTable component

Declaration

```
property KeyExclusive: Boolean;
```

The *KeyExclusive* property indicates whether range and search functions will exclude the matching records specified by the functions. *KeyExclusive* is *False* by default.

For the *SetRangeStart* and *SetRangeEnd* methods, *KeyExclusive* determines whether the filtered range excludes the range boundaries. The default is *False*, which means rows will be in the filtered range if they are greater than or equal to the start range specified and less than or equal to the end range specified. If *KeyExclusive* is *True*, the methods will filter strictly greater than and less than the specified values.

For the *GotoNearest* and *FindNearest* methods, *KeyExclusive* indicates whether a search will position the cursor on or after the record being searched for. If *KeyExclusive* is *False*, then *GoToNearest* and *FindNearest* will move the cursor to the record that matches the specified values, if found. If *True*, then the methods will go the record immediately following the matching record, if found.

Example

```
{ Limit the range from 1351 to 1356, excluding both 1351 and 1356 }
with Table1 do
  begin
{ Set the beginning key }
  EditRangeStart;
  IndexFields[0].AsString := '1351';
{ Exclude 1351 itself }
  KeyExclusive := True;
{ Set the ending key }
  EditRangeEnd;
  IndexFields[0].AsString := '1356';
{ Exclude 1356 itself }
  KeyExclusive := True;
{ Tell the dataset to establish the range }
  ApplyRange;
  end;
```

See also
ApplyRange method, *EditRangeStart* method, *EditRangeEnd* method, *KeyFieldCount* property

KeyFieldCount property

Applies to
TTable component

Declaration

```
property KeyFieldCount: Integer;
```

KeyFieldCount specifies the number of key fields to use with search functions (*GotoKey*, *FindKey*, *EditKey*, and so on) if you don't want to search on all the fields in the key.

See also
GotoKey method, *GotoNearest* method, *EditKey* method, *FindKey* method, *FindNearest* method, *SetKey* method

KeyPressed function WinCrt

Declaration

```
function KeyPressed: Boolean;
```

The *KeyPressed* function returns *True* if a key has been pressed on the keyboard

The key can be read using the *ReadKey* function.

Example

```
uses WinCrt;

begin
  repeat
    Write('Xx');
  until KeyPressed;
end;
```

See also

ReadKey function

KeyPreview property

Applies to

TForm component

Declaration

```
property KeyPreview: Boolean;
```

When the *KeyPreview* property is *True*, most key events (*OnKeyDown* event, *OnKeyUp* event, and *OnKeyPress* event) go to the form first, regardless of which control is selected on the form. This allows your application to determine how to process key events. After going to the form, key events are then passed to the control selected on the form. When *KeyPreview* is *False*, the key events go directly to the controls. The default value is *False*.

The exceptions are the navigation keys, such as *Tab*, *BackTab*, the arrow keys, and so on. If the selected control processes such keys, you can use *KeyPreview* to intercept them; otherwise, you can't.

If *KeyPreview* is *False*, all key events go to the selected control.

Example

This example changes a form's color to aqua when the user presses a key, even when a control on the form has the focus. When the user releases the key, the form returns to its original color.

```
var
  FormColor: TColor;

procedure TForm1.FormCreate(Sender: TObject);
begin
  KeyPreview := True;
end;

procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  FormColor := Form1.Color;
  Form1.Color := clAqua;
end;

procedure TForm1.FormKeyUp(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  Form1.Color := FormColor;
end;
```

KeyViolCount property

Applies to

TBatchMove component

Declaration

```
property KeyViolCount: Longint;
```

Run-time and read only. *KeyViolCount* reports the number of records which could not be replaced, added, or deleted from *Destination* because of an integrity (key) violations. If *AbortOnKeyViol* is *True*, *KeyViolCount* will never be greater than one, since the first violation will cause the move to terminate.

Example

```
  with BatchMove1 do
    begin
      Execute;
      if KeyViolCount <> 0 then { something went wrong };
    end;
```

KeyViolTableName property

Applies to

TBatchMove component

Declaration

```
property KeyViolTableName: TFileName;
```

KeyViolTableName, if specified, creates a local (Paradox) table containing all records from the source table that caused an integrity violation (such as a key violation) as a result of the batch operation.

If *AbortOnKeyViol* is *True*, then there will be at most one record in this table since the operation will be aborted with that first record. *KeyViolCount* will have the number of records placed in the new table.

Example

```
BatchMove1.KeyViolTableName := 'KeyViol';
```

See also

Destination property

Kind property

Applies to

TBitBtn, *TScrollBar* components

The *Kind* property specifies the style or type of component.

For bitmap buttons

Declaration

```
property Kind: TBitBtnKind;
```

The *Kind* property determines the kind of bitmap button. These are the possible values and their meanings:

Value	Meaning
bkCustom	You indicate which bitmap you want the bitmap button to have by setting the value of the *Glyph* property to the bitmap of your choice. Like push buttons, you can either select a *ModalResult* for the button, or you can supply the code to respond to an *OnClick* event.
bkOK	A green check mark and the text "OK" appears on the button face. The button becomes the default button (the *Default* property is automatically set to *True*). When the user chooses the button, the dialog box closes. The resulting *ModalResult* value of the bitmap button is *mrOK*.
bkCancel	A red X and the text "Cancel" appears on the button face. The button becomes the Cancel button (the *Cancel* property is automatically set to *True*). When the user chooses the button, the dialog box closes. The resulting *ModalResult* value of the bitmap button is *mrCancel*.
bkYes	A green check mark and the text "Yes" appears on the button face. The button becomes the default button (the *Default* property is automatically set to *True*). When the user chooses the button, any changes the user made in the dialog box are accepted and the dialog box closes. The resulting *ModalResult* value of the bitmap button is *mrYes.*
bkNo	A red no symbol and the text "No" appears on the button face. The button becomes the Cancel button (the *Cancel* property is automatically set to *True*). When the user chooses the button, any changes the user made in the dialog box are canceled and the dialog box closes. The resulting *ModalResult* value of the bitmap button is *mrNo.*
bkHelp	A cyan question mark and the text "Help" appears on the button face. When the user chooses the button, a Help screen in the application's Help file appears. The Help file that appears is the file specified as the value of the application's *HelpFile* property. The value of the *HelpContext* property of the button specifies which Help screen in the Help file appears.

Value	Meaning
bkClose	A door with a green exit sign over it (use your imagination) and the text “Close” appear on the button face. When the user chooses the button, the form closes. The *Default* property of the button is *True*.
bkAbort	A red X and the text “Abort” appears on the button face. The *Cancel* property of the button is automatically set to *True*.
bkRetry	A green circular arrow and the text “Retry” appear on the button face.
bkIgnore	A green man walking away and the text “Ignore” appears on the button face. Use it to allow the user to continue after an error has occurred.
bkAll	A double green check mark and the text “Yes to All” appears on the button face. The *Default* property of the button is automatically set to *True*.

Example

This example uses three bitmap buttons on a form. When the application runs, the *Kind* property for each bitmap button is set, and the *BitBtn1* button (the OK button) becomes the default button.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  BitBtn1.Kind := bkOK;
  BitBtn2.Kind := bkCancel;
  BitBtn3.Kind := bkHelp;
end;
```

See also

Cancel property, *Default* property, *ModalResult* property

For scroll bars

Declaration

```
property Kind: TScrollBarKind ;
```

The *Kind* property determines if a scroll bar is horizontal or vertical. These are the possible values:

Value	Meaning
sbHorizontal	Scroll bar is horizontal
sbVertical	Scroll bar is vertical

For scroll bars of type *TControlScrollBar* (form and scroll box scroll bars accessed through the *HorzScrollBar* and *VertScrollBar* properties), *Kind* is a read- and run-time-only property.

Example

This example uses a radio group box and a scroll bar on a form. When the user selects one of the radio buttons, the scroll bar changes orientation accordingly.

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
  RadioGroup1.Items.Add('Vertical');
  RadioGroup1.Items.Add('Horizontal');
  RadioGroup1.ItemIndex := 2;
end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  if RadioGroup1.Items[RadioGroup1.ItemIndex] = 'Vertical' then
    ScrollBar1.Kind := sbVertical;
  if RadioGroup1.Items[RadioGroup1.ItemIndex] = 'Horizontal' then
    ScrollBar1.Kind := sbHorizontal;
end;
```

LargeChange property

Applies to
TScrollBar component

Declaration

```
property LargeChange: TScrollBarInc;
```

The *LargeChange* property determines how far the scroll box moves when the user clicks the scroll bar on either side of the scroll box or presses *PgUp* or *PgDn*. The default value is 1 position.

For example, if the *LargeChange* property setting is 1000, each time the user clicks the scroll bar, the scroll box moves 1000 positions. How big the change from one position to another depends on the difference between the *Max* property value and the *Min* property value. If *Max* is 3000 and *Min* is 0, the user needs to click the scroll bar three times to move the scroll box from one end of the scroll bar to the other.

Example
This code determines that when the user clicks the scroll bar on either side of the scroll box, the scroll box moves 100 positions on the scroll bar:

```
ScrollBar1.LargeChange := 100;
```

See also
Max property, *Min* property, *Position* property, *SmallChange* property

Last method

Applies to
TList object; *TQuery*, *TStoredProc*, *TTable* components

For list objects

Declaration

```
function Last: Pointer;
```

The *Last* method returns a pointer that points to the last item referenced in the *List* property of a list object.

Example

This example inserts two records into a list object and displays the contents of the last record in the list on the form:

```
procedure TForm1.FormActivate(Sender: TObject);
type
  PMyList = ^AList;
  AList = record
    I: Integer;
    C: Char;
end;

var
  MyList: TList;
  ARecord: PMyList;

begin
  MyList := TList.Create;
  New(ARecord);
  ARecord^.I := 100;
  ARecord^.C := 'Z';
  MyList.Add(ARecord); {Add integer 100 and character Z to list}
  New(ARecord);
  ARecord^.I := 200;
  ARecord^.C := 'X';
  MyList.Add(ARecord); {Add integer 200 and character X to list}
  ARecord := MyList.Last;
  Canvas.TextOut(10, 10, IntToStr(ARecord^.I)); {Display I}
  Canvas.TextOut(10, 40, ARecord^.C);  {Display C}
  MyList.Free;
end;
```

See also

Capacity property, *First* method, *IndexOf* method, *Items* property

For tables, queries, and stored procedures

Declaration

```
procedure Last;
```

The *Last* method moves the cursor to the last record in the active range of records of the dataset. The active range of records is affected by the filter established with *SetRangeEnd*.

If the dataset is in Insert or Edit state, *Last* will perform an implicit *Post* of any pending data.

Example

```
Table1.Last;
```

See also

First method, *MoveBy* method, *Next* method, *Prior* method, *SetRangeEnd method*

Layout property

Applies to

TBitBtn, *TSpeedButton* components

Declaration

```
property Layout: TButtonLayout;
```

The *Layout* property determines where the image appears on the bitmap button or a speed button. These are the possible values:

Value	Meaning
blGlyphLeft	The image appears near the left side of the button.
blGlyphRight	The image appears near the right side of the button.
blGlyphTop	The image appears near the top of the button.
blGlyphBottom	The image appears near the bottom of the button.

Example

This example uses a bitmap button on a form that has a bitmap specified as the value of its *Glyph* property. When the user clicks the bitmap button, the bitmap randomly changes its position on the button:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  Randomize;
  case Random(4) of
    0: BitBtn1.Layout := blGlyphLeft;
    1: BitBtn1.Layout := blGlyphRight;
    2: BitBtn1.Layout := blGlyphTop;
    3: BitBtn1.Layout := blGlyphBottom;
  end;
end;
```

See also

Margin property, *Spacing* property

Left property

Applies to

All controls; *TFindDialog*, *TReplaceDialog* components

Declaration

```
property Left: Integer;
```

The *Left* property determines the horizontal coordinate of the left edge of a component relative to the form in pixels. For forms, the value of the *Left* property is relative to the screen in pixels. The default value is -1.

The *Left* property for the Find and Replace dialog boxes is available at run-time only.

Example

The following example moves the button 10 pixels to the right each time a user clicks it:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Left := Button1.Left + 10;
end;
```

See also

SetBounds method, *Top* property

LeftCol property

Applies to

TDrawGrid, *TStringGrid* components

Declaration

```
property LeftCol: Longint;
```

Run-time only. The *LeftCol* property determines which column in the grid appears at the far left side of the grid.

If you have one or more nonscrolling columns in the grid, they remain at the far left, regardless of the value of the *LeftCol* property. In this case, the column you specify as the far left column is the first column to the immediate right of the nonscrolling columns.

Example

This line of code positions the fourth column of a string grid at the left edge of the grid:

```
StringGrid1.LeftCol := 3;
```

See also

FixedCols property, *TopRow* property

Length function **System**

Declaration

```
function Length(S: string): Integer;
```

The *Length* function returns the dynamic length of the string *S*.

Example

```
  var
    S: string;
  begin
    S := 'The Black Knight';
    Canvas.TextOut(10, 10, 'String Length = ' + IntToStr(Length(S)));
  end;
```

See also

Concat function, *Copy function*, *Delete procedure*, *Insert procedure*, *Pos function*

Length property

Applies to

TMediaPlayer component

Declaration

```
property Length: Longint;
```

Run-time and read only. The *Length* property specifies the length of the medium in the open multimedia device. *Length* is specified using the current time format, which is specified by the *TimeFormat* property.

Example

The following code sets *Wait* to *False* if the *Length* of the media is over 10,000. If *TimeFormat* is *tfMilliseconds*, *Wait* is set to *False* if the media is over 10 seconds long.

```
  MediaPlayer1.Wait := (MediaPlayer1.Lenth > 10000);
```

See also

Position property, *Start* property, *TrackLength* property

Level property

Applies to

TOutlineNode object

Declaration

```
property Level: Word;
```

Run-time and read only. The *Level* property indicates the level of indentation of an item within the *TOutline* component. The value of *Level* is 1 for items on the top level. The value of *Level* is 2 for their children, and so on.

Example

The following code tests to determine if the fifth outline item is on the same level as the selected outline item.

```
if Outline1.Items[5].Level = Outline1.Items[Outline1.SelectedItem].Level then
  { The selected item is on the same level as the fifth item };
```

See also

ChangeLevelBy method, *TopItem* property

Lines property

Applies to

TDBMemo, *TDDEClientItem*, *TDDEServerItem*, *TMemo*, *TOutline* components

Lines property for memos

Declaration

```
property Lines: TStrings;
```

The *Lines* property contains the text lines in a memo component.

For a database memo control, the *Lines* property is a run-time property only.

Example

This example uses a button and a memo control on a form. When the user clicks the button, the contents of the system's AUTOEXEC.BAT file is loaded into the memo, and the sixth line of the file is written across the top of the form.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.Lines.LoadFromFile('C:\AUTOEXEC.BAT');
  Writeln('The 6th line of AUTOEXEC.BAT is: ', Memo1.Lines[5]);
end;
```

See also

GetTextBuf method, *SetTextBuf* method, *Text* property

Lines property for outlines

Declaration

```
property Lines: TStrings;
```

The *Lines* property contains the *Text* property values of the individual items in an outline.

If the *Lines* property is of an outline component, each line becomes an outline item in a *TOutlineNode* object. Leading tabs and spaces are converted into levels of the outline. Text without any leading tabs or spaces become level 1 items. For example, to create a level 2 item, lead the text of the item with one tab or space.

The *Lines* property of outlines is primarily useful for stuffing an outline with items at design time. While you can access the individual items with the *Lines* property at run time, it is much quicker to access an item with the *Items* property.

Example

The following two lines of code each produce the same result. In the first line, *Lines* is used to access the *Strings* value of the third outline node. In the second line, *Items* is used to access the *Text* value of the third outline node. Note that the index used with *Items* is one more than the index used with Lines.Strings.

```
   Edit1.text := Outline1.Lines.Strings[2];
   Edit2.Text := Outline1.Items[3].Text;
```

Lines property for DDE items

Declaration

```
property Lines: TStrings;
```

The *Lines* property contains the text data to exchange in a DDE conversation. For *TDDEClientItem* components, *Lines* specifies the text that is updated by the DDE server application. For *TDDEServerItem* components, *Lines* specifies the text that is sent to any DDE clients when the value of *Lines* changes or when a client requests to be updated. When *Lines* is changed, an *OnChange* event occurs.

Lines corresponds to the *Text* property. Whenever the value of *Lines* or *Text* is changed, the other is updated so that the first line of *Lines* is always equal to *Text*. Use *Lines* to contain text values longer than 255 characters (which is the limit of the *Text* property). For shorter strings, use the *Text* property.

If the *Lines* property is of a *TDDEClientItem* component, you can also send the text in *Lines* directly to the DDE server by poking data with the *PokeDataLines* method.

If the *Lines* property is of a *TDDEServerItem* component, the DDE client can change *Lines* by poking data. The poked data replaces the contents of *Lines* and an *OnChange* event occurs.

Example

The following code assigns the value to the *Lines* property of *DDEClientItem1* to the *Lines* of Memo1. This code is executed in the *OnChange* event handler of *DDEClientItem1*, so whenever the client is updated, the new data from the server is displayed.

```
procedure TForm1.DdeClientItem1Change(Sender: TObject);
begin
  Memo1.Lines := DDEClientItem1.Lines
end;
```

LineTo method

Applies to

TCanvas object

Declaration

```
procedure LineTo(X, Y: Integer);
```

The *LineTo* method draws a line on the canvas from the current drawing position (specified by the *PenPos* property) to the point specified by *X* and *Y* and sets the pen position to (*X*, *Y*).

Example

The following code draws a line from the upper left corner of a form to the point clicked with the mouse.

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.MoveTo(0, 0);
  Canvas.LineTo(X, Y);
end;
```

See also

MoveTo method

LinksDlg procedure **Toctrl**

Declaration

```
procedure LinksDlg(Form: TForm; HelpContext: THelpContext);
```

LinksDlg displays the Links dialog box. Use the Links dialog box to view and edit the current OLE links in your application. These are the parameters of *LinksDlg*:

Field	Description
Form	The form that owns the Links dialog box.
HelpContext	A Help context ID number to be used if the user chooses Help from within the Links dialog box. If you pass 0 for *HelpContext*, no Help button appears in the Links dialog box. Pass a number other than 0 if you want to provide context-sensitive online Help.

Example

The following code activates the Links dialog with a context-sensitive Help ID number of 1000.

```
LinksDlg(Form1, 1000);
```

See also

InsertOLEObjectDlg function, *LinksDlgEnabled* function, *PasteSpecialDlg* function

LinksDlgEnabled function

Toctrl

Declaration

```
function LinksDlgEnabled(Form: TForm): Boolean;
```

LinksDlgEnabled determines if the Links dialog box is enabled. If so, *LinksDlgEnabled* returns *True* and *LinksDlg* can be successfully called. If not, *LinksDlgEnabled* returns *False* and nothing happens if you call *LinksDlg*.

The *Form* parameter specifies the form that owns the Links dialog box.

Example

The following code activates the Links dialog box if it is enabled.

```
if LinksDlgEnabled(Form1) then LinksDlg(Form1, 0);
```

List property

Applies to

TList object

Declaration

```
property List: PPointerList;
```

Run-time and read only. The *List* property stores a list of pointers that reference objects of any type. The declaration of *PPointerList* is

```
PPointerList = ^TPointerList;
```

The declaration of *TPointerList* is

```
TPointerList = array[0..MaxListSize-1] of Pointer;
```

The elements of the *TPointerList* array each point to an item of the list.

Example

The following code creates *List1* and *Object1,* then adds *Object1* to *List1*. If the first item in the *List* property *List1* differs from the first item of the *Items* property of *List1* (which shouldn't happen), a message is displayed.

```
var
  List1: TList;
  Object1: TObject;
begin
  List1 := TList.Create;
  Object1 := TObject.Create;
  List1.Add(Object1);
  if List1.List^[0]<>List1.Items[0] then
    MessageDlg('Something is wrong here', mtInformation, [mbOK], 0);
  List1.Free;
  Object1.Free;
end;
```

See also

FileEditStyle property, *TStrings* object

Ln function — System

Declaration

```
function Ln(X: Real): Real;
```

The *Ln* function returns the natural logarithm (Ln(e) = n) of the real-type expression X.

Example

```
var
   e : real;
   S : string;
begin
   e := Exp(1.0);
   Str(ln(e):3:2, S);
   S := 'ln(e) = ' + S;
   Canvas.TextOut(10, 10, S);
end;
```

See also

Exp function

Lo function

System

Declaration

```
function Lo(X): Byte;
```

The *Lo* function returns the low-order *Byte* of the argument *X* as an unsigned value. *X* is an expression of type *Integer* or *Word*.

Example

```
var B: Byte;
begin
  B := Lo($1234);   { $34 }
end;
```

See also

Hi function, *Swap function*

LoadFromFile method

Applies to

TBitmap, *TGraphic*, *TIcon*, *TMetafile*, *TPicture*, *TStringList*, *TStrings* objects; *TBlobField*, *TGraphicField*, *TMemoField*, *TOLEContainer*, *TOutline* components

For graphics objects and outlines

Declaration

```
procedure LoadFromFile(const FileName: string);
```

The *LoadFromFile* method reads the file specified in *FileName* and loads the data into the object or component. The graphics objects load graphics, the OLE container loads an OLE object, and the outline and string objects load text.

Example

This example uses a bitmap button on a form. When the application runs and the form is created, a bitmap is placed on the bitmap button:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  BitBtn1.Glyph.LoadFromFile('TARTAN.BMP');
end;
```

See also

SaveToFile method, *Strings* property

For blob, graphic, and memo fields

Declaration

```
procedure LoadFromFile(const FileName: string);
```

The *LoadFromFile* method reads a file with the name passed in *FileName* and loads the contents in *TBlobField, TMemoField,* or *TGraphicField*.

Note For *TMemoField* and *TGraphicField*, the file should have been created by the *SaveToFile* or *SaveToStream* method.

Example

```
{ Load a blob field with the contents of autoexec.bat }
BlobField1.LoadFromFile('c:\autoexec.bat');
```

LoadFromStream method

Applies to

TBlobField, TGraphicField, TMemoField components

Declaration

```
procedure LoadFromStream(Stream: TStream);
```

The *LoadFromStream* method reads *Stream* and stores the contents in *TBlobField, TMemoField* or *TGraphicField*.

Note For a *TMemoField* or *TGraphicField*, the file should have been created by the *SaveToFile* or *SaveToStream* method.

Example

```
{ Load a blob field from an existing STream1 }
BlobField1.LoadFromStream(Stream1);
```

See also

LoadFromFile method, *SaveToStream* method

LoadMemo method

Applies to

TDBMemo component

Declaration

```
procedure LoadMemo;
```

The *LoadMemo* method loads a text BLOB into the database memo control. If the value of the *AutoDisplay* property is *False*, the text of a memo is not automatically loaded. If *AutoDisplay* is *False*, you can control when the text is loaded at run time by calling *LoadMemo* when you want the text to appear in the control.

Example

This example uses a database memo that is connected to a BLOB text field in the dataset. It also contains a button. When the user clicks the button, the BLOB loads into the memo.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DBMemo1.AutoDisplay := False;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  DBMemo1.LoadMemo;
end;
```

See also

LoadPicture method

LoadPicture method

Applies to

TDBImage component

Declaration

```
procedure LoadPicture;
```

The *LoadPicture* method loads the image specified as the value of the *Picture* property into the database image control.

If the value of the *AutoDisplay* property is *False*, the image of a database image control is not automatically loaded. If *AutoDisplay* is *False*, you can control when the image is loaded at run time by calling *LoadPicture* when you want the image to appear in the control.

Example

The following code loads the picture into *DBImage1*.

```
DBImage1.LoadPicture;
```

See also

LoadMemo method

LoadStr function — SysUtils

Declaration

```
function LoadStr(Ident: Word): string;
```

LoadStr loads the string resource given by *Ident* from the application's executable file. If the string resource does not exist, an empty string is returned.

Moving display strings into string resources makes your application easier to localize without rewriting your program.

Local property

Applies to

TQuery component

Declaration

```
property Local: Boolean;
```

Run-time and read only. The *Local* property specifies if the table referenced by the *TQuery* is a local dBASE or Paradox table or an SQL server table. If *Local* is *True*, then the table is a dBASE or Paradox table. If *Local* is *False*, the table is a SQL table.

For remote SQL tables, some operations (such as record counts) may take longer than for local tables, owing to network constraints.

Example

```
{ If the table is local, allow the data-aware controls to display the changes }
DataSource1.Enabled := Query1.Local;
```

Locale property

Applies to

TDataBase, *TTable*, *TQuery*, *TSession*, *TStoredProc* components

For tables, queries, and stored procedures

Declaration

```
property Locale: TLocale;
```

Run-time and read only. The *Locale* property identifies the language driver used with the *dataset for use with direct calls to* the Borland Database Engine API.

Under most circumstances you should not need to use this property, unless your application requires some functionality not encapsulated in the VCL.

For sessions

Declaration

```
property Locale: TLocale;
```

Run-time and read only. The *Locale* property identifies the language driver used with the *TSession* component. It enables you to reference the language driver when making direct calls to the Borland Database Engine API. Under most circumstances you should not need to use this property, unless your application requires some functionality not encapsulated in the VCL.

See also

Session variable

For database components

Declaration

```
property Locale: TLocale;
```

Run-time and read only. The *Locale* property identifies the language driver used with the *TDatabase* component. It allows you to make direct calls to the Borland Database Engine API. Under most circumstances you should not need to use this property, unless your application requires some functionality not encapsulated in the VCL.

Locked property

Applies to

TPanel component

Declaration

```
property Locked: Boolean;
```

The *Locked* property determines whether a panel is replaced by an in-place active OLE object. If *Locked* is *False*, the OLE server can replace the panel. If *Locked* is *True* and the panel is aligned to one of the edges of the form (its *Align* property is *alTop*, *alBottom*, *alLeft*, or *alRight*), then the panel remains when an OLE object in a *TOLEContainer* component is activated in place.

Use *Locked* to prevent status bars and the like from being replaced.

Example

The following code sets *Locked* to *True* for a panel named *StatusBar*.

```
  StatusBar.Locked := True;
```

See also
InPlaceActive property

LoginPrompt property

Applies to
TDataBase component

Declaration

```
property LoginPrompt: Boolean;
```

The *LoginPrompt* property is used to control how security is handled for SQL databases.

If *True*, (the default), the standard Delphi Login dialog box will be opened when the application attempts to establish a database connection. The user must then enter a proper user name and password to connect to a database on the server.

If *False*, then an application will look for login parameters in the *Params* property of the *TDatabase* component. These are the USERNAME and PASSWORD parameters. For example,

```
USERNAME = SYSDBA
PASSWORD = masterkey
```

This is generally not recommended since it compromises server security.

Example

```
{ Do not display the login prompt }
Database1.LoginPrompt := False;
```

See also
OnLogin event

LongRec — SysUtils

Declaration

```
LongRec = record
  Lo, Hi: Word;
end;
```

LongRec declares a utility record that stores the high and low order bytes of the specified variable as type *Word*.

LongRec is useful in handling double-word length variables.

See also
Hi function, *Lo* function

LookupDisplay property

Applies to

TDBLookupCombo, *TDBLookupList* components

Declaration

```
property LookupDisplay : string;
```

The *LookupDisplay* property determines which field in the lookup table displays in the database lookup combo box or database lookup list box. Before you specify a *LookupDisplay* field, link the two datasets using the *LookupField* property.

You can choose to display multiple fields from the lookup dataset. Each field appears in a separate column. To specify more than one field to display, separate each field name with a semicolon. For example, this line of code displays three columns in the drop-down list of a database lookup combo box. Column 1 is the name of the company, column 2 is the city where the company is located, and column 3 is the country.

```
DBLookupCombo1.LookupDisplay := 'Company;City;Country';
```

You can choose to include titles for the field columns and you can choose to have lines between the rows and columns using the *Options* property.

Example

The following code specifies that the 'Company' field is displayed in *DBLookupCombo1*.

```
DBLookupCombo1.LookupDisplay := 'Company';
```

See also

LookupField property, *Options* property

LookupField property

Applies to

TDBLookupCombo, *TDBLookupList* components

Declaration

```
property LookupField: string;
```

The *LookupField* property links the dataset the database lookup combo box or database lookup list box uses to "look up" data to the primary dataset you are working with.

Although the name of the field specified as the *LookupField* does not have to be the same as the name of the field specified as the *DataField*, the two fields must contain the same values. For example, the *LookupField* value can be *CustomerNumber* and the *DataField* value can be *CustNo*, as along as both fields use the same number to identify a particular customer. When you specify a *LookupField*, the current value of that field appears in the control, if the *Active* property of both datasets is *True*.

After you specify a *LookupField* field, you can choose which field you prefer to display in the control with the *LookupDisplay* property.

Example

The following code designates that *DBLookupCombo1* looks up data in the 'CustomerNumber' field.

```
DBLookupCombo1.LookupField := 'CustomerNumber';
```

See also

DataSource property, *LookupSource* property

LookupSource property

Applies to

TDBLookupCombo, *TDBLookupList* components

Declaration

```
property LookupSource: TDataSource;
```

The *LookupSource* of a database lookup combo box or lookup list box is the data source component (*TDataSource*) that identifies the dataset you want the control to use to "look up" the information you want displayed in the control.

Example

The following code specifies that *DataSource1* is the lookup source for *DBLookupCombo1*.

```
DBLookupCombo1.LookupSource := DataSource1;
```

See also

LookupDisplay property, *LookupField* property

Low function System

Declaration

```
function Low(X);
```

The *Low* function returns the lowest value in the range of the argument.

Result type is *X*, or the index type of *X* where *X* is either a type identifier or a variable reference.

Type	Low returns
Ordinal type	The lowest value in the range of the type
Array type	The lowest value within the range of the index type of the array

Type	Low returns
String type	Returns 0
Open array	Returns 0
String parameter	Returns 0

Example

```
function Sum( var X: array of Double): Double;
var
  I: Word;
  S: Real;
begin
  S := 0;              { Note that open array index range is always zero-based. }
  for I := 0 to High(X) do S := S + X[I];
  Sum := S;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  List1: array[0..3] of Double;
  List2: array[5..17] of Double;
  X: Word;
  S, TempStr: string;
begin
  for X := Low(List1) to High(List1) do
      List1[X] := X * 3.4;
  for X := Low(List2) to High(List2) do
      List2[X] := X * 0.0123;
  Str(Sum(List1):4:2, S);
  S := 'Sum of List1: ' + S + #13#10;
  S := S + 'Sum of List2: ';
  Str(Sum(List2):4:2, TempStr);
  S := S + TempStr;
  MessageDlg(S, mtInformation, [mbOk], 0);
end;
```

See also
High function

LowerCase function — SysUtils

Declaration

```
function LowerCase(const S: string): string;
```

The *LowerCase* function returns a string with the same text as the string passed in *S*, but with all letters converted to lowercase. The conversion affects only 7-bit ASCII characters between 'A' and 'Z'. To convert 8-bit international characters, use *AnsiLowerCase*.

Example

This example uses two edit boxes and a button on a form. When the user clicks the button, the text in the *Edit1* edit box displays in the *Edit2* edit box in lowercase letters.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit2.Text := LowerCase(Edit1.Text);
end;
```

See also

AnsiLowerCase function, *UpperCase* function

MainForm property

Applies to

TApplication component

Declaration

```
property MainForm: TForm;
```

Run-time and read only. The *MainForm* property identifies which form in the application is the main form, which is the form that is always created first. When the main form closes, the application terminates.

When you create a new project, *Form1* automatically becomes the value of the *MainForm* property. If you want to make another form become the main form, use the Forms page of the Options | Project Options dialog box.

See also

Application variable, *CreateForm* method, *Run* method

Mappings property

Applies to

TBatchMove component

Declaration

```
property Mappings: TStrings;
```

By default *TBatchMove* matches columns based on their position in the source and destination tables. That is, the first column in the source is matched with the first column in the destination, and so on.

To override the default column mappings, use the *Mappings* property. This is a list of column mappings (one per line) in one of two forms. To map the column ColName in the source table to the column of the same name in the destination table use:

```
ColName
```

Or, to map the column named SourceColName in the source table to the column named DestColName in the destination table:

```
DestColName = SourceColName
```

If source and destination column data types are not the same, *TBatchMove* will perform a "best fit". It will trim character data types, if necessary, and attempt to perform a limited amount of conversion if possible. For example, mapping a CHAR(10) column to a CHAR(5) column will result in trimming the last five characters from the source column.

As an example of conversion, if a source column of character data type is mapped to a destination of integer type, *TBatchMove* will convert a character value of '5' to the corresponding integer value. Values that cannot be converted will generate errors.

Fields in *Destination* which have no entry in *Mappings* will be set to NULL.

Example

```
var Maps: TStringList;
...
with Maps do
  begin
  Clear;
{ Map the CustomerNum field to CustNo }
  Add('CustNo=CustomerNum');
  end;
MatchMove1.Mappings := Maps;
```

See also
Source property

Margin property

Applies to
TBitBtn, *TControlScrollBar*, *TSpeedButton* components

For bitmap buttons and speed buttons

Declaration

```
property Margin: Integer;
```

The *Margin* property determines the number of pixels between the edge of the image (specified in the *Glyph* property) and the edge of the button. The edges that the margin separates depends on the layout of the image and text (specified in the *Layout* property). For example, if *Layout* is *blGlyphLeft*, the margin appears between the left edge of the image and the left edge of the button. If *Margin* is 3, three pixels separates the image and the button edges. If *Margin* is 0, no distance in pixels separates the image and the button edges.

If *Margin* is -1 (which it is by default), then the image and text (specified in the *Caption* property) are centered. The number of pixels between the image and button edge is equal to the number of pixels between the opposite edge of the button and the text.

Example

This example uses a moderately large bitmap button on a form. When the application runs, a bitmap (or glyph) is loaded on to the button, the bitmap appears on the right side of the button, and bitmap is placed 30 pixels from the right edge of the bitmap button.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
with BitBtn1 do
  begin
   Glyph.LoadFromFile('C:WINDOWS\CARS.BMP');
   Layout := blGlyphRight;
   Margin := 30;
  end;
end;
```

See also

Caption property, *Glyph* property, *Layout* property

For form and scroll box scroll bars

Applies to

TControlScrollBar component

Declaration

```
procedure Margin: Word;
```

The *Margin* property value is the minimum number of pixels you want controls on a form or in a scroll box to be from the edge of the form or scroll box. This number is automatically added to the *Range* value to ensure that the user has a scroll bar whenever the distance from a control and the edge of the form or scroll box becomes less than the *Margin* value.

The default value is 0.

Example

This example uses a button and a label on a form. Place the label near the left side of the form, and place the button somewhere near the middle of the form. When the user runs the application, a horizontal scroll bar does not appear, because no control on the form is close enough to the right edge. Each time the user clicks the button, the button moves 25 pixels to the right, and the calculated *Range* value is reported in the caption of the label. Repeatedly clicking the button eventually moves the button close enough to the edge of the form (within the *Margin* amount) so that a horizontal scroll bar appears:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  with HorzScrollBar do
  begin
    Margin:= 25;
    Increment := 10;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Left := Button1.Left + 25;
  Label1.Caption := IntToStr(HorzScrollBar.Range);
end;
```

See also
Align property, *HorzScrollBar* property, *Range* property, *VertScrollBar* property

Mask property

Applies to
TFileListBox, *TFilterComboBox* components

Declaration

```
property Mask: string
```

For filter combo boxes

Declaration

```
property Mask: string
```

Run-time and read only. The *Mask* property value is the string selected as the filter in the filter combo box.

Example
This example uses a filter combo box and a label on a form. When the user selects a filter in the filter combo box, the selected mask appears in the caption of the label:

```
procedure TForm1.FilterComboBox1Change(Sender: TObject);
begin
   Label1.Caption := 'The selected mask is ' + FilterComboBox1.Mask;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  FilterComboBox1.Filter := 'All files (*.*)|*.*| Pascal files (*.pas)|*.pas';
end;
```

See also
Filter property

For file list boxes

```
property Mask: string
```

The *Mask* property determines which files are displayed in the file list box. A file mask or file filter is a file name that usually includes wildcard characters (*.PAS, for example). Only files that match the mask are displayed in list box. The file mask *.* displays all files, which is the default value.

You can specify multiple file masks. Separate the file mask specifications with semicolons. For example, *.PAS; *.EXE.

Example
This example uses a file list box on a form. When the application runs, the list box displays only files with a .PAS file extension:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FileListBox1.Mask := '*.PAS';
end;
```

See also
Filter property

MasterFields property

Applies to
TTable component

Declaration

```
property MasterFields: string;
```

Use the *MasterFields* property to specify the column(s) to link a detail table with a master table that is specified by the *MasterSource* property. *MasterFields* is a string consisting of one or more column names that join the two tables. Separate multiple column names with semicolons. Each time the current record in the master table changes, the new values in those fields are used to select corresponding records from the detail table for display. At design time, use the Field Link Designer to set this property.

Example
Suppose you have a master table named *Customer* that contains a *CustNo* field, and you also have a detail table named *Orders* that also has a *CustNo* field. To display only those records in *Orders* that have the same *CustNo* value as the current record in *Customer*, write this code:

```
   Customer.MasterFields := 'CustNo';
```

If you want to display only the records in the detail table that match more than one field value in the master table, specify each field and separate them with a semicolon.

```
   Customer.MasterFields := 'CustNo;SaleDate';
```

See also

IndexName property

MasterSource property

Applies to

TTable component

Declaration

```
property MasterSource: TDataSource;
```

When linking a detail table to a master table, use the *MasterSource* property to specify the *TDataSource* from which the *TTable* will get data for the master table.

Example

```
   Table2.MasterSource := DataSource1;
```

See also

IndexName property, *MasterFields property*

Max property

Applies to

TScrollBar component

Declaration

```
property Max: Integer;
```

The *Max* property along with the *Min* property determines the number of possible positions the scroll box can have on the scroll bar. The *LargeChange* and *SmallChange* properties use the number of positions to determine how far to move the scroll box when the user uses the scroll bar.

For example, if *Max* is 30000 and *Min* is 0, the scroll box can assume 30,000 positions on a horizontal scroll bar. If the *LargeChange* property setting is 10000 and the scroll box position is at the far left of the scroll bar (*Position* is 0), the user can click the scroll bar three times to the right of the scroll box before the scroll box is moved all the way to the right of the scroll bar (30000/10000 = 3).

If you want to change the *Min*, *Max*, and *Position* values all at once at run time, call the *SetParams* method.

Example

This code changes the maximum position of the scroll bar from 100, the default value, to 30000:

```
ScrollBar1.Max := 30000;
```

See also

LargeChange property, *Min* property, *Position* property, *SetParams* method, *SmallChange* property

MaxAvail function System

Declaration

```
function MaxAvail: Longint;
```

The *MaxAvail* function returns the size of the largest contiguous free block in the heap.

MaxAvail returns the larger of:

- The largest free blocks within the heap manager's sub-allocation space
- The Windows global heap

The value corresponds to the size of the largest dynamic variable that can be allocated at that time.

To find the total amount of free memory in the heap, call *MemAvail*.

Example

```
uses Dialogs;

type
  FriendRec = record
    Name: string[30];
    Age: Byte;
  end;
var
  P: Pointer;
begin
  if MaxAvail < SizeOf(FriendRec) then
    MessageDlg('Not enough memory', mtWarning, [mtOk], 0)
  else
  begin
    { Allocate memory on heap }
    GetMem(P, SizeOf(FriendRec));
    { ... }
  end;
end;
```

See also
MemAvail function

MaxFontSize property

Applies to
TFontDialog component

Declaration

```
property MaxFontSize: Integer;
```

The *MaxFontSize* property determines the largest font size available in the Font dialog box. Use the *MaxFontSize* property when you want to limit the font sizes available to the user. To limit the font sizes available, the *Options* set property of the Font dialog box must also contain the value *fdLimitSize*. If *fdLimitSize* is *False*, setting the *MaxFontSize* property has no affect on number of fonts available in the Font dialog box.

The default value is 0, which means there is no maximum font size specified.

Example
This example uses a Font dialog box, a button, and a label on a form. When the user clicks the button, the Font dialog box appears. The font sizes available are within the range of 10 to 14. When the user chooses OK, the selected font is applied to the caption of the label.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  FontDialog1.Options := [fdLimitSize];
  FontDialog1.MaxFontSize := 14;
  FontDialog1.MinFontSize := 10;
  if FontDialog1.Execute then
    Label1.Font := FontDialog1.Font;
end;
```

See also
MinFontSize property

MaxLength property

Applies to
TComboBox, *TDBEdit*, *TDBLookupCombo*, *TDBMemo*, *TEdit*, *TMaskEdit*, *TMemo* components

Declaration

```
property MaxLength: Integer;
```

The *MaxLength* property specifies the maximum number of characters the user can enter in an edit box, memo, or combo box. The default setting for *MaxLength* is 0, which means that there is no limit on the number of characters the control can contain. Any other number limits the number of characters the control accepts.

Example

The following example sets the maximum number of characters for an edit box to 80:

```
Edit1.MaxLength := 80;
```

MaxPage property

Applies to

TPrintDialog component

Declaration

```
property MaxPage: Integer;
```

The *MaxPage* property determines the greatest page number the user can use when specifying pages to print. If the user specifies a number greater than the value in *MaxPage*, a warning message appears and the user must enter a valid number or close the dialog box. The default value is 0.

Note The user can specify pages numbers only if the *Options property* set includes the value *poPageNums*.

Example

This example uses a button and a Print dialog box on a form. When the user clicks the button, the code makes page four the highest page number the user can select in the Print dialog box and displays the dialog box:

```
procedure TForm1.Button1Click(Sender:TObject);
begin
  PrintDialog1.Options := [poPageNums];
  PrintDialog1.ToPage := 4;
  PrintDialog1.MaxPage := 4;
  if PrintDialog1.Execute then
    ...;
end;
```

See also

MinPage property

MaxRecords property

Applies to

TReport component

Declaration

```
property MaxRecords: Word;
```

The value of the *MaxRecords* property is the number of database records you want to use to create the report. For example, if you just want to see a sample report and your database contains 50,000 records, you can specify a *MaxRecords* value that limits the number of records in the report to a much smaller number.

Example

The following code sets the maximum number of records to be used by *Report1* to 3.

```
Report1.MaxRecords := 3;
```

MaxTabNameLen constant

Declaration

```
MaxTabNameLen = 20;
```

The *MaxTabNameLen* constant specifies that the longest string that can be displayed on a tab set control (*TTabSet*) is 20 characters.

MaxValue property

Applies to

TCurrencyField, *TFloatField*, *TIntegerField*, *TSmallintField*, *TWordField* component

Declaration

```
property MaxValue: Longint;
```

The *MaxValue* property limits the maximum value in the field. Assigning a value greater than *MaxValue* raises an exception.

Example

```
{ Limit a field to 1 to 10}
Field1.MaxValue := 10;
Field1.MinValue := 1;
```

See also

MinValue property

MDIChildCount property

Applies to

TForm component

Declaration

```
property MDIChildCount: Integer;
```

Run-time and read only. The value of the *MDIChildCount* property is the number of child windows open in an MDI application.

Example

The following code closes *Form1* if it has no MDI children open.

```
   if Form1.MDIChildCount = 0 then Form1.Close;
```

See also

ActiveMDIChild property, *FormStyle* property, *MDIChildren* property

MDIChildren property

Applies to

TForm component

Declaration

```
property MDIChildren[I: Integer]: TForm;
```

Run-time and read only. The *MDIChildren* property array provides access to a child window or form in an MDI application through an index value, *I*. The value of *I* is determined by the order in which the window was created. For example, the first MDI child window has an *I* value of 0.

Example

The following code closes all the MDI children of *Form1*.

```
   var
     I: Integer;
   begin
     with Form1 do
       for I := 0 to MDIChildCount-1 do
         MDIChildren[I].Close;
   end;
```

See also

FormStyle property, *MDIChildCount* property

MemAvail function **System**

Declaration

```
function MemAvail: Longint;
```

The *MemAvail* function returns the amount of all free memory in the heap.

Note that a contiguous block of storage the size of the returned value is unlikely to be available due to fragmentation of the heap. To find the largest free block, call *MaxAvail*.

Example

```
var
  S: string;
begin
  S := IntToStr(MemAvail) + ' bytes available' + #13#10;
  S := S + 'Largest free block is ' + IntToStr(MaxAvail) + ' bytes';
  Canvas.TextOut(10, 10, S);
end;
```

See also

MaxAvail function

Menu property

Applies to

TForm component

Declaration

```
property Menu: TMainMenu;
```

The *Menu* property designates the menu bar for the form.

Example

This code displays a new menu named *NewMenu* when the user clicks the button *ChangeMenu* button.

```
procedure TForm1.ChangeMenuClick(Sender: TObject);
begin
  Menu := NewMenu;
end;
```

Merge method

Applies to

TMainMenu component

Declaration

```
procedure Merge(Menu: TMainMenu);
```

The *Merge* method merges a main menu of one form with a main menu of another for non-MDI applications. For example, when your application uses the main menu of the first form as the main menu for the application, and your application displays a second form, you can call *Merge* to merge the main menu on the second form with the main menu of the application.

Specify the menu you want merged with this menu as the *Menu* parameter.

Depending on the value of the *GroupIndex* property of menu items on the main menu, the merged menu items can replace menu items on the menu bar, or add or insert menu items into the menu bar. See *GroupIndex* for information on how to do these things.

It you want merging and unmerging to occur automatically when another form is displayed, change the value of the *AutoMerge* property to *True*.

Example

This example uses two forms, each containing a main menu created with the Menu Designer. It also uses a button on *Form1*. When the user clicks the button, *Form2* appears and the main menu of *Form2* merges with that of *Form1*.

Before running this example, add *Unit2* to the **uses** clause of *Unit1*.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Show;
  MainMenu1.Merge(Form2.MainMenu1);
end;
```

See also

AutoMerge property, *Unmerge* method

MessageBox method

Applies to

TApplication component

Declaration

```
function MessageBox(Text, Caption: PChar; Flags: Word): Integer;
```

The *MessageBox* method is an encapsulation of the Windows API MessageBox function except that you don't need to supply a window handle.

The *MessageBox* method displays a generic dialog box that displays a message and one or more buttons. The value of the *Text* parameter is the message, which can be longer than 255 characters if necessary. Long messages are automatically wrapped in the message box. The value of the *Caption* property is the caption that appears in the title bar of the dialog box. Captions can be longer than 255 characters, but they don't wrap. A long caption results in a wide message box.

To see the possible values of the *Flags* parameter, see the *MessageBox* function in the Windows API Help file (WinAPI.HLP). The corresponding parameter on that Help screen is called *TextType*. The values determine the buttons that appear in the message box and the behavior of the message box. The values can be combined to obtain the effect your want.

The return value of the *MessageBox* method is 0, if there wasn't enough memory to create the message box, or one of these values:.

Value	Numeric value	Meaning
IDABORT	3	The user chose the Abort button
IDCANCEL	2	The user chose the Cancel button
IDIGNORE	5	The user chose the Ignore button
IDNO	7	The user chose the No button
IDOK	1	The user chose the OK button
IDRETRY	4	The user chose the Retry button
IDYES	6	The user chose the Yes button

Example

This example uses a button and a label on a form. When the user clicks the button, a message box appears. When the user responds to the message box, the button selected is reported in the caption of the label:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Button: Integer;
begin
  Button := Application.MessageBox('Welcome to Delphi!', 'Message Box', mb_OKCancel +
    mb_DefButton1);
 if Button = IDOK then
   Label1.Caption := 'You chose OK';
 if Button = IDCANCEL then
   Label1.Caption := 'You chose Cancel';
end;
```

See also

MessageDlg function, *MessageDlgPos* function, *ShowMessage* procedure, *ShowMessagePos* procedure

MessageDlg function — Dialogs

Declaration

```
function MessageDlg(const Msg: string; AType: TMsgDlgType; AButtons: TMsgDlgButtons;
  HelpCtx: Longint): Word;
```

The *MessageDlg* function displays a message dialog box for your application in the center of your screen. The message box displays the value of the *Msg* string constant.

The *AType* parameter determines the type of message box that appears. These are the possible values:

Value	Meaning
mtWarning	A message box containing a yellow exclamation point symbol.
mtError	A message box containing a red stop sign.
mtInformation	A message box containing a blue "i".
mtConfirmation	A message box containing a green question mark.
mtCustom	A message box containing no bitmap. The caption of the message box is the name of the application's executable file.

The *AButtons* parameter determines which buttons appear in the message box. *AButtons* is of type *TMsgDlgBtns*, which is a set, so you can include multiple buttons within the set. These are the values you can include in the set:

Value	Meaning
mbYes	A button with a green check mark and the text 'Yes' on its face
mbNo	A button with a red circle and slash mark through the circle and the text 'No' on its face
mbOK	A button with a green check mark and the text 'OK' on its face
mbCancel	A button with a red X and the text 'Cancel' on its face
mbHelp	A button with a cyan question mark and the text 'Help' on its face
mbAbort	A button with a red check mark and the text 'Abort' on its face
mbRetry	A button with two green circular arrows and the text 'Retry' on its face
mbIgnore	A button with a green man walking away and the text 'Ignore' on its face
mbAll	A button with a green double check marks and the text 'All' on its face

In addition to the individual set values, VCL defines three constants that are predefined sets that include common button combinations:

Value	Meaning
mbYesNoCancel	A set that puts the Yes, No, and Cancel buttons in the message box
mbOkCancel	A set that puts the OK and Cancel buttons in the message box
mbAbortRetryIgnore	A set that puts an Abort, Retry, and Ignore buttons in the message box

When using these constants, remember not to add the brackets [ ] to define the set. These constants are already predefined sets.

The *HelpCtx* parameter determines which Help screen is available for the message box. For more information about Help context values, see the *HelpContext* property.

The function returns the value of the button the user selected. These are the possible return values:

Return values		
mrNone	*mrAbort*	*mrYes*
mrOk	*mrRetry*	*mrNo*
mrCancel	*mrIgnore*	*mrAll*

The *MsgDlgButtonStyle* typed constant in the *Dialogs* unit is declared like this:

```
MsgDlgButtonStyle: TButtonStyle = bsAutoDetect;
```

This ensures that the style of the buttons matches the style used by the operating environment your application is running under. If you prefer to always use a particular style, change the value of the *MsgDlgButtonStyle*. See the *Style* property for bitmap buttons for the possible values and their meanings.

The *MsgDlgGlyphs* typed constant in the *Dialogs* unit is declared like this:

```
MsgDlgGlyphs: Boolean = True;
```

This declaration ensures that bitmaps (or glyphs) appear on the message dialog box buttons. If you prefer that the bitmaps are not present, change the value of *MsgDlgButtonStyle* to *False*.

Example

This example uses a button on a form. When the user clicks the button, a message box appears, asking if the user wants to exit the application. If the user chooses Yes, another dialog box appears informing the user the application is about to end. When user chooses OK, the application ends.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if MessageDlg('Welcome to my Object Pascal application.  Exit now?',
    mtInformation, [mbYes, mbNo], 0) = mrYes then
  begin
    MessageDlg('Exiting the Object Pascal application.', mtInformation,
      [mbOk], 0);
    Close;
  end;
end;
```

This example uses a button on a form. When the user clicks the button, a message box appears with a Yes, No, and Cancel button on it:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MessageDlg('Are you there?', mtConfirmation, mbYesNoCancel, 0);
end;
```

See also

Kind property, *MessageDlgPos* function, *ModalResult* property, *MessageBox* method, *ModalResult* property, *ShowMessage* procedure, *ShowMessagePos* procedure

MessageDlgPos function

Dialogs

Declaration

```
function MessageDlgPos(const Msg: string; AType: TMsgDlgType;
  AButtons: TMsgDlgButtons; HelpCtx: Longint; X, Y: Integer): Word;
```

The *MessageDlg* function displays a message dialog box in your application at the position you specify. The message box displays the value of the*Msg* string constant.

The *AType* parameter determines the type of message box that appears. These are the possible values:

Value	Meaning
mtWarning	A message box containing yellow exclamation point symbol.
mtError	A message box containing a red stop sign.
mtInformation	A message box containing a blue “i”.
mtConfirmation	A message box containing a green question mark.
mtCustom	A message box containing no bitmap. The caption of the message box is the name of the application’s executable file.

The *AButtons* parameter determines which buttons appear in the message box.*AButtons* is of type *TMsgDlgBtns*, which is a set, so you can include multiple buttons within the set. These are the values you can include in the set:

Value	Meaning
mbYes	A button with a green check mark and the text ‘Yes’ on its button face
mbNo	A button with a red circle and slash mark through the circle and the text ‘No’ on its button face
mbOK	A button with a green check mark and the text ‘OK’ on its button face
mbCancel	A button with a red X and the text ‘Cancel’ on its button face
mbHelp	A button with a cyan question mark and the text ‘Help’ on its button face
mbAbort	A button with a red check mark and the text ‘Abort’ on its face
mbRetry	A button with two green circular arrows and the text ‘Retry’ on its face
mbIgnore	A button with a green man walking away and the text ‘Ignore’ on its face
mbAll	A button with a green double check marks and the text ‘All’ on its face

In addition to the individual set values, VCL defines three constants that are predefined sets which include common button combinations:

Value	Meaning
mbYesNoCancel	A set that puts the Yes, No, and Cancel buttons in the message box
mbOkCancel	A set that puts the OK and Cancel buttons in the message box
mbAbortRetryIgnore	A set that puts an Abort, Retry, and Ignore buttons in the message box

When using these constants, remember not to add the brackets [ ] to define the set. These constants are already predefined sets.

The *HelpCtx* parameter determines which Help screen is available for the message box. For more information about Help context values, see the *HelpContext* property.

The *X* and *Y* integer parameters are the screen coordinates in pixels where the top left corner of the message box appears.

The function returns the value of the button the user selected. These are the possible return values:

Return values		
mrNone	*mrAbort*	*mrYes*
mrOk	*mrRetry*	*mrNo*
mrCancel	*mrIgnore*	*mrAll*

The *MsgDlgButtonStyle* constant in the *Dialogs* unit is declared like this:

```
MsgDlgButtonStyle: TButtonStyle = bsAutoDetect;
```

This ensures that the style of the buttons matches the style used by the operating environment your application is running under. If you prefer to always use a particular style, change the value of the *MsgDlgButtonStyle*. See the *Style* property for bitmap buttons for the possible values and their meanings.

The *MsgDlgGlyphs* typed constant in the *Dialogs* unit is declared like this:

```
MsgDlgGlyphs: Boolean = True;
```

This declaration ensures that bitmaps (or glyphs) appear on the message dialog box buttons. If you prefer that the bitmaps are not present, change the value of *MsgDlgButtonStyle* to *False*.

Example

This example displays a confirmation style message box at screen coordinates 125, 25 that asks users if they want to color the form green. If the user chooses Yes, the form turns bright green:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ButtonSelected: Word;
begin
  if MessageDlgPos('Color the form green?', mtConfirmation,
    [mbYes, mbNo], 0, 125, 25) := mrYes then
    Color := clLime;
end;
```

This example uses a button on a form. When the user clicks the button, a message box appears with a Yes, No, and Cancel button on it:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MessageDlgPos('Are you there?', mtConfirmation, mbYesNoCancel, 0, 200, 200);
end;
```

See also

Kind property, *MessageBox* method, *MessageDlg* function, *ModalResult* property, *ShowMessage* procedure, *ShowMessagePos* procedure

Metafile property

Applies to

TPicture object

Declaration

```
property Metafile: TMetafile
```

The *Metafile* property specifies the contents of the *TPicture* object as a Windows metafile graphic (.WMF file format). If *Metafile* is referenced when the *TPicture* contains a *Bitmap* or *Icon* graphic, the graphic won't be converted. Instead, the original contents of the *TPicture* are discarded and *Metafile* returns a new, blank metafile.

Example

The following line of code displays the pixels-per-inch of the coordinate mapping of a metafile. The *Inch* property of the metafile stored in the *MyGraphic* is converted to text and assigned to the *Caption* of *Label1*.

```
  Label1.Caption := IntToStr(MyGraphic.Metafile.Inch);
```

See also

Graphic property

Min property

Applies to

TScrollBar component

Declaration

```
property Min: Integer;
```

The *Min* property along with the *Max* property determines the number of possible positions the scroll box can have on the scroll bar. The *LargeChange* and *SmallChange* properties use the number of positions to determine how far to move the scroll box when the user uses the scroll bar.

For example, if *Max* is 3000 and *Min* is 0, the scroll box can assume 3000 positions on a horizontal scroll bar. If the *LargeChange* property setting is 1000 and the scroll box position is at the far left of the scroll bar (*Position* is 0), the user can click the scroll bar three times to the right of the scroll box before the scroll box is moved all the way to the right of the scroll bar (3000/1000 = 3).

If you want to change the *Min*, *Max*, and *Position* values all at run time, call the *SetParams* method.

Example

The following code sets the minimum position to the value specified in an edit box, and sets the maximum position to 1000 more than the minimum position.

```
ScrollBar1.Min := StrToInt(Edit1.Text);
ScrollBar1.Max := ScrollBar1.Min + 1000;
```

See also

LargeChange property, *Max* property, *Position* property, *SetParams* method, *SmallChange* property

MinFontSize property

Applies to

TFontDialog component

Declaration

```
property MinFontSize: Integer;
```

The *MinFontSize* property determines the smallest font size available in the Font dialog box. Use the *MinFontSize* property when you want to limit the font sizes available to the user. To limit the font sizes available, the *Options* set property of the Font dialog box must also contain the value *fdLimitSize*. If *fdLimitSize* is *False*, setting the *MinFontSize* property has no affect on number of fonts available in the Font dialog box.

The default value is 0, which means there is no minimum font size specified.

Example

This example uses a Font dialog box, a button, and a label on a form. When the user clicks the button, the Font dialog box appears. The font sizes available are within the range of 10 to 14. When the user chooses OK, the selected font is applied to the caption of the label.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  FontDialog1.Options := [fdLimitSize];
  FontDialog1.MaxFontSize := 14;
  FontDialog1.MinFontSize := 10;
  if FontDialog1.Execute then
    Label1.Font := FontDialog1.Font;
end;
```

See also

MinFontSize property

Minimize method

Applies to

TApplication component

Declaration

```
procedure Minimize;
```

The *Minimize* method shrinks your application into an icon on your Windows desktop.

Example

This example uses a button named *Shrink* on a form. When the user clicks the button, the application minimizes to an icon:

```
procedure TForm1.ShrinkClick(Sender: TObject);
begin
  Application.Minimize;
end;
```

See also

Application variable, *Icon* property

MinPage property

Applies to

TPrintDialog component

Declaration

```
property MinPage: Integer;
```

The *MinPage* property determines the smallest page number the user can use when specifying pages to print. If the user specifies a number less than the value of *MinPage*, a warning message appears and the user must enter a valid number or close the dialog box. The default value is 0.

Note The user can specify pages numbers only if the *Options* property set includes the value *poPageNums*.

Example

This example uses a button and a Print dialog on a form. When the user clicks the button, the code sets the lowest and the highest possible page numbers the user can select and displays the dialog box:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with PrintDialog1 do
  begin
```

```
      Options := [poPageNums];
      MinPage := 2;
      FromPage := 2;
      ToPage := 10;
      MaxPage := 10;
      PrintRange := prPageNums;
      Execute;
    end;
  end;
```

See also
MaxPage property

MinValue property

Applies to
TCurrencyField, *TFloatField*, *TIntegerField*, *TSmallintField*, *TWordField* component

Declaration

```
property MinValue: Longint;
```

The *MinValue* property limits the minimum value in the field. Assigning a value less than *MinValue* raises an exception.

Example

```
  { Limit the field to 1 to 10}
  Field1.MaxValue := 10;
  Field1.MinValue := 1;
```

See also
MaxValue property

MkDir procedure System

Declaration

```
procedure MkDir(S: string);
```

The *MkDir* procedure creates a new subdirectory with the path specified by string *S*. The last item in the path cannot be an existing file name.

{$I+} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using **{$I–}**, you must use *IOResult* to check for an I/O error.

CreateDir performs the same function as *MkDir*, but it takes a null-terminated string rather than a Pascal-style string.

Example

```
uses Dialogs;

begin
  {$I-}
  { Get directory name from TEdit control }
  MkDir(Edit1.Text);
  if IOResult <> 0 then
    MessageDlg('Cannot create directory', mtWarning, [mbOk], 0)
  else
    MessageDlg('New directory created', mtInformation, [mbOk], 0);
end;
```

See also

ChDir procedure, *GetDir procedure*, *RmDir procedure*

ModalResult property

Applies to

TBitBtn, *TButton*, *TForm* components

Declaration

```
property ModalResult: TModalResult;
```

Run-time only. The *ModalResult* property for forms is used to terminate a modal form. By default, *ModalResult* is 0. Setting *ModalResult* to any nonzero value ends the form's modal state. When the user chooses to close a modal form, the button click sets *ModalResult* to close the form. The value assigned to *ModalResult* becomes the return value of the *ShowModal* function call which displayed the modal form.

Button controls have a *ModalResult* property also that is read only. Use a button's *ModalResult* property when you want a click of the button to close a modal form. For example, if you create a dialog box with two buttons, OK and Cancel, set the *ModalResult* property to *mrOK* for the OK button and *mrCancel* for the Cancel button. When the user chooses either of these two buttons, the dialog box's modal state ends because *ModalResult* is greater than *mrNone* and the dialog box disappears. Using *ModalResult*, you don't have to write an event handler just to close the dialog box.

These constants are possible *ModalResult* values:

Constant	Value
mrNone	*0*
mrOk	*idOK*
mrCancel	*idCancel*
mrAbort	*idAbort*
mrRetry	*idRetry*
mrIgnore	*idIgnore*

Constant	Value
mrYes	*idYes*
mrNo	*idNo*
mrAll	*mrNo* + 1

Example

The following methods in a form are used as a modal dialog box. The methods cause the dialog box to terminate when the user clicks either the OK or Cancel button, returning *mrOK* or *mrCancel* from *ShowModal*, respectively:

```
procedure TMyDialogBox.OKButtonClick(Sender: TObject);
begin
  ModalResult := mrOK;
end;

procedure TMyDialogBox.CancelButtonClick(Sender: TObject);
begin
  ModalResult := mrCancel;
end;
```

You could also set the *ModalResult* value to *mrOK* for the OK button and *mrCancel* for the Cancel button to accomplish the same thing. When the user clicks either button, the dialog box closes.

See also

Kind property, *ShowModal* method

M

Mode property

Applies to

TPen object; *TMediaPlayer* component

For pen objects

Declaration

```
property Mode: TPenMode;
```

The *Mode* property determines how the pen draws lines on the canvas. The following table describes the behavior for each pen mode.

Mode	Pixel color
pmBlack	Always black.
pmWhite	Always white.
pmNop	Unchanged.
pmNot	Inverse of screen color.
pmCopy	Pen color specified in *Color* property.

Mode	Pixel color
pmNotCopy	Inverse of pen color.
pmMergePenNot	Combination of pen color and inverse of screen color.
pmMaskPenNot	Combination of colors common to both pen and inverse of screen.
pmMergeNotPen	Combination of screen color and inverse of pen color.
pmMaskNotPen	Combination of colors common to both screen and inverse of pen.
pmMerge	Combination of pen color and screen color.
pmNotMerge	Inverse of *pmMerge* combination of pen color and screen color.
pmMask	Combination of colors common to both pen and screen.
pmNotMask	Inverse of *pmMask* combination of colors common to both pen and screen.
pmXor	Combination of colors in either pen or screen, but not both.
pmNotXor	Inverse of *pmXor* combination of colors in either pen or screen, but not both.

Example

The following code sets the mode of the pen of the *Canvas* of *Form1* to the inverse of the pen *Color*.

```
Form1.Canvas.Pen.Mode := pmNotCopy;
```

See also

Pen property, *TPen* object

For media player controls

Declaration

```
property Mode: TMPModes;
```

Run-time and read only. The *Mode* property specifies the mode of the currently open multimedia device. The following table lists the possible values for *Mode*:

Value	Mode
mpNotReady	Not ready
mpStopped	Stopped
mpPlaying	Playing
mpRecording	Recording
mpSeeking	Seeking
mpPaused	Paused
mpOpen	Open

Example

The following code declares an array of strings named *ModeStr*, indexed by the *TMPModes* type. The *Caption* of a form is then set to the string describing the current mode of the device:

```
const
```

```
    ModeStr: array[TMPModes] of string[10] = ('Not ready', 'Stopped', 'Playing',
      'Recording', 'Seeking', 'Paused', 'Open');
  {Later in your code}
  Caption := ModeStr[MediaPlayer1.Mode];
```

For batch move components

Declaration

```
property Mode: TBatchMode;
```

The *Mode* property specifies what the *TBatchMove* object will do:

Property	Purpose
batAppend	Append records to the destination table. The destination table must already exist. This is the default mode.
batUpdate	Update records in the destination table with matching records from the source table. The destination table must exist and must have an index defined to match records.
batAppendUpdate	If a matching record exists in the destination table, update it. Otherwise, append records to the destination table. The destination table must exist and must have an index defined to match records.
batCopy	Create the destination table based on the structure of the source table. The destination table must not already exist—if it does, the operation will delete it.
batDelete	Delete records in the destination table that match records in the source table. The destination table must already exist and must have an index defined.

Example

```
  BatchMove1.Mode := batAppendUpdate;
```

Modified property

Applies to
TBitmap, *TGraphic*, *TIcon*, *TMetafile* objects, *TDBEdit*, *TDBMemo*, *TEdit*, *TMaskEdit*, *TMemo*, *TOLEContainer*, *TQuery*, *TStoredProc*, *TTable* components

For graphics objects

Declaration

```
property Modified: Boolean;
```

The *Modified* property specifies if the graphics object has been changed or edited. If *Modified* is *True*, the graphics object has changed. If *Modified* is *False*, the graphics object is in the same state as when the object was loaded.

Note The *Modified* property only indicates if bitmap objects have been modified. *Modified* is not *True* if the graphics object contains an icon or metafile graphic.

If the graphics object was modified, you can save the changes to a file with the *SaveToFile* method. The next time the application is run, the object can be loaded from the file with the *LoadFromFile* method.

Example

The following code saves the bitmap object in *Graphic1* to a file if it was modified.

```
if Graphic1.Modified then Graphic1.SaveToFile('myfile.bmp');
```

For OLE containers

Declaration

```
property Modified: Boolean;
```

The *Modified* property specifies if the OLE object in an OLE container component was changed or edited since the OLE container was initialized. If *Modified* is *True*, the OLE object was changed. If *Modified* is *False*, the OLE object is in the same state as when the OLE container was initialized.

If the OLE object was modified, changes to the object are lost when the OLE container application is closed unless the object is saved to a file with the *SaveToFile* method. The next time the OLE container application is run, the object should be loaded from the file with the *LoadFromFile* method.

Example

The following code saves the object in *OLEContainer1* to the file OBJ.OLE if it was modified.

```
if OLEContainer1.Modified then OLEContainer1.SaveToFile('OBL.OLE');
```

For edit boxes and memos

Declaration

```
property Modified: Boolean;
```

Run-time only. The *Modified* property determines whether the text of an edit box or memo control was changed since it was created or since the last time the *Modified* property was set to *False*. If *Modified* is *True*, the text was changed. If *Modified* is *False*, the text was not changed.

Example

```
procedure TForm1.Button1Click(Sender: TObject);
begin
if Edit1.Modified = True then
  begin
    MessageDlg('Edit box text was modified',
      mtInformation, [mbOK], 0);
    Edit1.Modified := False;
  end
  else
    MessageDlg('Edit box text was not modified',
      mtInformation, [mbOK], 0);
end;
```

See also
Text property

For tables, queries, and stored procedures

Declaration

```
property Modified: Boolean;
```

Run-time and read only. The *Modified* property is *True* if a field in the current record has been changed. It is reset to *False* when the record is updated through a call to the *Cancel* or *Post* methods.

See also
UpdateRecord method

Monochrome property

Applies to
TBitmap object

Declaration

```
property Monochrome: Boolean;
```

The *Monochrome* property determines if the bitmap displays in monochrome. If *True,* the bitmap is monochrome. If *False,* the bitmap displays in color.

Example
The following code create *Bitmap1* and sets its *Monochrome* property to *True*.

```
var
  Bitmap1: TBitmap;
begin
  Bitmap1 := TBitmap.Create;
  Bitmap1.Monochrome := True;
end;
```

MouseToCell method

Applies to

TDrawGrid, *TStringGrid* components

Declaration

```
procedure MouseToCell(X, Y: Integer; var ACol, ARow: Longint);
```

The *MouseToCell* method returns the column and row of the cell the mouse pointer is positioned on. The X and Y parameters are the screen coordinates of the mouse pointer. The *ACol* parameter is the number of the column where the mouse pointer is positioned, and the *ARow* parameter is the number of the row.

Usually the *MouseToCell* method is used in a mouse event handler, which supplies the mouse coordinates to the method call.

Example

This example uses a string grid on a form. When the user selects a cell in the grid and releases the mouse button, the column and row coordinates for the cell appear in the cell. The code for displaying the coordinates is written in the *OnMouseUp* event handler:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  StringGrid1.DefaultColWidth := 100;
end;

procedure TForm1.StringGrid1MouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  Column, Row: Longint;
begin
  StringGrid1.MouseToCell(X, Y, Column, Row);
  StringGrid1.Cells[Column, Row] := 'Col ' + IntToStr(Column) +
    ',Row ' + IntToStr(Row);
end;
```

See also

CellRect method, *OnMouseDown* event, *OnMouseMove* event, *OnMouseUp* event

Move method

Applies to

TList, *TStringList*, *TStrings* objects

Declaration

```
procedure Move(CurIndex, NewIndex: Integer);
```

The *Move* method changes the position of an item in the list of a list object or in a list of strings in a string object by giving the item a new index value. The *CurIndex* parameter is the item's current index, and the *NewIndex* parameter is the item's new index value.

If a string in a string object has an associated object in the *Objects* property, *Move* moves both the string and the object.

Example

This example uses a list box and a button on a form. The list box contains items when the form appears. When the user clicks the button, the fifth item in the list box is moved to the top of the list box:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 5 do
    ListBox1.Items.Add('Item ' + IntToStr(I));
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Move(4, 0);
end;
```

See also

Add method, *Delete* method, *Exchange* method, *Objects* property, *Strings* property

Move procedure **System**

Declaration

```
procedure Move(var Source, Dest; Count: Word);
```

The *Move* procedure copies *Count* bytes from a *Source* to *Dest*. No range-checking is performed.

When the segment parts of *Source* and *Dest* are equal, *Move* compensates for overlaps between the source and destination blocks. If the source and destination overlap but their segment parts are not equal, *Move* will not compensate for overlaps and there is a 50% chance that *Move* will not work correctly. Borland Pascal's static and dynamic (heap) memory allocation schemes never create overlapping variables whose addresses have different segment parts, so this problem can only occur if the addresses of *Source* and *Dest* are modified or normalized by your program, or if they are provided by an external source.

Whenever possible, use *SizeOf* to determine the count.

Example

```
var
```

```
    A: array[1..4] of Char;
    B: Longint;
  begin
    Move(A, B, SizeOf(A));            { SizeOf = safety! }
  end;
```

See also
FillChar procedure, *SizeOf function*

MoveBy method

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure MoveBy(Distance: Integer);
```

The *MoveBy* method moves the dataset cursor by *Distance* records. If *Distance* is negative, the move is backward. If *Distance* is positive, the movement is forward. If *Distance* is zero, no move is done.

If the dataset is in Insert or Edit state, *MoveBy* will perform an implicit *Post* of any pending data.

Example

```
  { Skip three records forward }
  Table1.MoveBy(3);
```

See also
First method, *Last* method, *Next* method, *Prior* method

MovedCount property

Applies to
TBatchMove component

Declaration

```
property MovedCount: Longint;
```

Run-time and read only. *MovedCount* is the number of records which were actually processed by the *Execute* method. This includes any records which had integrity or data size problems.

Example

```
with BatchMove1 do
  begin
  Execute;
  MessageDlg(IntToStr(MoveCount) + ' records read', mtInformation, [mbOK], 0);
  end;
```

See also

ChangedCount property, *KeyViolCount* property, *ProblemCount* property, *RecordCount* property

MoveTo method

Applies to

TCanvas, TOutlineNode objects

For canvases

Applies to

TCanvas object

Declaration

```
procedure MoveTo(X, Y: Integer);
```

The *MoveTo* method changes the current drawing position to the coordinates passed in *X* and *Y*. The current position is given by the *PenPos* property. You should use *MoveTo* to set the current position rather than setting *PenPos* directly.

Example

The following code draws a line from the upper-eft corner of a form to the point clicked with the mouse:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.MoveTo(0, 0);
  Canvas.LineTo(X, Y);
end;
```

See also

LineTo method

For outline nodes

Applies to

TOutlineNode object

Declaration

```
procedure MoveTo(Destination: Longint; AttachMode: TAttachMode);
```

Description

The *MoveTo* method moves an outline item from one location to another within an outline (*TOutline* component). The *Destination* parameter determines where to move the item. Pass the *Index* value of another outline item in the *Destination* parameter. The *AttachMode* parameter specifies how you want to attach the item to the destination position. These are the possible values of *AttachMode*:

Value	Meaning
oaAdd	The item is attached as if added with the *Add* method. The moved item becomes the last sibling of the item specified by the *Destination* parameter. The moved item will share the same parent as the *Destination* item.
oaAddChild	The item is attached as if added with the *AddChild* method. The moved item becomes the last child of the item specified by the *Destination* parameter. The *Destination* item will become the parent of the moved item.
oaInsert	The item is attached as if inserted with the *Insert* method. The moved item replaces the *Destination* item in the outline, while the *Destination* item and all other following items are moved down one row.

MoveTo returns the new *Index* value of the moved item.

Note When an item is moved, all its subitems move with it.

When an item (and any subitems) is moved, the other items in the outline are reindexed to obtain new valid *Index* values. This happens automatically unless *BeginUpdate* has been called.

Example

The following code moves the selected item to become the first item in the outline.

```
with Outline1.Items[Outline1.SelectedItem] do
  MoveTo(0, oaInsert);
```

See also

ChangeLevelBy method

MSecsPerDay constant **SysUtils**

Declaration

```
MSecsPerDay = 24 * 60 * 60 * 1000;
```

MSecsPerDay declares the number of milliseconds per day.

MultiSelect property

Applies to
TListBox, *TFileListBox* components

Declaration

```
property MultiSelect: Boolean;
```

The *MultiSelect* property determines whether the user can select more than one element at a time from the list. If *MultiSelect* is *True*, the user can select multiple items. If *MultiSelect* if *False*, multiple items can be selected in the list box at the same time. The default value is *False*.

Example
This line of code ensures that the user can select multiple items in a list box:

```
   ListBox1.MultiSelect := True;
```

See also
ExtendedSelect property, *Selected* property

Name property

Applies to
All components; *TFieldDef*, *TFieldDefs*, *TIndexDef*, *TIndexDefs*, *TFont*, *TParam*, *TParams* objects

For components

Declaration

```
property Name: TComponentName;
```

The *Name* property contains the name of the component as referenced by other components. By default, Delphi assigns sequential names based on the type of the component, such as ‘Button1’, ‘Button2’, and so on. You may change these to suit your needs.

Note Change component names only at design time.

Example

The following code lists the names of all the components of *Form1* in a list box.

```
var
  I: Integer;
begin
  for I := 0 to Form1.ComponentCount-1 do
    ListBox1.Items.Add(Form1.Components[I].Name);
end;
```

For font objects

Declaration

```
property Name: TFontName;
```

The *Name* property of a font object determines the name of the font contained within the font object.

Example

This code sets the font for all text that appears on the form to Times New Roman. If the controls on the form have their *ParentFont* property set to *True*, text on these controls will also be in Times New Roman.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Font.Name := 'Times New Roman';
end;
```

See also

ParentFont property

For TIndexDef objects

Declaration

```
property Name: string;
```

Run-time and read only. *Name* is the name of the index.

For TParam objects

Declaration

```
property Name: string;
```

The *Name* property is the name of the parameter.

Example

```
{ Change the name of the first parameter column to 'CustNo' }
Params[0].Name := 'CustNo';
```

For TFieldDef objects

Declaration

```
property Name: string;
```

Run-time and read only. *Name* is the name of the physical field within the table.

Example

```
{ Display the field name and number }
with FieldDef1 do
  MessageDlg(Name + ' is field ' + IntToStr(FieldNo), mtInformation, [mbOK], 0);
```

See also

TField component

NativeToAnsi procedure **DB**

Declaration

```
procedure NativeToAnsi(Locale: TLocale; NativeStr: PChar; var AnsiStr: string);
```

The *NativeToAnsi* procedure translates native characters in *NativeStr* to the ANSI character set according to *Locale*. *NativeToAnsi* returns the translated string in *AnsiStr*.

NetFileDir property

Applies to

TSession component

Declaration

```
property NetFileDir: string;
```

Run-time only. The *NetFileDir* property specifies the directory that contains the BDE network control file, PDOXUSRS.NET. This property enables multiple users to share Paradox tables on network drives. *NetFileDir* overrides the specification defined for the Paradox driver in the BDE Configuration Utility.

All applications that need to share the same Paradox database must specify the same directory, and all must have read/write/create rights for the directory.

See also
Session variable

New procedure **System**

Declaration

```
procedure New(var P: Pointer);
```

```
function New( <pointer type> ): Pointer;
```

The *New* procedure creates a new dynamic variable and sets a pointer variable to point to it. Reference the newly created variable as *P*^.

If there is not enough space available in the heap to allocate to the new variable a run-time error occurs. However, **{$I+}** lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

The *New* function returns a pointer value and applies to all data types, not just object types.

The parameter passed to *New* is the type of pointer pointing to the object, rather than the pointer variable itself.

Example

```
type
  Str18 = string[18];
var
  P: ^Str18;
begin
  New(P);
  P^ := 'Now you see it...';
  Dispose(P);                   { Now you don't... }
end;
```

See also
Dispose procedure, FreeMem procedure, GetMem procedure

NewPage method

Applies to
TPrinter object

Declaration

```
procedure NewPage;
```

The *NewPage* method forces the current print job to begin printing on a new page in the printer. It also increments the value of the *PageNumber* property and resets the value of the *Pen* property of the *Canvas* back to (0, 0).

Example

This example uses a button on a form. When the user clicks the button, a rectangle is printed twice, one per page.

To run this example successfully, you must add the *Printers* unit to the **uses** clause of your unit.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with Printer do
  begin
    BeginDoc;
    Canvas.Rectangle(10, 10, 200, 200);
    NewPage;
    Canvas.Rectangle(10, 10, 200, 200);
    EndDoc;
  end;
end;
```

See also

BeginDoc method, *EndDoc* method, *Printer* variable

NewStr function SysUtils

Declaration

```
function NewStr(const S: string): PString;
```

The *NewStr* function allocates a copy of the string *S* on the heap and returns a pointer to the newly allocated string. When your application finishes using the allocated string, you should use *DisposeStr* to dispose of the string on the heap.

Do not change the length of strings allocated with *NewStr*. Increasing the length of the string overwrites other variables on the heap. Decreasing the length of the string prevents some of the memory from being deallocated.

Example

The following code allocates space for and places the string ‘New String’ in memory. The pointer *S* points to the new string:

```
var
  S: PString;
begin
  S := NewStr('New String');
.
.
DisposeStr(S);
```

See also
DisposeStr procedure

Next method

Applies to
TForm, *TMediaPlayer*, *TQuery*, *TStoredProc*, *TTable* components

The *Next* method either activates the next form, media track, or record.

For forms

Declaration

```
procedure Next;
```

The *Next* method makes the next child form in the form sequence the active form.

For example, if you have three child forms within a parent form in your MDI application and *Form2* is the active form, the *Next* method makes *Form3* the active form. Calling *Next* again makes *Form4* active. The next time your application calls *Next*, the sequence starts over again and *Form2* becomes the active form once again.

The *Next* method applies only to forms that are MDI parent forms (have a *FormStyle* property value of *fsMDIForm*).

Example
The following code activates the next child of *Form1*.

```
  Form1.Next;
```

See also
ArrangeIcons method, *Cascade* method, *Previous* method, *Tile* method

For media player controls

Declaration

```
procedure Next;
```

The *Next* method goes to the beginning of the next track of the currently loaded medium. If the current position is at the last track when *Next* is called, *Next* makes the current position the beginning of the last track. If the multimedia device doesn't use tracks, *Next* goes to the end of the medium. *Next* is called when the Next button on the media player control is clicked at run time.

Upon completion, *Next* stores a numerical error code in the *Error* property, and the corresponding error message in the *ErrorMessage* property.

The *Wait* property determines whether control is returned to the application before the *Next* method has completed. The *Notify* property determines whether *Next* generates an *OnNotify* event.

Example

The following code opens a WAV audio file and fast-forwards to the end of the medium.

```
MediaPlayer1.DeviceType := dtWAVAudio;
MediaPlayer1.FileName := 'c:\chimes.wav';
MediaPlayer1.Open;
MediaPlayer1.Next;
```

See also

Position property, *Previous* method, *Tracks* property

For tables, queries, and stored procedures

Declaration

```
procedure Next;
```

The *Next* method moves the cursor forward by one record. If the cursor is already on the last record, it does not move. If the dataset is in Insert or Edit state, *Next* will perform an implicit *Post* of any pending data.

Example

```
{ Move to the next record }
Table1.Next;
if Table1.Eof then { No more records };
```

See also

First method, *Last* method, *MoveBy* method, *Prior* method

NormalizeTopMosts method

Applies to

TApplication component

Declaration

```
procedure NormalizeTopMosts;
```

The *NormalizeTopMosts* method makes forms that have been designated as topmost forms (their *FormStyle* is *fsStayOnTop*) behave as if they were not topmost forms. You'll find this method convenient to use if you want a message box or dialog box to appear on top of a topmost form.

For example, while you do not have to call *NormalizeTopMosts* to use the Delphi methods and functions that display message boxes (such as *MessageBox* and *MessageDlg*), you should call it if you want to call Windows API functions directly to display a message box. If you neglect to call *NormalizeTopMosts*, the message box won't display on top of the form, but the form remains on top. Any time you call Windows API functions to display a window on top of a form, call *NormalizeTopMosts* first.

To return the forms designated as *fsStayOnTop* to be topmost again, call *RestoreTopMosts*.

Example

The following code normalizes topmost forms before calling the *MessageBox* function in the *WinProcs* unit. After the message box is closed, the topmost forms are restored.

```
begin
  Application.NormalizeTopMosts;
  MessageBox(Form1.Handle, 'This should be on top.', 'Message Box', MB_OK);
  Application.RestoreTopMosts;
end;
```

See also

FormStyle property, *RestoreTopMosts* method

Notify property

Applies to

TMediaPlayer component

Declaration

```
property Notify: Boolean;
```

Run-time only. The *Notify* property determines whether the next call to a media control method (*Back, Close, Eject, Next, Open, Pause, PauseOnly, Play, Previous, StartRecording, Resume, Rewind, Step,* or *Stop*) generates an *OnNotify* event when the method has completed.

If *Notify* is *True*, the next media control method generates *OnNotify* event upon completion and stores the notification message in the *NotifyValue* property. If *Notify* is *False*, the method does not generate an *OnNotify* event and *NotifyValue* remains unchanged.

Notify affects only the next call to a media control method. After an *OnNotify* event, *Notify* must be reset to affect any subsequent media control methods.

By default, *Play* and *StartRecording* function as if *Notify* is *True*. You must set *Notify* to *False* before calling *Play* or *StartRecording* to prevent an *OnNotify* event from being generated when playing or recording has finished. By default, all other media control methods function as if *Notify* is *False*.

Note Set *Notify* to *True* if the next media control is expected to take a long time, so your application is notified when the media control method has completed. If you set *Notify* to *True*, you might want to set *Wait* to *False* so that control returns to the application before the media control method is finished.

Note If you try to resume a device that doesn't support *Resume*, the device is resumed as if you called the *Play* method. If you have assigned *True* to *Notify* before calling *Resume* (or any other media control method), *Notify* doesn't affect the call to *Resume*. *Resume* does not generate an *OnNotify* event upon completion, and *NotifyValue* remains unchanged.

Example

The following code sets *Notify* to *True* after opening and playing the Microsoft Video for the Windows file named DUCK.AVI. When the *Play* method is completed, an *OnNotify* event occurs, which displays a message.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  with MediaPlayer1 do begin
    FileName := 'duck.avi';
    Open;
    Play;
    Notify := True;
  end;
end;
procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
  if MediaPlayer1.NotifyValue=nvSuccessful then
    MessageDlg('Done playing video.', mtInformation, [mbOK], 0);
end;
```

NotifyValue property

Applies to

TMediaPlayer component

Declaration

```
property NotifyValue: TMPNotifyValues;
```

Run-time and read only. The *NotifyValue* property reports the result of the last media control method (*Back*, *Close*, *Eject*, *Next*, *Open*, *Pause*, *PauseOnly*, *Play*, *Previous*, *StartRecording*, *Resume*, *Rewind*, *Step*, or *Stop*) that requested a notification. Set *Notify* to *True* before calling a media control method to request notification.

The following table lists the possible values for *NotifyValue*.

Value	Result
nvSuccessful	Command completed successfully
nvSuperseded	Command was superseded by another command

Value	Result
nvAborted	Command was aborted by the user
nvFailure	Command failed

Example

This example uses a media player component named *MediaPlayer1*. When the application runs, the code attempts to play a CD in the CD audio device, and displays a message dialog box indicating whether the attempt to play the CD was successful.

Before you can run this example, you must have a CD audio device installed correctly.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  with MediaPlayer1 do
  begin
    DeviceType := dtCDAudio;
    Open;
     Play;
     if NotifyValue <> nvSuccessful then
       MessageDlg('Error playing CD audio', mtError, [mbOk], 0)
     else
       MessageDlg('Playing CD audio', mtInformation, [mbOk], 0);
    Visible := False;
  end;
end;
```

See also

OnNotify event

Now function — SysUtils

Declaration

```
function Now: TDateTime;
```

The *Now* function returns the current date and time, corresponding to *Date* + *Time*.

Example

This example uses a label and a button on a form. When the user clicks the button, the current date and time appear as the caption of the label.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := 'The date and time is  ' + DateTimeToStr(Now);
end;
```

See also

Date function, *DateTimeToStr* function, *Time* function

NullStr constant

SysUtils

Declaration

```
NullStr: PString = @EmptyStr;
```

NullStr declares a pointer to *EmptyStr*.

NullStr is the return value for many string handling routines when the string is empty.

NumGlyphs property

Applies to

TBitBtn, *TSpeedButton* components

Declaration

```
property NumGlyphs: TNumGlyphs;
```

The *NumGlyphs* property indicates the number of images that are in the graphic specified in the *Glyph* property for use on a bitmap button or speed button.

If you have multiple images in a bitmap, you must specify the number of images that are in the bitmap with the *NumGlyphs* property. All images must be the same size and next to each other in a row. Valid *NumGlyphs* values are 1 to 4. The default value is 1.

You can provide up to four images on a bitmap button or speed button with a single bitmap. Delphi then displays one of these images depending on the state of the button. Only one image is required in a bitmap.

Image position in bitmap	Speed button state	Description
First	Up	This image appears when the button is unselected. If no other images exist in the bitmap, Delphi uses this image for all other images.
Second	Disabled	This image usually appears dimmed and indicates that the button can't be selected.
Third	Down	This image appears when a button is clicked. The up state image then reappears when the user releases the mouse button.
Fourth	Stay down	This image appears when a button stays down indicating that it remains selected.

If only one image is present, Delphi attempts to represent the other states by altering the image slightly for the different states, although the stay down state is always the same as the up state. If you aren't satisfied with the results, you can provide additional images in the bitmap.

Example

This example uses a speed button and a label on a form. When the example runs, the number of images in the specified bitmap appears as the caption of the label.

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  SpeedButton1.Glyph.LoadFromFile('C:\WINDOWS\CARS.BMP');
  Label1.Caption := IntToStr(SpeedButton1.NumGlyphs) + ' image(s)';
end;
```

See also
Glyph property

ObjClass property

Applies to
TOLEContainer component

Declaration
```
property ObjClass: string;
```

Specify the OLE class of an object in the *ObjClass* property. The class of an object is typically the application name of the OLE server application without the .EXE extension. See the documentation for the OLE server for specific information about its OLE class.

At design time, specifying the *ObjClass* property displays the Insert Object dialog box and initializes the OLE object. At run time, the *ObjClass* property is specified automatically when you initialize the OLE object with the *PInitInfo* property.

Example
The following code tests to determine if “Paintbrush Picture” is the object class. If so, a message is displayed in *Label1* when *Button1* is clicked.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OLEContainer1.ObjClass = 'Paintbrush Picture' then
    Label1.Caption := 'The object is a Paintbrush Picture';
end;
```

See also
ObjDoc property, *ObjItem* property

ObjDoc property

Applies to
TOLEContainer component

Declaration
```
property ObjDoc: string;
```

Specify the OLE document of an object in the *ObjDoc* property. The document of an object is typically the name of the file containing the OLE information. See the OLE server documentation for specific information about its OLE documents.

At design time, specifying the *ObjDoc* property displays the Insert Object dialog box and initializes the OLE object. At run time, the *ObjDoc* property is specified automatically when you initialize the OLE object with the *PInitInfo* property, if the object is linked to data in a file.

Example

The following code tests to determine if 'c:\windows\256color.bmp' is the object document. If so, a message is displayed in *Label1* when *Button1* is clicked.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OLEContainer1.ObjDoc = 'c:\windows\256color.bmp' then
    Label1.Caption := 'The object document is c:\windows\256color.bmp';
end;
```

See also

ObjClass property, *ObjItem* property

ObjectMenuItem property

Applies to

TForm component

O

Declaration

```
property ObjectMenuItem: TMenuItem;
```

ObjectMenuItem is used to specify the OLE object menu item. If you create a menu item and specify it as the OLE object menu item with the *ObjectMenuItem* property, the item is automatically enabled when an OLE object in an OLE container is selected.

The OLE object menu item can be used to activate or convert the selected object. All you need to do is specify the menu item in the *ObjectMenuItem* component. The processing of activating or converting the object is handled by the OLE server application.

Example

The following code assigns *MyObject1* to the *ObjectMenuItem* property of *Form1*. When an OLE container that contains an object is selected, the caption of the *MyObject1* menu item can be modified by the OLE server and the functionality of *MyObject1* will be handled by the server.

```
Form1.ObjectMenuItem := MyObject1;
```

Objects property

Applies to

TStringList, *TStrings* objects; *TStringGrid* component

For string objects

Declaration

```
property Objects[Index: Integer]: TObject;
```

Run-time only. The *Objects* property gives you access to an object in the list of objects associated with the list of strings. Each string in the list of strings can have an associated object.

The most common use of objects in a string and string list objects is to associate bitmaps with strings so that you can use the bitmaps in owner-draw controls. For example, if you have an owner-draw list box, you can add a string 'Banana' and a bitmap of a banana to the *Items* property of the list box using the *AddObject* method. You can then access the 'Banana' string using the *Strings* property or the bitmap using the *Objects* property.

Specify the object you want to access with its position in the list as the value of the *Index* parameter. The index is zero-based, so the first object in the list of objects has a value of 0, the second object has a value of 1, and so on.

To associate an object with an existing string, assign the object to the *Objects* property using the same index as that of the existing string in the *Strings* property. For example, if a string object named *Fruits* contains the string 'Banana' and an existing bitmap of a banana called BananaBitmap, you could make the following assignment:

```
Fruits.Objects[Fruits.IndexOf('Banana')] := BananaBitmap;
```

Example

The following code allows the user to specify a bitmap file with the *OpenDialog1* open dialog box component when *Form1* is created. Then, the bitmap file specified is added to the *Items* list of *ListBox1*.

If *ListBox1* is an owner-draw control (specified by a *Style* property of *lbOwnerDrawFixed* or *lbOwnerDrawVariable*), the second procedure is the *OnDrawItem* event handler for *ListBox1*. The bitmap in the *Object* property and the text of an item are retrieved and displayed in *Listbox1*.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  TheBitmap: TBitmap;
begin
  if OpenDialog1.Execute then
  begin
    TheBitmap := TBitmap.Create;
    TheBitmap.LoadFromFile(OpenDialog1.FileName);
```

```
      ListBox1.Items.AddObject(OpenDialog1.FileName, TheBitmap);
    end;
  end;

  procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index: Integer;
    Rect: TRect; State: TOwnerDrawState);
  var
    DrawBitmap: TBitmap;
  begin
    DrawBitmap := TBitmap(ListBox1.Items.Objects[Index]);
    with ListBox1.Canvas do
    begin
      Draw(Rect.Left, Rect.Top + 4, DrawBitmap);
      TextOut(Rect.Left + 2 + DrawBitmap.Width, Rect.Top + 2, ListBox1.Items[Index]);
    end;
  end;
```

See also
AddObject method, *InsertObject* method, *IndexOf* method, *IndexOfObject* method

For string grids

Declaration

```
property Objects[ACol, ARow: Integer]: TObject;
```

Run-time only. The *Objects* property is an array of objects, one for each cell in the grid. The *ColCount* and *RowCount* values define the size of the array of objects. Use the *Objects* property to access an object within a particular cell. *ACol* is the column coordinate of the cell, and *ARow* is the row coordinate of the cell.

If you put an object into the *Objects* array, the object will still exist even if the string grid is destroyed. You must destroy the object explicitly.

Example
The following code stores a *TBitmap* object called *MyBitmap* in row 3, column 10 of *StringGrid1*.

```
StringGrid1.Objects[10, 3] := MyBitmap;
```

See also
Cells property, *Cols* property, *Free* method, *Rows* property

ObjItem property

Applies to
TOLEContainer component

Declaration

```
property ObjItem: string;
```

Specify the OLE item of an object in the *ObjItem* property. The item of an object is a discrete unit of data within the OLE document containing the OLE information. See the OLE server documentation for specific information about its OLE documents.

At design time, specifying the *ObjItem* property displays the Paste Special dialog box and initializes the OLE object. At run time, the *ObjItem* property is specified automatically when you initialize the OLE object with the *PInitInfo* property, if the object linked is a more specific piece of data than is specified by the *ObjDoc* property.

Example

The following code tests to determine if “29 8 337 96” is the object item (this could be the item if you copied a portion of a Paintbrush picture to the Clipboard and wanted to link to the bitmap defined by the coordinates (29, 8) and (337, 96)). If so, a message is displayed in *Label1* when *Button1* is clicked.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OLEContainer1.ObjDoc = '29 8 337 96' then
    Label1.Caption := 'The object item is 29 8 337 96';
end;
```

See also

ObjClass property

Odd function **System**

Declaration

```
function Odd(X: Longint): Boolean;
```

The *Odd* function tests if the argument is an odd number.

Odd returns *True* if *X* is an odd number.

Example

```
begin
   if Odd(5) then
     Canvas.TextOut(10, 10, '5 is odd.')
   else
     Canvas.TextOut(10, 10, 'Something is odd!');
 end;
```

OEMConvert property

Applies to

TEdit, *TMemo* components

Declaration

```
property OEMConvert: Boolean;
```

Description

The OEMConvert property determines whether the text in the control is converted to OEM characters. If *True*, the text is converted. If *False*, the characters remain as ANSI characters. The default value is *False*. You should have the text converted to OEM characters if the text consists of file names.

Ofs function **System**

Declaration

```
function Ofs(X): Word;
```

The *Ofs* function returns the offset of a specified object.

X is any variable, or a procedure or function identifier. The result of type *Word* is the offset part of the address of *X*.

Example

```
function MakeHexWord(w: Word): string;
const
  hexChars: array [0..$F] of Char ='0123456789ABCDEF';
var
  HexStr : string;
begin
  HexStr := '';
  HexStr := HexStr + hexChars[Hi(w) shr 4];
  HexStr := HexStr + hexChars[Hi(w) and $F];
  HexStr := HexStr + hexChars[Lo(w) shr 4];
  HexStr := HexStr + hexChars[Lo(w) and $F];
  MakeHexWord := HexStr;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  Y: Integer;
  S: string;
begin
  Y := 10;
  S := 'The current code segment is $' + MakeHexWord(CSeg);
```

```
    Canvas.TextOut(5, Y, S);
    Y := Y + Canvas.TextHeight(S) + 5;
    S := 'The global data segment is $' + MakeHexWord(DSeg);
    Canvas.TextOut(5, Y, S);
    Y := Y + Canvas.TextHeight(S) + 5;
    S := 'The stack segment is $' + MakeHexWord(SSeg);
    Canvas.TextOut(5, Y, S);
    Y := Y + Canvas.TextHeight(S) + 5;
    S := 'The stack pointer is at $' + MakeHexWord(SPtr);
    Canvas.TextOut(5, Y, S);
    Y := Y + Canvas.TextHeight(S) + 5;
    S := 'i is at offset $' + MakeHexWord(Ofs(i));
    Canvas.TextOut(5, Y, S);
    Y := Y + Canvas.TextHeight(S) + 5;
    S := 'in segment $' + MakeHexWord(Seg(i));
    Canvas.TextOut(5, Y, S);
  end;
```

See also

Addr function, Seg function

OLEObjAllocated method

Applies to

TOLEContainer component

Declaration

```
function OleObjAllocated: Boolean;
```

The *OLEObjAllocated* method specifies whether an OLE container has been initialized and therefore contains an OLE object. *OLEObjAllocated* returns *True* if an OLE object has been allocated, or *False* if the OLE container is empty.

Example

The following code only initializes *OLEContainer1* if it does not already contain an OLE object, assuming *TheInitInfo* points to valid initialization information.

```
  if not OLEContainer1.OLEObjAllocated then
    OLEContainer1.PInitInfo := TheInitInfo;
```

See also

PInitInfo property

OnActivate event

Applies to

TApplication, TForm, TOLEContainer components

For forms

Declaration

```
property OnActivate: TNotifyEvent;
```

The *OnActivate* event for a form occurs when the form becomes active. A form becomes active when focus is transferred to it (when the user clicks on the form, for example).

For MDI child windows (forms with *FormStyle* property values of *fsMDIChild*), *OnActivate* occurs only when focus is shifted from one child to another. If focus is shifted from a non-MDI child window to an MDI child, the *OnActivate* event occurs for the MDI parent form.

Note The *OnActivate* event of the application (*TApplication*), not the form, occurs when Windows switches control from another application to your application.

Example

The following code adds the caption of *Form2* to a list box in *Form1* when a *Form2* is activated. To refer to *Form1* in *Form2*'s *OnActivate* event handler you must include the name of the unit in which *Form1* is declared to a **uses** clause in *Form2*'s unit. To avoid a circular unit reference (if *Form2* is already referenced by *Form1*'s **uses** clause), put the new **uses** clause in the **implementation** section of *Form2*'s unit.

```
procedure TForm2.FormActivate(Sender: TObject);
begin
  Form1.ListBox1.Items.Add(Screen.ActiveForm.Caption);
end;
```

See also

ActiveForm property, *Show* method, *ShowModal* method

For OLE containers

Declaration

```
property OnActivate: TNotifyEvent;
```

The *OnActivate* event for an OLE container occurs when the OLE object is activated as specified by the *AutoActivate* property.

Example

The following code displays the number of times an OLE container has been activated in *Label1*. The code assumes that *TimesActivated* is an *Integer* field of *Form1* that is initialized to 0 in the *OnCreate* event of *Form1*.

```
procedure TForm1.OleContainer1Activate(Sender: TObject);
begin
  TimesActivated := TimesActivated + 1;
  Form1.Label1.Caption := 'Times activated: '+IntToStr(TimesActivated);
end;
```

For an application

Declaration

```
property OnActivate: TNotifyEvent;
```

The *OnActivate* event for an application occurs when the application becomes active. Your application becomes active when it is initially run or when focus is shifted from another Windows application to your application.

Note Search Help for “Handling Application Events” for more information about creating event handlers for application events.

Example

The following code is the entire unit which assigns the *ApplicationActivate* procedure to the *OnActivate* event of the application. Note that *ApplicationActivate* is declared as a method of *Form1*. The code that you add is notes with comments. The rest of the code is generated by Delphi.

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    procedure ApplicationActivate(Sender: TObject);   {Add this declaration line}
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.DFM}

procedure TForm1.ApplicationActivate(Sender: TObject); {Write this procedure}
begin
  {Put code for your Application.OnActivate here}
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnActivate := ApplicationActivate;       {Write this line of code}
end;

end.
```

See also
Application variable, *OnDeactivate* event

OnActiveControlChange event

Applies to
TScreen component

Declaration

```
property OnActiveControlChange: TNotifyEvent;
```

The *OnActiveControlChange* event occurs when the focus on the screen shifts from one control to another. This change in focus means that a new control is now the value of the *ActiveControl* property of the screen. Use the *OnActiveControlChange* event to specify special processing you want to occur just before the new control becomes the active control.

Example
This example uses an edit box and a memo on a form. When the user switches the focus between the two controls, the control that currently has the focus becomes red:

```
type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Memo1: TMemo;
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    procedure ColorControl(Sender: TObject);
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.DFM}

procedure TForm1.ColorControl(Sender: TObject);
begin
  if Edit1.Focused then
    Edit1.Color := clRed
  else
    Edit1.Color := clWindow;
  if Memo1.Focused then
    Memo1.Color := clRed
  else
    Memo1.Color := clWindow;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Screen.OnActiveControlChange := ColorControl;
end;
```

See also

ActiveControl property, *ActiveForm* property, *OnActiveFormChange* event

OnActiveFormChange event

Applies to

TScreen component

Declaration

```
property OnActiveFormChange: TNotifyEvent;
```

The *OnActiveFormChange* event occurs when a new form becomes the active form on the screen (the form becomes the value of the *ActiveForm* property). Use the *OnActiveFormChange* event to specify any special processing to occur just before a new form becomes the active form.

Example

This example uses a two forms with a button on the first form. When the user clicks the button, the second form appears. As the user switches between forms, the form that is active is colored aqua:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    procedure ColorForm(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

uses Unit2;

procedure TForm1.ColorForm(Sender: TObject);
begin
  Color := clBtnFace;
  Form2.Color := clBtnFace;
  Screen.ActiveForm.Color := clAqua;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Screen.OnActiveFormChange := ColorForm;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Show;
end;
```

See also
ActiveForm property, *OnActiveControlChange* event

OnApply event

Applies to
TFontDialog component

Declaration

```
property OnApply: TFDApplyEvent
```

The *OnApply* event occurs when the user clicks the Apply button in the Font dialog box. The Apply button won't appear in the Font dialog box unless the form has an *OnApply* event handler. The user can use the Apply button to apply the font selected in the dialog box to a component immediately before the dialog box is closed.

Example
This code displays the Font dialog box and puts an Apply button in it. When the user clicks the Apply button, the font selected in the dialog box is applied to the *Button1* button while the dialog box is still open.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  FontDialog1.Execute;
```

```
end;

procedure TForm1.FontDialog1Apply(Sender: TObject; Wnd: Word);
begin
  Button1.Font := FontDialog1.Font;
end;
```

See also

Execute method, *Font* property

OnCalcFields event

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
property OnCalcFields: TDataSetNotifyEvent;
```

The *OnCalcFields* event is used to set the values of calculated fields. *OnCalcFields* is called when a dataset reads a record from the database. In addition, if the dataset's *AutoCalcFields* property is *True*, *OnCalcFields* is called when a non-calculated field is modified while the dataset is in Edit or Insert state.

Typically, the *OnCalcFields* event will be called often, so it should be kept short. *OnCalcFields* should not perform any actions that modify the dataset (or the linked dataset if it is part of a master-detail relationship), because this can lead to recursion.

While the *OnCalcFields* event is executed, a dataset will be put in *CalcFields* state. When a dataset is in *CalcFields* state, you cannot set the values of any fields other than calculated fields. After *OnCalcFields* is completed, the dataset will return to its previous state.

The first call to the *OnCalcFields* event handler may occur before all components in your application have been initialized. If your handler requires access to another component, use the Edit | Creation Order command to ensure that the components are created in the correct order.

OnChange event

Applies to

TBitmap, *TBrush*, *TCanvas*, *TFont*, *TGraphic*, *TGraphicsObject*, *TMetafile*, *TPen*, *TPicture*, *TStringList* objects; *TComboBox*, *TDBComboBox*, *TDBEdit*, *TDBLookupCombo*, *TDBMemo*, *TDBRadioGroup*, *TDDEClientItem*, *TDDEServerItem*, *TDirectoryListBox*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TFilterComboBox*, *TMaskEdit*, *TMemo*, *TQuery*, *TScrollBar*, *TTable*, *TTabSet*, *TBCDField*, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

For tab set controls

Declaration

```
property OnChange: TTabChangeEvent;
```

The *OnChange* event occurs just before a new tab is selected (the *TabIndex* value is about to change). To prevent the *TabIndex* value from changing, you need to write code in the *OnChange* event handler to stop it from doing so.

Example

The following code uses a check box and a tab set control on a form. If the check box is checked, another tab on the tab set can't be selected.

```
procedure TForm1.TabSet1Change(Sender: TObject; NewTab: Integer;
  var AllowChange: Boolean);
begin
  AllowChange := not CheckBox1.Checked;
end;
```

For DDE client item and DDE server item controls

Declaration

```
property OnChange: TNotifyEvent;
```

An *OnChange* event occurs when the value of the *Value* property of a DDE client item or DDE server item component changes.

If the value changed is that of a DDE client item component, the DDE server application continuously updates the *Value* property of the DDE client item component.

If the value changed is that of a DDE server item component, your application can change the *Value* property of the DDE server item component by assigning a new value to it. The DDE client can change *Value* by poking data (transferring data from the DDE client to the DDE server). See the documentation of the DDE client application for information about how data is poked. Delphi DDE client applications poke data using the *PokeData* method.

Example

The following code updates the contents of an edit box with the linked text from a DDE server when the data is updated.

```
procedure TForm1.DdeClientItem1Change(Sender: TObject);
begin
  Edit1.Text := DDEClientItem1.Text;
end;
```

For data-aware components

Declaration

```
property OnChange: TNotifyEvent;
```

The *OnChange* event for data-aware controls occurs when the contents of the field the control is accessing changes. Specify any special processing you want to occur at that time in the *OnChange* event handler

Example

The following code displays a message if the data accessed by *DBMemo1* changes.

```
procedure TForm1.DBMemo1Change(Sender: TObject);
begin
  MessageDlg('Data has changed',mtInformation,[mbOK],0);
end;
```

For fields

Declaration

```
property OnChange: TFieldNotifyEvent;
```

OnChange is activated when the contents of the field are modified. If a data-aware control is linked to the field, *OnChange* is not activated until the control attempts to store the changes into the current record.

You can take any special actions required by the event by assigning a method to this property.

Example

```
Field1.OnChange := CapitalizeFirstLetter;
```

For other components and objects

Declaration

```
property OnChange: TNotifyEvent;
```

The *OnChange* event specifies which event handler should execute when the contents of a component or object changes.

For graphics objects, *OnChange* occurs when the specific graphics item encapsulated by the object changes. For example, the *OnChange* event for a pen occurs when the *Color*, *Mode*, *Style*, or *Width* properties of the *TPen* object are modified.

For components, *OnChange* occurs when the main value or values of the component are modified. For example, *OnChange* occurs when the *Text* property of an edit box is modified.

For combo boxes, the *OnChange* event also occurs when an item is selected in the drop down list.

For string list objects, the *OnChange* event occurs when a change to a string stored in the list of strings changes.

Example

This example uses a color grid on a form. The color grid is a component on the Samples page of the Component palette. When the user clicks a color rectangle or drags the mouse cursor across the color grid, the color of the form changes.

```
procedure TForm1.ColorGrid1Change(Sender: TObject);
begin
  Color := ColorGrid1.ForegroundColor;
end;
```

See also

OnChanging event

OnChanging event

Applies to

TCanvas object

Declaration

```
property OnChanging: TNotifyEvent;
```

An *OnChanging* event occurs immediately before the graphic contained in the canvas is modified.

OnClick event

Applies to

TBitBtn, *TButton*, *TCheckBox*, *TComboBox*, *TDBCheckBox*, *TDBComboBox*, *TDBEdit*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBLookupList*, *TDBMemo*, *TDBNavigator*, *TDBRadioGroup*, *TDBText*, *TDirectoryListBox*, *TDrawGrid*, *TDriveComboBox*, *TFileListBox*, *TFilterComboBox*, *TForm*, *TGroupBox*, *TImage*, *TLabel*, *TListBox*, *TMaskEdit*, *TMediaPlayer*, *TMemo*, *TMenuItem*, *TNotebook*, *TOutline*, *TPaintBox*, *TPanel*, *TRadioButton*, *TScrollBar*, *TScrollBox*, *TSpeedButton*, *TStringGrid*, *TTabSet* components

For the media player components

Declaration

```
property OnClick: EMPNotify;
```

An *OnClick* event occurs when the user presses and releases the mouse button when the mouse pointer is over one of the control buttons of the media player control, or when the user presses *Spacebar* when the media player control has focus. When the media player control has focus, the user can select which control button to click when the *Spacebar* is pressed with the *Left Arrow* or *Right Arrow* keys.

Example

This example uses a label and a media player on a form. When the user clicks one of the media player buttons, the caption of the label indicates which button was clicked. For this example to run successfully, you must have a CD audio device installed correctly.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MediaPlayer1.DeviceType := dtCDAudio;
  MediaPlayer1.Open;
  MediaPlayer1.Left := 20;
  MediaPlayer1.Top := 12;
  Label1.Top := 44;
  Label1.Left := 20;
  Label1.Color := clYellow;
  Label1.Font.Name := 'Arial';
  Label1.Caption := 'Click Me';
end;

procedure TForm1.MediaPlayer1Click(Sender: TObject; Button: TMPBtnType;
  var DoDefault: Boolean);
begin
  case Button of
    btPlay :
    begin
      Label1.Caption := 'Playing';
      Label1.Left := 20;
    end;
    btPause:
    begin
      Label1.Caption := 'Paused';
      Label1.Left := 48;
    end;
    btStop:
    begin
      Label1.Caption := 'Stopped';
      Label1.Left := 76;
    end;
    btNext:
    begin
      Label1.Caption := 'Next';
      Label1.Left := 104;
    end;
    btPrev:
    begin
      Label1.Caption := 'Previous';
      Label1.Left := 132;
    end;
    btEject:
    begin
      Label1.Caption := 'Eject';
      Label1.Left := 244;
    end;
  end;
end;
```

See also

OnPostClick event

For database navigators

Declaration

```
property OnClick: ENavClick;
```

The *OnClick* event occurs when the user presses and releases the mouse button with the mouse pointer over one of the database navigator control buttons, or when the user presses *Spacebar* while the database navigator has focus. Calling the *Click* method also triggers *OnClick*.

Example

The following code determines which database navigator button was clicked and displays a message identifying the name of the button.

```
procedure TForm1.DBNavigator1Click(Sender: TObject; Button: TNavigateBtn);
var
  BtnName: string;
begin
  case Button of
    nbFirst  : BtnName := 'nbFirst';
    nbPrior  : BtnName := 'nbPrior';
    nbNext   : BtnName := 'nbNext';
    nbLast   : BtnName := 'nbLast';
    nbInsert : BtnName := 'nbInsert';
    nbDelete : BtnName := 'nbDelete';
    nbEdit   : BtnName := 'nbEdit';
    nbPost   : BtnName := 'nbPost';
    nbCancel : BtnName := 'nbCancel';
    nbRefresh: BtnName := 'nbRefresh';
  end;
  MessageDlg(BtnName + ' button clicked.', mtInformation, [mbOK], 0);
end;
```

For forms and other components

Declaration

```
property OnClick: TNotifyEvent;
```

The *OnClick* event occurs when the user clicks the component. Typically, this is when the user presses and releases the primary mouse button with the mouse pointer over the component. This event can also occur when

- The user selects an item in a grid, outline, list, or combo box by pressing an arrow key.
- The user presses *Spacebar* while a button or check box has focus.
- The user presses *Enter* when the active form has a default button (specified by the *Default* property).
- The user presses *Esc* when the active form has a cancel button (specified by the *Cancel* property).
- The user presses the accelerator key for a button or check box. For example, if the value of the *Caption* property of a check box is '&Bold', the B is underlined at run time and the *OnClick* event of the check box is triggered when the user presses *Alt+B*.
- The *Checked* property of a radio button is set to *True.*
- The value of the *Checked* property of a check box is changed.
- The *Click* method of a menu item is called.

The user presses the accelerator key for a button or check boxFor a form, an *OnClick* event occurs when the user clicks a blank area of the form or on a disabled component.

Example

The form in this example changes color each time the user clicks it:

```
procedure TForm1.FormClick(Sender: TObject);
begin
  Randomize;
  Color := Random(65535);
end;
```

See also

Click method, *OnDblClick* event

OnClose event

Applies to

TDDEClientConv, *TForm* components

For forms

Declaration

```
property OnClose: TCloseEvent;
```

The *OnClose* event specifies which event handler to call when a form is about to close. The handler specified by *OnClose* might, for example, test to make sure all fields in a data-entry form have valid contents before allowing the form to close.

A form is closed by the *Close* method or when the user chooses Close from the form's system menu.

The *TCloseEvent* type of *OnClose* has an *Action* parameter. The value of *Action* determines whether the form can actually close. These are the possible values of *Action*:

Value	Meaning
caNone	The form is not allowed to close, so nothing happens.
caHide	The form is not closed, but just hidden. Your application can still access a hidden form.
caFree	The form is closed and all allocated memory for the form is freed.
caMinimize	The form is minimized, rather than closed. This is the default action for MDI child forms.

Example

This example displays a message dialog box when the user attempts to close the form. If the user clicks the Yes button, the form closes; otherwise, the form remains open.

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  if MessageDlg('Close application ?', mtConfirmation,
    [mbYes, mbNo], 0) = mrYes then
    Action := caFree
  else
    Action := caNone;
end;
```

See also

OnCloseQuery event, *OnOpen* event

For DDE components

Declaration

```
property OnClose: TNotifyEvent;
```

An *OnClose* event occurs when a DDE conversation is terminated. A conversation is terminated when one of the applications involved is closed, or when the *CloseLink* method is called.

Example

The following code displays a message when a conversation is closed.

```
procedure TForm1.DdeClientConv1Close(Sender: TObject);
begin
  MessageDlg('This conversation is finished!', mtInformation, [mbOK],0);
end;
```

OnCloseQuery event

Applies to

TForm component

Declaration

```
property OnCloseQuery: TCloseQueryEvent;
```

The *OnCloseQuery* event occurs when an action to close the form takes place (when the *Close* method is called or when the user chooses Close from the form's System menu). An *OnCloseQuery* event handler contains a *Boolean CanClose* variable that determines whether a form is allowed to close. It's default value is *True*. See the *TCloseQueryEvent* type for more information about *CanClose.*

You can use an *OnCloseQuery* event handler to ask users if they are sure they really want the form closed immediately. For example, you can use the handler to display a message box that prompts the user to save a file before closing the form.

Example

When the user attempts to close the form in this example, a message dialog appears that asks the user if it is OK to close the form. If the user chooses the OK button, the form closes. If the user chooses Cancel, the form doesn't close.

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  if MessageDlg('Close the form?', mtConfirmation,
    [mbOk, mbCancel], 0) = mrCancel then
     CanClose := False;
end;
```

See also

Close method, *OnClose* event

OnColEnter event

Applies to

TDBGrid component

Declaration

```
property OnColEnter: TNotifyEvent;
```

The *OnColEnter* event occurs when the user clicks a cell in a column or moves to a column with the Tab key within the data grid. Use the *OnColEnter* event to specify any processing you want to occur as soon as a column is entered.

Example

The following code concatenates an asterisk to the display label of a field when the column is entered.

```
procedure TForm1.DBGrid1ColEnter(Sender: TObject);
begin
  with DBGrid1.SelectedField do
    DisplayLabel := '* ' + DisplayLabel;
end;
```

See also

OnColExit event

OnColExit event

Applies to

TDBGrid component

Declaration

```
property OnColExit: TNotifyEvent;
```

The *OnColExit* event occurs when the user uses the Tab key to move out of a column or clicks a cell in another column. Use the *OnColExit* event to specify any special processing you want to occur when exiting the column.

Example

The following code deletes the first two characters from the display label of the selected field when exiting a column. Note that *FirstTime* is a *Boolean* field that prevents characters from being deleted the first time a column is exited. Use this code in conjunction with code in the example of *OnColEnter* to modify the appearance of the display label of columns while they are entered.

```
procedure TForm1.DBGrid1ColExit(Sender: TObject);
var
  TheLabel: string;
begin
  if FirstTime then
    FirstTime := False
  else
  begin
    with DBGrid1.SelectedField do
    begin
      TheLabel := DisplayLabel;
      Delete(TheLabel, 1, 2);
```

```
        DisplayLabel := TheLabel;
      end;
    end;
  end;
```

See also
OnColExit event

OnCollapse event

Applies to
TOutline component

Declaration

```
property OnCollapse: EOutlineChange;
```

An *OnCollapse* event occurs when an expanded outline item that has subitems is collapsed. An expanded outline item is collapsed when the user double-clicks it at run time, when the *FullCollapse* method is called, or when its *Expanded* property is set to *False*. When collapsed, the subitems no longer appear in the outline and the plus picture or closed picture for the parent item is displayed if the appropriate *OutlineStyle* has been selected.

Example
The following code displays the text from a collapsed outline item in a message dialog box.

```
procedure TForm1.Outline1Collapse(Sender: TObject; Index: Longint);
var
  TheStr: string;
begin
  TheStr :=  Outline1.Items[Index].Text;
  MessageDlg(TheStr+' has collapsed.', mtInformation, [mbOK],0);
end;
```

See also
OnExpand event, *PictureClosed* property, *PicturePlus* property

OnColumnMoved event

Applies to
TDrawGrid, *TStringGrid* components

Declaration

```
property OnColumnMoved: TMovedEvent;
```

The *OnColumnMoved* event occurs when the user moves a column using the mouse. The user can move a column only if the *Options* property set includes the value *goColMoving*.

Example

The following code permits one column to be moved (assuming [*goColMoving*] is specified for the *Options* property at design time), then locks the columns by preventing any more moves.

```
procedure TForm1.StringGrid1ColumnMoved(Sender: TObject; FromIndex, ToIndex: Longint);
begin
  StringGrid1.Options := StringGrid1.Options - [goColMoving];
end;
```

See also

OnRowMoved event

OnCreate event

Applies to

TForm component

Declaration

```
property OnCreate: TNotifyEvent;
```

The *OnCreate* event specifies which event handler to call when the form is first created. You can write code in the event handler that sets initial values for properties and does any processing you want to occur before the user begins interacting with the form.

Delphi creates a form when the application is run by calling the *Create* method.

Note When writing code in an *OnCreate* event handler, don't fully qualify a component reference by including the name of the form in the reference. For example, if the form is named *Form1* and contains an *Edit1* edit box control, don't refer to the edit box control with the *Form1.Edit1* name. Because *Form1* doesn't yet exist when this code executes, your application would crash if you used the fully qualified name. Instead, simply use the name *Edit1*.

When a form is being created and its *Visible* property is *True*, the following events occur in the order listed:

- *OnActivate*
- *OnShow*
- *OnCreate*
- *OnPaint*

Example

This very simple *OnCreate* event handler assures that the form is the same color as the Windows system color of your application workspace:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Color := clAppWorkSpace;
end;
```

Note The *Color* property in this example is not prefaced with the name of the form. If you write the statement like this,

```
Form1.Color := clAppWorkSpace;
```

the application won't run without error, because *Form1* does not yet exist at the time this code is executed.

See also

OnActivate event, *OnDestroy* event, *OnPaint* event

OnDataChange event

Applies to

TDataSource component

Declaration

```
property OnDataChange: TDataChangeEvent;
```

The *OnDataChange* occurs when the *State* property changes from *dsInactive*, or when a data-aware control notifies the *TDataSource* that something has changed.

Notification occurs when the following items change because of field modification or scrolling to a new record: field component, record, dataset component, content, and layout. The *Field* parameter to the method may be **nil** if more than one of the fields changed simultaneously (as in a move to a different record). Otherwise, *Field* is the field which changed.

See also

OnStateChange event, *State* property

OnDblClick event

Applies to

TComboBox, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBLookupList*, *TDBMemo*, *TDBNavigator*, *TDBText*, *TDirectoryListBox*, *TDrawGrid*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TFilterComboBox*, *TForm*, *TGroupBox*, *TImage*, *TLabel*, *TListBox*, *TMaskEdit*, *TMemo*, *TNotebook*, *TOLEContainer*, *TOutline*, *TPaintBox*, *TPanel*, *TRadioButton*, *TScrollBox*, *TSpeedButton*, *TStringGrid* components

Declaration

```
property OnDblClick: TNotifyEvent;
```

The *OnDblClick* event occurs when the user double-clicks the mouse button while the mouse pointer is over the component.

Example

This example notifies the user that the form was double-clicked.

```
procedure TForm1.FormClick(Sender: TObject);
begin
  MessageDlg('You double-clicked the form', mtInformation, [mbOk], 0);
end;
```

See also

OnClick event

OnDeactivate event

Applies to

TApplication component

Declaration

```
property OnDeactivate: TNotifyEvent;
```

The *OnDeactivate* event occurs when the user switches from your application to another Windows application. Use the *OnDeactive* event to do any special processing you want to occur before your application is deactivated.

Note Search Help for "Handling Application Events" for more information about creating event handlers for application events.

Example

The following code minimizes an application when it's deactivated. Note that *AppDeactivate* should be declared a method of *TForm1*.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnDeactivate := AppDeactivate;
end;

procedure TForm1.AppDeactivate(Sender: TObject);
begin
  Application.Minimize;
end;
```

See also

OnActivate event

OnDestroy event

Applies to
TForm component

Declaration

```
property OnDestroy: TNotifyEvent;
```

The *OnDestroy* event occurs when a form is about to be destroyed. A form is destroyed by the *Destroy*, *Free*, or *Release* methods, or when the main form of the application is closed.

Example
The following code explicitly allocates memory for a a pointer in the *OnCreate* event of *Form1,* then releases the memory in the *OnDestroy* event. Assume that *MyPtr* is a Pointer type field of *TForm1*.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  New(MyPtr);
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  Dispose(MyPtr);
end;
```

See also
OnCreate event

OnDragDrop event

Applies to
TBitBtn, *TButton*, *TCheckBox*, *TComboBox*, *TDBCheckBox*, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBLookupList*, *TDBMemo*, *TDBNavigator*, *TDBText*, *TDBRadioGroup*, *TDirectoryListBox*, *TDrawGrid*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TFilterComboBox*, *TForm*, *TGroupBox*, *TImage*, *TListBox*, *TMaskEdit*, *TMemo*, *TNotebook*, *TOLEContainer*, *TOutline*, *TPaintBox*, *TPanel*, *TRadioButton*, *TScrollBar*, *TScrollBox*, *TShape*, *TStringGrid*, *TTabSet* components

Declaration

```
property OnDragDrop: TDragDropEvent;
```

The *OnDragDrop* event occurs when the user drops an object being dragged. Use the *OnDragDrop* event handler to specify what you want to happen when the user drops an object. The *Source* parameter of the *OnDragDrop* event is the object being dropped, and the *Sender* is the control the object is being dropped on. The *X* and *Y* parameters are the coordinates of the mouse positioned over the control.

Example

This code comes from an application that contains a list box and three labels, each with a different font and color. The user can select a label and drag it to a list box and drop it. When the label is dropped, the items in the list box assume the color and font of the dropped label. This is the *OnDragDrop* event handler.

```
procedure TForm1.ListBox1DragDrop(Sender, Source: TObject; X, Y: Integer);
begin
  if (Sender is TListBox) and (Source is TLabel) then
  begin
    (Sender as TListBox).Font := (Source as TLabel).Font;
  end;
end;
```

The *Source* in this example is the label, and the *Sender* is the list box.

See also

DragCursor property, *DragMode* property, *OnDragOver* event, *OnEndDrag* event

OnDragOver event

Applies to

TBitBtn, *TButton*, *TCheckBox*, *TComboBox*, *TDBCheckBox*, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBLookupList*, *TDBMemo*, *TDBNavigator*, *TDBText*, *TDBRadioGroup*, *TDirectoryListBox*, *TDrawGrid*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TFilterComboBox*, *TForm*, *TGroupBox*, *TImage*, *TListBox*, *TMaskEdit*, *TMemo*, *TNotebook*, *TOLEContainer*, *TOutline*, *TPaintBox*, *TPanel*, *TRadioButton*, *TScrollBar*, *TScrollBox*, *TShape*, *TStringGrid*, *TTabSet* components

Declaration

```
property OnDragOver: TDragOverEvent;
```

The *OnDragOver* event occurs when the user drags an object over a component. Usually you'll use an *OnDragOver* event to accept an object so the user can drop it.

The *OnDragOver* event accepts an object when its *Accept* parameter is *True*.

Usually, you will want the cursor to change shape, indicating that the control can accept the dragged object if the user drops it. You can change the shape of the cursor by changing the value of the *DragCursor* property for the control at either design or run time before an *OnDragOver* event occurs.

Example

This *OnDragOver* event handler permits the list box to accept a dropped label:

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

The *Source* parameter identifies what is being dragged. The *Sender* is the control being dragged over.

This code permits the list box to accept any dropped control:

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

See also

DragMode property, *OnDragDrop* event, *OnEndDrag* event, *TDragState* type

OnDrawCell event

Applies to

TDrawGrid, *TStringGrid* components

Declaration

```
property OnDrawCell: TDrawCellEvent;
```

The *OnDrawCell* event occurs whenever the contents of a grid cell need to be redisplayed. For example, it occurs when the user selects a cell or scrolls the grid. How a cell is redrawn depends on the value of the *DefaultDrawing* property.

If *DefaultDrawing* is *False*, you must write the code that handles all drawing within the cell in the *OnDrawCell* event handler.

Example

The following code draws a focus rectangle around each of the cells of *StringGrid1*.

```
procedure TForm1.StringGrid1DrawCell(Sender: TObject; Col, Row: Longint;
  Rect: TRect; State: TGridDrawState);
begin
  StringGrid1.Canvas.DrawFocusRect(Rect);
end;
```

OnDrawDataCell event

Applies to

TDBGrid component

Declaration

```
property OnDrawDataCell: TDrawDataCellEvent;
```

The *OnDrawDataCell* event occurs whenever the contents of a data grid cell need to be redisplayed. For example, it occurs when the user selects a cell or scrolls the grid. How a cell is redrawn depends on the value of the *DefaultDrawing* property.

If *DefaultDrawing* is *False*, you must write the code that handles all the drawing within the cell in the *OnDrawDataCell* event handler.

Example

The following code fills the cells of *DBGrid1* with the pattern defined by the *Brush* of the *Canvas* of *DBGrid1*.

```
procedure TForm1.DBGrid1.DrawDataCell(Sender: TObject; Rect: TRect; Field: TField;
  State: TGridDrawState)
begin
  DBGrid1.Canvas.FillRect(Rect);
end;
```

OnDrawItem event

Applies to

TComboBox, *TDBComboBox*, *TDBListBox*, *TListBox*, *TOutline* components

Declaration

```
property OnDrawItem: TDrawItemEvent;
```

The *OnDrawItem* event occurs whenever an item in an owner-draw outline, list box, or combo box needs to be redisplayed. For example, it occurs when the user selects an item or scrolls the outline, list box, or combo box. *OnDrawItem* events occur only for outlines with the *Style* value *osOwnerDraw*, list boxes with the *Style* values *lbOwnerDrawFixed* or *lbOwnerDrawVariable,* and for combo boxes with the *Style* values *csOwnerDrawFixed* or *csOwnerDrawVariable.*

OnDrawItem passes four parameters to its handler describing the item to be drawn:

- a reference to the control containing the item
- the index of the item in that control
- a rectangle in which to draw
- the state of the item (selected, focused, and so on)

The size of the rectangle that contains the item is determined either by the *ItemHeight* property for fixed owner-draw controls or by the response to the *OnMeasureItem* event for variable owner-draw controls.

Example

Here is a typical handler for an *OnDrawItem* event. In the example, a list box with the *lbOwnerDrawFixed* style draws a bitmap to the left of each string.

```
procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index: Integer; Rect: TRect;
  State: TOwnerDrawState);
```

```
var
  Bitmap: TBitmap;                                { temporary variable for the item's bitmap }
  Offset: Integer;                                                       { text offset width }
begin
  with (Control as TListBox).Canvas do      { draw on the control canvas, not on the form }
  begin
    FillRect(Rect);                                                     { clear the rectangle }
    Offset := 2;                                                    { provide default offset }
    Bitmap := TBitmap(Items.Objects[Index]);                   { get the bitmap for this item }
    if Bitmap <> nil then
    begin
      BrushCopy(Bounds(Rect.Left + 2, Rect.Top, Bitmap.Width, Bitmap.Height), Bitmap,
        Bounds(0, 0, Bitmap.Width, Bitmap.Height), clRed);                { render the bitmap }
      Offset := Bitmap.width + 6;              { add four pixels between bitmap and text }
    end;
    TextOut(Rect.Left + Offset, Rect.Top, Items[Index])                    { display the text }
  end;
end;
```

Note that the *Rect* parameter automatically provides the proper location of the item within the control's canvas.

See also

ItemHeight property, *OnMeasureItem* event

OnDrawTab event

Applies to

TTabSet component

Declaration

```
property OnDrawTab: TDrawTabEvent;
```

The *OnDrawTab* event occurs when a tab needs to redisplay only for tab set controls that have the *Style* property value of *tsOwnerDraw.* For example, it happens when the user selects a tab or scrolls the tabs using an owner-draw tab set control.

You must write the code in the *OnDrawTab* event handler to draw the tab.

OnDrawTab occurs just after the *OnMeasureTab* event, which contains the code to calculate the width of the tab needed. The height of the tab is determined by the value of the *TabHeight* property of the tab set control. The code you write in the *OnDrawTab* event handler, therefore, must use the width determined with the *OnMeasureTab* event to draw the tab.

Example

The following code loads a bitmap from the *Objects* property of the *Tabs* list of the *DriveTabSet* tab set component. This bitmap is then drawn on the tab, along with the text from the *Tabs* list.

```
procedure TFMForm.DriveTabSetDrawTab(Sender: TObject; TabCanvas: TCanvas;
  R: TRect; Index: Integer; Selected: Boolean);
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap(DriveTabSet.Tabs.Objects[Index]);
  with TabCanvas do
  begin
    Draw(R.Left, R.Top + 4, Bitmap);
    TextOut(R.Left + 2 + Bitmap.Width, R.Top + 2, DriveTabSet.Tabs[Index]);
  end;
end;
```

See also

OnMeasureTab event, *TabHeight* property

OnDropDown event

Applies to

TComboBox, *TDBListBox*, *TDBComboBox*, *TDBLookupCombo*, *TListBox* components

Declaration

```
property OnDropDown: TNotifyEvent;
```

The *OnDropDown* event occurs when the user opens (drops down) a combo box or list box.

Example

The following code doesn't sort the items in a combo box until the user opens it.

```
procedure TForm1.ComboBox1DropDown(Sender: TObject);
begin
  ComboBox1.Sorted := True;
end;
```

OnEndDrag event

Applies to

TBitBtn, *TButton*, *TCheckBox*, *TComboBox*, *TDBCheckBox*, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBMemo*, *TDBNavigator*, *TDBText*, *TDBRadioGroup*, *TDirectoryListBox*, *TDrawGrid*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TFilterComboBox*, *TGroupBox*, *TImage*, *TListBox*, *TMaskEdit*, *TMemo*, *TNotebook*, *TOLEContainer*, *TOutline*, *TPanel*, *TRadioButton*, *TScrollBar*, *TScrollBox*, *TShape*, *TStringGrid*, *TTabSet* components

Declaration

```
property OnEndDrag: TEndDragEvent;
```

The *OnEndDrag* event occurs whenever the dragging of an object ends, either by dropping the object or by canceling the dragging. Use the *OnEndDrag* event handler to specify any special processing you want to occur when dragging stops. If the dragged object was dropped and accepted by the control, the *Target* parameter of the *OnEndDrag* event is *True*. If the object was not dropped successfully, the value of *Target* is **nil**.

Example

This code displays a message in a label named *Status*. The message displayed depends on whether or not the dragged label control was dropped into and accepted by a list box control successfully:

```
procedure TForm1.LabelEndDrag(Sender, Target: TObject; X, Y: Integer);
var
  S: string;
begin
  S := (Sender as TLabel).Name + ' was dropped... and ';
  if Target <> nil then S := S + 'accepted!'
  else S := S + 'rejected!';
  Status.Caption := S;
end;
```

The *Target* parameter is the list box and *Sender* is the label.

See also

EndDrag method, *OnDragDrop* event

OnEnter event

Applies to

All windowed controls

Declaration

```
property OnEnter: TNotifyEvent;
```

The *OnEnter* event occurs when a component becomes active. Use the *OnEnter* event handler to specify any special processing you want to occur when a component becomes active.

Note The *OnEnter* event does not occur when switching between forms or between another Windows application and your application.

Note The *OnEnter* event for a *TPanel* or *THeader* component never occurs as panels and headers never receive focus.

Example
This example uses an edit box and a memo control on a form. When either the edit box or the memo is the active control, it is colored yellow. When the active control becomes inactive, the color of the control returns to the Windows system color for a window.

```
procedure TForm1.Edit1Enter(Sender: TObject);
begin
  Edit1.Color := clYellow;
end;

procedure TForm1.Edit1Exit(Sender: TObject);
begin
  Edit1.Color := clWindow;
end;

procedure TForm1.Memo1Enter(Sender: TObject);
begin
  Memo1.Color := clYellow;
end;

procedure TForm1.Memo1Exit(Sender: TObject);
begin
  Memo1.Color := clWindow;
end;
```

See also
ActiveControl property, *OnActivate* event, *OnExit* event

OnException event

Applies to
TApplication component

Declaration

```
property OnException: TExceptionEvent;
```

The *OnException* event occurs when an unhandled exception occurs in your application. By default, the *HandleException* method calls the *OnException* event handler, which calls *ShowException* to display a message dialog box appears indicating an error occurred. You can change this behavior by specifying what processing you want to occur in the *OnException* event handler.

Note Search Help for “Handling Application Events” for more information about creating event handlers for application events.

Example
The following code defines the default exception handling of the application, assuming *AppException* is declared a method of *TForm1*.

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

```
    Application.OnException := AppException;
  end;
  procedure TForm1.AppException(Sender: TObject; E: Exception);
  begin
    Application.ShowException
  end;
```

OnExecuteMacro event

Applies to

TDDEServerConv component

Declaration

```
property OnExecuteMacro : TMacroEvent;
```

The *OnExecuteMacro* event occurs when a DDE client application sends a macro to a DDE server conversation component. Write code to process the macro in the *OnExecuteMacro* event handler. See the DDE client application documentation for information about how it sends macros. If the DDE client is a Delphi application, a macro is sent with the *ExecuteMacro* method of the *TDDEClientConv* component.

Example

The following code clears the contents of a memo in the server application if the appropriate message is sent from the client application.

```
  procedure TForm1.DdeServerConv1ExecuteMacro(Sender: TObject; Msg: TStrings);
  begin
    if Msg.Strings[0] = 'Edit|Clear' then
      Memo1.Clear;
  end;
```

OnExit event

Applies to

All windowed controls

Declaration

```
property OnExit: TNotifyEvent;
```

The *OnExit* event occurs when the input focus shifts away from one control to another. Use the *OnExit* event handler when you want special processing to occur when this control ceases to be active.

Note The *OnExit* event does not occur when switching between forms or between another Windows application and your application.

Note The *OnExit* event for a *TPanel* or *THeader* component never occurs as panels and headers never receive focus.

Note The *ActiveControl* property is updated before an *OnExit* event occurs.

Example

This example uses an edit box and a memo control on a form. When either the edit box or the memo is the active control, it is colored yellow. When the active control becomes inactive, the color of the control returns to the Windows system color for a window.

```
procedure TForm1.Edit1Enter(Sender: TObject);
begin
  Edit1.Color := clYellow;
end;

procedure TForm1.Edit1Exit(Sender: TObject);
begin
  Edit1.Color := clWindow;
end;

procedure TForm1.Memo1Enter(Sender: TObject);
begin
  Memo1.Color := clYellow;
end;

procedure TForm1.Memo1Exit(Sender: TObject);
begin
  Memo1.Color := clWindow;
end;
```

See also

OnEnter event

OnExpand event

Applies to

TOutline component

Declaration

```
property OnExpand: EOutlineChange;
```

An *OnExpand* event occurs when a collapsed outline item having subitems is expanded. A collapsed outline item is expanded when the user double-clicks on it at run time, when its *Expanded* property is set to *True*, or when the *FullExpand* method of the *TOutlineNode* object is called. When expanded, the subitems appear in the outline and the minus picture or open picture for the parent item is displayed if the appropriate *OutlineStyle* has been selected.

Example

The following code displays the text from a collapsed outline item in a message dialog box.

```
procedure TForm1.Outline1Expand(Sender: TObject; Index: Longint);
```

```
var
  TheStr: string;
begin
  TheStr :=  Outline1.Items[Index].Text;
  MessageDlg(TheStr+' has expanded.', mtInformation, [mbOK],0);
end;
```

See also

OnCollapse event, *PictureMinus* property, *PictureOpen* property

OnFind event

Applies to

TFindDialog, *TReplaceDialog* components

Declaration

```
property OnFind: TNotifyEvent;
```

The *OnFind* event occurs whenever the user chooses the Find Next button in the Find or Replace dialog box. Use the *OnFind* event to specify what you want to happen when the user chooses the Find Next button.

Example

The following text compares the *FindText* to the *Text* of the *Items* of *Outline1*. If the string is found, a message is displayed and *I*-1 specifies the index of the matching item.

```
procedure TForm1.FindDialog1Find(Sender: TObject);
var
  I: Integer;
  Found: Boolean;
begin
  I := 1;
  Found := False;
  repeat
    if Outline1.Items[I].Text = FindDialog1.FindText then
    begin
      MessageDlg('Found!', mtInformation, [mbOK], 0);
      Found := True;
    end;
    I := I+1;
  until (I > Outline1.ItemCount) or (Found);
end;
```

See also

OnReplace event

OnGetEditMask event

Applies to
TDrawGrid, *TStringGrid* components

Declaration

```
property OnGetEditMask: TGetEditEvent;
```

The *OnGetEditMask* event occurs when the *Options* property set contains the value *goEditing* and the grid needs to redisplay the text of a cell in the grid using a specified edit mask. For example, the grid needs to redisplay the text when the user scrolls the grid or the user changes the data.

You write the code to specify the edit mask for the cell in the *OnGetEditMask* event handler.

Example
This example specifies an edit mask commonly used to display American telephone numbers for the cell in column 2, row 3 of the string grid:

```
procedure TForm1.StringGrid1GetEditMask(Sender: TObject; ACol,
  ARow: Longint; var Value: OpenString);
begin
  if ACol = 2 then
    if ARow = 3 then
      Value :=  '!\(999\)000-0000;1';
end;
```

See also
EditMask property, *OnGetEditText* event

OnGetEditText event

Applies to
TDrawGrid, *TStringGrid* components

Declaration

```
property OnGetEditText: TGetEditEvent;
```

The *OnGetEditText* event occurs when the *Options* property set contains the value *goEditing* and the grid needs to redisplay the text of a cell in the grid. For example, the grid needs to redisplay the text when the user scrolls the grid or the user changes the data.

You write the code to retrieve the text of the cell in the *OnGetEditText* event handler.

When the user edits data in a grid, the *OnSetEditText* event occurs to change the actual data, then the *OnGetEditText* event occurs to display the changed data in the grid.

Example

The following code appends 'My ' to any text entered in *StringGrid1*.

```
procedure TForm1.StringGrid1GetEditText(Sender: TObject; ACol,
  ARow: Longint; var Value: OpenString);
begin
  Value := 'My ' + Value;
end;
```

See also

OnGetEditMask event, *OnSetEditText* event

OnGetText event

Applies to

TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
property OnGetText: TFieldGetTextEvent;
```

The *OnGetText* event is activated when the *DisplayText* or *Text* properties are referenced. The *DisplayText* parameter indicates if the event should supply the text in display format or in edit format for the *Text* property. If *OnGetText* has been assigned a method, the default processing for *DisplayText* or *Text* does not occur; the event handler is expected to perform any conversion required to display the value.

By assigning a method to this property, you can take any special actions required by the event.

Example

```
Field1.OnGetText := MyFormatMethod;
```

OnHide event

Applies to

TForm component

Declaration

```
property OnHide: TNotifyEvent;
```

The *OnHide* event occurs just before the form is hidden on the screen. Use the *OnHide* event to specify any special processing you want to happen just before the form disappears.

A form that is an MDI child form (*FormStyle* is *fsMDIChild*) loses its window handle when it is hidden. If your application performs some operation that causes the window handle to come back, such as adding items to a list box on the form, an exception is raised.

Example

This example uses two forms, each with a label and a button. When the user clicks a button, the other form appears and the current form disappears. Also, a message appears in the label of the form that is showing, indicating that the other form is hidden. This is the implementation section of *Unit1*:

```
implementation

{$R *.DFM}

uses Unit2;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Show;
  Hide;
end;

procedure TForm1.FormHide(Sender: TObject);
begin
  Form2.Label1.Caption := 'Form1 is hiding';
end;

end.
```

This is the implementation section of *Unit2*:

```
implementation

{$R *.DFM}

uses Unit1;

procedure TForm2.Button1Click(Sender: TObject);
begin
  Form1.Show;
  Hide;
end;

procedure TForm2.FormHide(Sender: TObject);
begin
  Form1.Label1.Caption := 'Form2 is hiding';
end;

end.
```

See also

OnShow event

OnHelp event

Applies to

TApplication component

Declaration

```
property OnHelp: THelpEvent;
```

The *OnHelp* event occurs when your application receives a request for help. Use the *OnHelp* event handler to specify any special processing you want to occur when help is requested.

The *HelpContext* and the *HelpJump* methods automatically trigger the *OnHelp* event.

Note Search Help for "Handling Application Events" for more information about creating event handlers for application events.

Example

The following code changes the Help file for the application to the results of the Open dialog component. *AppHelp* should be assigned to the *OnHelp* event handler of *Application* in the *OnCreate* event of *Form1*.

```
function TForm1.AppHelp(Command: Word; Data: Longint): Boolean;
begin
  if OpenDialog1.Execute then
    Application.HelpFile := OpenDialog1.FileName;
end;
```

See also

HelpCommand method, *HelpContext* property, *HelpFile* property, *THelpEvent* type

OnHint event

Applies to

TApplication component

Declaration

```
property OnHint: TNotifyEvent;
```

The *OnHint* event occurs when the user positions the mouse pointer over a control with a *Hint* property value other than an empty string (''). Use the *OnHint* event handler to perform any special processing you want to happen when the *OnHint* event occurs.

A common use of the *OnHint* event is to display the value of a control or menu item's *Hint* property as the caption of a panel control (*TPanel*), thereby using the panel as a status bar. Using the *Hint* property, you can specify a Help Hint and a usually longer hint that appears elsewhere when the *OnHint* event occurs.

Note Search Help for “Handling Application Events” for more information about creating event handlers for application events.

Example

This example uses a panel component, a menu, and an edit box on a form. You can design the menu as you want, but remember to include a value for the *Hint* property for each menu item in the menu. Also, specify a value for the *Hint* property of the edit box. Align the panel at the bottom of the form (choose *alBottom* as the value of the *Align* property), and left justify the caption of the panel (choose *taLeftJustify* as the value of the *Alignment* property).

The *OnHint* event is an event of the *TApplication* component. You can’t use the Object Inspector to generate an empty event handler for *TApplication*, so you will need to write your own *OnHint* event handler. To accomplish this, you create a method of the *TForm1* object and give it an appropriate name, such as *DisplayHint*. You write the method in the **implementation** part of the unit, but you must also remember to declare the method in the *TForm1* type declaration in the **public** section.

In the *DisplayHint* method, you assign the *Hint* property of the application to the *Caption* property of the panel component.

One task remains. The *OnHint* event is an event of *TApplication*, so you must assign the new method you created as the method used by the *OnHint* event. You can do this in the form’s *OnCreate* event handler.

This code shows the complete type declaration, the new method, and the *OnCreate* event handler. When the user runs the application and positions the cursor over the edit box or a menu item on the menu, the specified hint appears as the caption of the panel at the bottom of the form:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    Panel1: TPanel;
    Edit1: TEdit;
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    procedure DisplayHint(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation

{$R *.FRM}

procedure TForm1.DisplayHint(Sender: TObject);
begin
  Panel1.Caption := Application.Hint;
end;
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnHint := DisplayHint;
end;
```

See also

OnCreate event

OnIdle event

Applies to

TApplication component

Declaration

```
property OnIdle: TIdleEvent
```

The *OnIdle* event occurs whenever the application is idle. Use the *OnIdle* event handler to specify any special processing to occur when your application is idle. Your application is idle when it is processing code, for example, or when it is waiting for input from the user.

The *TIdleEvent* type has a *Boolean* parameter *Done* that is *True* by default. When *Done* is *True*, the Windows API *WaitMessage* function is called when *OnIdle* returns. *WaitMessage* yields control to other applications until a new message appears in the message queue of your application. If *Done* is *False*, *WaitMessage* is not called.

Note Search Help for “Handling Application Events” for more information about creating event handlers for application events.

Example

The following code allows other applications to be processed while *Application* is idle. *AppIdle* should be declared as a method of *TForm1*.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnIdle := AppIdle;
end;

procedure TForm1.AppIdle(Sender: TObject; var Done: Boolean);
begin
  Done := True;
end;
```

OnKeyDown event

Applies to

TBitBtn, *TButton*, *TCheckBox*, *TComboBox*, *TDBCheckBox*, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBMemo*, *TDirectoryListBox*, *TDrawGrid*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TFilterComboBox*, *TForm*, *TListBox*, *TMaskEdit*, *TMemo*, *TOLEContainer*, *TOutline*, *TRadioButton*, *TScrollBar*, *TStringGrid* components

Declaration

```
property OnKeyDown: TKeyEvent;
```

The *OnKeyDown* event occurs when a user presses any key while the control has focus. Use the *OnKeyDown* event handler to specify special processing to occur when a key is pressed. The *OnKeyDown* handler can respond to all keyboard keys including function keys and keys combined with the *Shift*, *Alt*, and *Ctrl* keys and pressed mouse buttons. The *Key* parameter of the *OnKeyDown* event handler is of type *Word*; therefore, you must use virtual key codes to determine the key pressed. You can find a table of virtual key codes in the Help system; search for the topic Virtual Key Codes.

Example

This event handler displays a message dialog when the user presses *Alt+F10*:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if ((Shift = [ssAlt]) and (Key = VK_F10)) then
    MessageDlg('Alt+F10 pressed down', mtInformation, [mbOK], 0);
end;
```

See also

KeyPreview property, *OnKeyPress* event, *OnKeyUp* event

OnKeyPress event

Applies to

TBitBtn, *TButton*, *TCheckBox*, *TComboBox*, *TDBCheckBox*, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBMemo*, *TDirectoryListBox*, *TDrawGrid*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TFilterComboBox*, *TForm*, *TListBox*, *TMaskEdit*, *TMemo*, *TOLEContainer*, *TOutline*, *TRadioButton*, *TScrollBar*, *TStringGrid* components

Declaration

```
property OnKeyPress: TKeyPressEvent;
```

The *OnKeyPress* event occurs when a user presses a single character key. Use the *OnKeyPress* event handler when you want something to happen as a result of pressing a single key.

The *Key* parameter in the *OnKeyPress* event handler is of type *Char*; therefore, the *OnKeyPress* event registers the ASCII character of the key pressed. Keys that don't correspond to an ASCII *Char* value (*Shift* or *F1*, for example) don't generate an *OnKeyPress* event. Key combinations (such as *Shift+A*), generate only one *OnKeyPress* event (for this example, *Shift+A* results in a *Key* value of "A" if *Caps Lock* is off). If you want to respond to non-ASCII keys or key combinations, use the *OnKeyDown* or *OnKeyUp* event handlers.

Example

This event handler displays a message dialog box specifying which key was pressed:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  MessageDlg(Key + ' has been pressed', mtInformation, [mbOK], 0)
end;
```

See also

KeyPreview property

OnKeyUp event

Applies to

TBitBtn, *TButton*, *TCheckBox*, *TComboBox*, *TDBCheckBox*, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBMemo*, *TDirectoryListBox*, *TDrawGrid*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TFilterComboBox*, *TForm*, *TListBox*, *TMaskEdit*, *TMemo*, *TOLEContainer*, *TOutline*, *TRadioButton*, *TScrollBar*, *TStringGrid* components

Declaration

```
property OnKeyUp: TKeyEvent;
```

The *OnKeyUp* event occurs when the user releases a key that has been pressed. Use the *OnKeyUp* event handler when you want special processing to occur when a key is released. The *OnKeyUp* handler can respond to all keyboard keys including function keys and keys combined with the *Shift*, *Alt*, and *Ctrl* keys and pressed mouse buttons. The *Key* parameter of the *OnKeyUp* event handler is of type *Word*; therefore, you must use virtual key codes to determine the key pressed. You can find a table of virtual key codes in the Help system; search for the topic Virtual Key Codes.

Example

The following code changes a form's color to aqua when a key is pressed. When the key is released, the form's color reverts to the original color. Note that the *KeyPreview* property of the form must be set to *True* to capture all key presses, even if a control has focus:

```
{In the declarations section of the form}
var
  FormColor: TColor;

{OnKeyDown event handler}
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  FormColor := Form1.Color;
  Form1.Color := clAqua;
end;

{OnKeyUp event handler}
procedure TForm1.FormKeyUp(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  Form1.Color := FormColor;
end;
```

See also

KeyPreview property, *OnKeyDown* event, *OnKeyPress* event

OnLogin event

Applies to

TDataBase component

Declaration

```
property OnLogin: TLoginEvent;
```

The *OnLogin* event is activated whenever a *TDatabase* component assigned to an SQL database is opened and the *LoginPrompt* property is *True*. Use the *OnLogin* event to set login parameters. The *OnLogin* event gets a copy of the *TDatabase*'s login parameters array, *Params*. Use the *Values* property to change these parameters:

```
LoginParams.Values['SERVER NAME'] := 'MYSERVERNAME';
LoginParams.Values['USER NAME'] := 'MYUSERNAME';
LoginParams.Values['PASSWORD'] := 'MYPASSWORD';
```

When control returns from your *OnLogin* event handler, these parameters will be used to establish a connection.

Note For Paradox, dBASE, and ASCII databases, the only possible parameter is PATH, so the *OnLogin* event will not be activated.

OnMeasureItem event

Applies to

TComboBox, *TDBComboBox*, *TDBListBox*, *TListBox* components

Declaration

```
property OnMeasureItem: TMeasureItemEvent;
```

The *OnMeasureItem* event occurs whenever an application needs to redisplay an item in an owner-draw list box or combo box with a variable style. That is, for a list box, the *Style* property is *lbOwnerDrawVariable*, or for a combo box, the *Style* property is *csOwnerDrawVariable*.

The *OnMeasureItem* event passes three parameters to its handler describing the item to measure:

- The control containing the item
- The index of the item in the control
- The height of the item

The *OnMeasureItem* event handler should specify the height in pixels that the given item will occupy in the control. The *Height* parameter is a **var** parameter, which initially contains the default height of the item or the height of the item text in the control's font. The handler can set *Height* to a value appropriate to the contents of the item, such as the height of a graphical image to be displayed within the item.

After the *OnMeasureItem* event occurs, the *OnDrawItem* event occurs, rendering the item with the measured size.

Example

Here is a typical handler for an *OnMeasureItem* event. The example assumes that a variable owner-draw list box already has bitmaps associated with each of its strings. It sets the height of the item to the height of the associated bitmap if that height is greater than the default height.

```
procedure TForm1.ListBox1MeasureItem(Control: TWinControl; Index: Integer;
  var Height: Integer);
var
  Bitmap: TBitmap;
begin
  with Control as TListBox do
  begin
    Bitmap := TBitmap(Items.Objects[Index]);
    if Bitmap <> nil then
      if Bitmap.Height > Height then Height := Bitmap.Height;
  end;
end;
```

OnMeasureTab event

Applies to

TTabSet component

Declaration

```
property OnMeasureTab: TMeasureTabEvent;
```

The *OnMeasureTab* event occurs when the *Style* property of the tab set control is *tsOwnerDraw* and an application needs to redisplay a tab in a tab set control. In the *OnMeasureTab* event handler, you write the code to calculate the width needed to draw the tab. After the *OnMeasureTab* event occurs, the *OnDrawTab* event occurs. You write the code to draw the tab using the width calculated in *OnMeasureTab* in the *OnDrawTab* event handler.

The *Index* parameter of the *TMeasureTabEvent* method pointer is the position of the tab in the tab set control. The *TabWidth* parameter is the width of the tab.

Example

The following code measures the width of a bitmap stored in the *Objects* property of the *Tabs* list of the *DriveTabSet* tab set component. It then makes the width of the tab two pixels wider than the bitmap width.

```
procedure TFMForm.DriveTabSetMeasureTab(Sender: TObject; Index: Integer;
  var TabWidth: Integer);
var
  BitmapWidth: Integer;
begin
  BitmapWidth := TBitmap(DriveTabSet.Tabs.Objects[Index]).Width;
  Inc(TabWidth, 2 + BitmapWidth);
end;
```

OnMessage event

Applies to

TApplication component

Declaration

```
property OnMessage: TMessageEvent;
```

The *OnMessage* event occurs when your application receives a Windows message. By creating an *OnMessage* event handler in your application, you can call other handlers that respond to the message. If your application doesn't have a specific handler for an incoming message, the message is dispatched and Windows handles the message. An *OnMessage* event handler lets your application trap a Windows message before Windows itself processes it.

Note Search Help for "Handling Application Events" for more information about creating event handlers for application events.

Example

The following code displays the time of the most recently received Windows message in the *Caption* of *Label1*. *AppMessage* should be declared a method of *TForm1*.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnMessage := AppMessage;
```

```
end;

procedure TForm1.AppMessage(var Msg: TMsg; var Handled: Boolean);
begin
  Label1.Caption := IntToStr(Msg.Time);
end;
```

See also
ProcessMessages method

OnMinimize event

Applies to
TApplication component

Declaration

```
property OnMinimize: TNotifyEvent;
```

The *OnMinimize* event occurs when the application is minimized, either because the user minimizes the main window, or because of a call to the *Minimize* method. Use the *OnMinimize* event handler to put code that performs any special processing you want to happen when the application is minimized.

See also
OnRestore event, *Restore* method

OnMouseDown event

Applies to
TBitBtn, *TButton*, *TCheckBox*, *TDBCheckBox*, *TDBEdit*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBMemo*, *TDBNavigator*, *TDBText*, *TDirectoryListBox*, *TDrawGrid*, *TEdit*, *TFileListBox*, *TForm*, *TGroupBox*, *TImage*, *TLabel*, *TListBox*, *TMaskEdit*, *TMemo*, *TNotebook*, *TOLEContainer*, *TOutline*, *TPaintBox*, *TPanel*, *TRadioButton*, *TScrollBox*, *TShape*, *TSpeedButton*, *TStringGrid*, *TTabSet* components

Declaration

```
property OnMouseDown: TMouseEvent;
```

The *OnMouseDown* event occurs when the user presses a mouse button with the mouse pointer over a control. Use the *OnMouseDown* event handler when you want some processing to occur as a result of pressing a mouse button.

The *Button* parameter of the *OnMouseDown* event identifies which mouse button was pressed. By using the *Shift* parameter of the *OnMouseDown* event handler, you can respond to the state of the mouse buttons and shift keys. Shift keys are the *Shift*, *Ctrl*, and *Alt* keys.

Example

The following code creates and displays a label when a mouse button is pressed. If you attach this event handler to the *OnMouseDown* event of a form, a label specifying the coordinates of the mouse pointer appears when the user clicks the mouse button. Note that the *StdCtrls* unit must be added to the **uses** clause of the **interface** section of the form's unit to be able to create labels dynamically.

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  NewLabel: TLabel;
begin
  NewLabel := TLabel.Create(Form1);
  NewLabel.Parent := Self;
  NewLabel.Left := X;
  NewLabel.Top := Y;
  NewLabel.Caption := '(' + IntToStr(X) + ',' + IntToStr(Y) + ')';
  NewLabel.Visible := True;
end;
```

See also

OnMouseMove event, *OnMouseUp* event

OnMouseMove event

Applies to

TBitBtn, *TButton*, *TCheckBox*, *TDBCheckBox*, *TDBEdit*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBMemo*, *TDBNavigator*, *TDirectoryListBox*, *TDrawGrid*, *TEdit*, *TFileListBox*, *TForm*, *TGroupBox*, *TImage*, *TLabel*, *TListBox*, *TMaskEdit*, *TMemo*, *TNotebook*, *TOLEContainer*, *TOutline*, *TPaintBox*, *TPanel*, *TRadioButton*, *TScrollBox*, *TShape*, *TSpeedButton*, *TStringGrid*, *TTabSet* components

Declaration

```
property OnMouseMove: TMouseMoveEvent;
```

The *OnMouseMove* occurs when the user moves the mouse pointer when the mouse pointer is over a control. Use the *OnMouseMove* event handler when you want something to happen when the mouse pointer moves within the control.

By using the *Shift* parameter of the *OnMouseDown* event handler, you can respond to the state of the mouse buttons and shift keys. Shift keys are the *Shift*, *Ctrl*, and *Alt* keys.

Example

The following code updates two labels when the mouse pointer is moved. The code assumes you have two labels on the form, *lblHorz* and *lblVert*. If you attach this code to the *OnMouseMove* event of a form, *lblHorz* continually displays the horizontal position of the mouse pointer, and *lblVert* continually displays the vertical position of the mouse pointer while the pointer is over the form.

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin
  lblHorz.Caption := IntToStr(X);
  lblVert.Caption := IntToStr(Y);
end;
```

See also

OnMouseDown event, *OnMouseUp* event

OnMouseUp event

Applies to

TBitBtn, *TButton*, *TCheckBox*, *TDBCheckBox*, *TDBEdit*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBMemo*, *TDBNavigator*, *TDBText*, *TDirectoryListBox*, *TDrawGrid*, *TEdit*, *TFileListBox*, *TForm*, *TGroupBox*, *TImage*, *TLabel*, *TListBox*, *TMaskEdit*, *TMemo*, *TNotebook*, *TOLEContainer*, *TOutline*, *TPaintBox*, *TPanel*, *TRadioButton*, *TScrollBox*, *TShape*, *TSpeedButton*, *TStringGrid*, *TTabSet* components

Declaration

```
property OnMouseUp: TMouseEvent;
```

The *OnMouseUp* event occurs when the user releases a mouse button that was pressed with the mouse pointer over a component. Use the *OnMouseUp* event handler when you want processing to occur when the user releases a mouse button.

The *OnMouseUp* event handler can respond to left, right, or center mouse button presses and shift key plus mouse button combinations. Shift keys are the *Shift*, *Ctrl*, and *Alt* keys.

Example

The following code draws a rectangle when the user presses a mouse button, moves the mouse, and releases the mouse button. When the mouse button is released, the rectangle appears on the form's canvas. Its top-left and bottom-right corners are defined by the location of the mouse pointer when the user pressed and released the mouse button.

```
var
  StartX, StartY: Integer;  {Declare in interface section of form's unit}

{Use this code as the OnMouseDown event handler of the form:}
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  StartX := X;
  StartY := Y;
end;

{Use this code as the OnMouseUp event handler of the form:}
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Form1.Canvas.Rectangle(StartX, StartY, X, Y);
  end;
```

See also
OnMouseDown event, *OnMouseMove* event

OnNewRecord event

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
property OnNewRecord: TDataSetNotifyEvent;
```

The *OnNewRecord* event is activated whenever a new record is added to the *dataset*. The event occurs after the *BeforeInsert* event and before the *AfterInsert* event. *OnNewRecord* enables you to initialize any fields of the record without marking the record as *Modified*. Any changes to the record after this event will cause *Modified* to be set.

See also
Append method, *Insert* method

OnNotify event

Applies to
TMediaPlayer component

Declaration

```
property OnNotify: TNotifyEvent;
```

An *OnNotify* event occurs upon the completion of a media control method (*Back, Close, Eject, Next, Open, Pause, PauseOnly, Play, Previous, Resume, Rewind, StartRecording, Step,* or *Stop*) when the *Notify* property is set to *True* before the call to the media control method. After an *OnNotify* event, the *Notify* property must be reset to *True* for the next *OnNotify* event to occur.

Example
Attach the following code to the *OnNotify* event handler of a media player named *MediaPlayer1*. If the *Notify* property of *MediaPlayer1* is set to *True*, this code displays the value of the *NotifyValue* property in a message dialog box.

```
  procedure TForm1.MediaPlayer1Notify(Sender: TObject);
  var
    MyString: string;
  begin
    case MediaPlayer1.NotifyValue of
      nvSuccessful : MyString := 'Success!';
      nvSuperseded : MyString := 'Superseded!';
      nvAborted    : MyString := 'Aborted!';
      nvFailure    : MyString := 'Failure!';
    end;
    MessageDlg('Notify value indicates: ' + MyString, mtInformation, [mbOk], 0)
  end;
```

See also

NotifyValue property, *Wait* property

OnOpen event

Applies to

TDDEClientConv, *TDDEServerConv* components

Declaration

```
property OnOpen: TNotifyEvent;
```

An *OnOpen* event occurs when a DDE conversation is opened. A DDE conversation can be initiated automatically or manually. Automatically open a conversation by setting the value of the *ConnectMode* property to *ddeAutomatic*. When the form containing the DDE client conversation component is created at run time, the DDE conversation opens. Manually open a conversation by setting the value of *ConnectMode* to *ddeManual* and calling the *OpenLink* method.

Example

The following code sends a macro to the server and closes the link immediately after opening it.

```
procedure TForm1.DdeClientConv1Open(Sender: TObject);
begin
  with DDEClientConv1 do
  begin
    ExecuteMacro('File|New', False);
    CloseLink;
  end;
end;
```

See also

OnClose event

OnPageChanged event

Applies to

TNotebook component

Declaration

```
property OnPageChanged: TNotifyEvent;
```

The *OnPageChanged* event occurs just after a new page becomes the active page. Use the *OnPageChanged* event handler to specify special processing you want to happen at that time.

Example

This example changes the color notebook page each time the *OnPageChanged* event occurs. To set up the example, add pages to the notebook with the Object Inspector using the *Pages* property.

```
var
  NewColor: TColor;

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

procedure TForm1.TabSet1Change(Sender: TObject; NewTab: Integer;
  var AllowChange: Boolean);
begin
  Notebook1.PageIndex := TabSet1.TabIndex;
end;

procedure TForm1.Notebook1PageChanged(Sender: TObject);
begin
  NewColor := Notebook1.Color + 3475;
  Notebook1.Color := NewColor;
end;
```

See also

ActivePage property, *Pages* property

OnPaint event

Applies to

TForm, *TPaintBox* component

Declaration

```
property OnPaint: TNotifyEvent;
```

The *OnPaint* event occurs when Windows requires the form or paint box to paint, such as when the form or paint box receives focus or becomes visible when it wasn't previously. Your application can use this event to draw on the canvas of the form or paint box.

Example

The following code is an entire unit that loads a background bitmap onto the *Canvas* of the main form in the *OnPaint* event handler.

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    procedure FormPaint(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    TheGraphic: TBitmap; { Add this declaration for the graphic}
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormPaint(Sender: TObject); { OnPaint event handler}
begin
  Form1.Canvas.Draw(0, 0, TheGraphic); { Draw the graphic on the Canvas }
end;

procedure TForm1.FormCreate(Sender: TObject); { OnCreate event handler }
begin
  TheGraphic := TBitmap.Create; { Create the bitmap object }
  TheGraphic.LoadFromFile('C:\APP\BKGRND.BMP'); { Load the bitmap from a file}
end;

end.
```

See also

Canvas property

OnPassword event

Applies to

TSession component

Declaration

```
property OnPassword: TPasswordEvent;
```

Run-time only. The *OnPassword* event is activated whenever a Paradox table is opened and the Borland Database Engine reports that the application does not have sufficient access rights. The value of *Sender* is the *Session* component. *Continue* determines whether the caller will make another attempt to access the database. The procedure should add any available additional passwords and set *Continue* to *True*. If there are no additional passwords available, set *Continue* to *False*.

If no *OnPassword* event is defined, *Session* will create a default dialog box for the user to enter a new password.

OnPokeData event

Applies to

TDDEServerItem component

Declaration

```
property OnPokeData: TNotifyEvent
```

The *OnPokeData* event occurs when the DDE client application pokes data to your DDE server application. When a client pokes data, it sends text to the linked DDE server. The *Text* and *Lines* properties will be updated to contain the poked data, then the *OnPokeData* event occurs.

If the DDE client is a Delphi application that uses a *TDDEClientConv* component, data is poked when the *PokeData* or *PokeDataLines* method is called.

Example

The following code uses a Boolean variable *FInPoke* to protect poked data from being lost by a DDE server application. *DoOnPoke* is the *OnPokeData* event handler for the DDE server item component named *DDETestItem*. When data is poked, *FInPoke* is set to *True*, the poked data is stored in the *Lines* property of *Memo2*, and *FInPoke* is set back to *False*.

The data should be protected because the server data is updated when the *Lines* of *Memo1* are updated by the user. *DoOnChange*, the *OnChange* event handler for *Memo1*, tests *FInPoke* before updating the server data in *DDETestItem*. Otherwise, data poked from the client could be lost when *Memo1.Lines* is changed.

```
var
  FInPoke: Boolean;
.
.
.
procedure TDdeSrvrForm.doOnPoke(Sender: TObject);
begin
  FInPoke := True;
  Memo2.Lines := DdeTestItem.Lines;
  FInPoke := False;
end;
```

```
procedure TDdeSrvrForm.doOnChange(Sender: TObject);
begin
  if not FInPoke then
    DdeTestItem.Lines := Memo1.Lines;
end;
```

See also
OnExecuteMacro event

OnPopup event

Applies to
TPopupMenu component

Declaration

```
property OnPopup: TNotifyEvent;
```

The *OnPopup* event occurs whenever a pop-up menu appears either because the user right-clicks the component when the pop-up menu's *AutoPopup* is *True* or because the *Popup* method executed. Use the *OnPopup* event handler when you want some special processing to occur when the component's pop-up menu appears.

Example
The following code enables the Paste item from the pop-up menu if the Clipboard has text data.

```
procedure TForm1.PopupMenu1Popup(Sender: TObject);
begin
  Paste1.Enabled := Clipboard.HasFormat(CF_TEXT);
end;
```

See also
PopupMenu property

OnPostClick event

Applies to
TMediaPlayer component

Declaration

```
property OnPostClick: EMPPostNotify;
```

An *OnPostClick* event is generated after the code of the *OnClick* event handler has been called. If *Wait* is *True* when the media player was clicked, *OnPostClick* won't be called until the completion of the *OnClick* code. If *Wait* is *False*, control can return to the application before completion of the *OnClick* code; therefore, the *OnPostClick* event may occur before the actions initiated by the *OnClick* event have completed.

For example, if the user clicks the Play button and the *DoDefault* parameter of the *OnClick* event handler for the media player is *True*, the media is played. If the media is long enough, it will still be playing when the *OnPostClick* event is generated if *Wait* is *True*. If *Wait* is False, however, *OnPostClick* won't occur until the media has finished playing.

See also

OnClick event

OnReplace event

Applies to

TReplaceDialog components

Declaration

```
property OnReplace: TNotifyEvent;
```

The *OnReplace* event occurs whenever the user chooses either the Replace or the Replace All button in the Replace dialog box. Use the *OnReplace* event to specify the processing that replaces text.

Because the *OnReplace* event occurs when the user chooses either the Replace or Replace All button, the code you write in the *OnReplace* event handler should determine which button was chosen and supply the appropriate logic. Use the *frReplace* and *frReplaceAll* values in the *Options* set to determine which button was chosen.

Example

The following code calls the user-defined routine *DoReplace* if the Replace button was clicked, or calls the user-defined routine *DoReplaceAll* if the ReplaceAll button was clicked.

```
procedure TForm1.ReplaceDialog1Replace(Sender: TObject);
begin
  if (ReplaceDialog1.Options*[frReplace])=[frReplace] then DoReplace
  else if (ReplaceDialog1.Options*[frReplaceAll])=[frReplaceAll] then DoReplaceAll;
end;
```

See also

OnFind event

OnResize event

Applies to

TDBNavigator, *TForm*, *TPanel*, *TScrollBox* components

Declaration

```
property OnResize: TNotifyEvent;
```

The *OnResize* event occurs whenever the form is resized while an application is running. Use the *OnResize* event handler when you want something to happen in your application when the form is resized.

Example

The following code keeps the right edge on *Button1* against the right edge of *Form1* when *Form1* is resized.

```
procedure TForm1.FormResize(Sender: TObject);
begin
  Button1.Left := (Form1.Width)-Button1.Width;
end;
```

OnRestore event

Applies to

TApplication component

Declaration

```
property OnRestore: TNotifyEvent;
```

The *OnRestore* event occurs when the previously minimized application is restored to its normal size, either because the user restores the application, or because the application calls the *Restore* method. Use the *OnRestore* event handler to put code that performs any special processing you want to happen as the application is restored.

See also

Minimize method, *OnMinimize* event

OnRowMoved event

Applies to

TDrawGrid, *TStringGrid* components

Declaration

```
property OnRowMoved: TMovedEvent;
```

The *OnRowMoved* event occurs when the user moves a row using the mouse. The user can move a row only if the *Options* property set includes the value *goRowMoving*.

Example

The following code displays the number of rows a row was moved in a label.

```
procedure TForm1.StringGrid1RowMoved(Sender: TObject; FromIndex, ToIndex: Longint);
begin
  Label1 := IntToStr(Abs(FromIndex-ToIndex));
end;
```

See also
OnColumnMoved event

OnScroll event

Applies to
TScrollBar component

Declaration

```
property OnScroll: TScrollEvent;
```

The *OnScroll* event occurs whenever the user uses the scroll bar control. Use the *OnScroll* event handler if you want something to happen when the user uses the scroll bar control. Within the handler, write the code that responds to the user using the scroll bar.

Example
The following code repositions the thumb tab position by varying amounts. If *Page Up* was pressed, the box moves up only one. If *Page Down* was pressed, the box moves down 10. This shows how you can use the *OnScroll* event handler to move the thumb tab by different increments than specified by the *LargeChange* and *SmallChange* properties.

```
procedure TForm1.ScrollBar1Scroll(Sender: TObject; ScrollCode: TScrollCode;
  var ScrollPos: Integer);
begin
  if ScrollCode = scPageUp then ScrollPos := ScrollPos - 1
    else if ScrollCode = scPageDown then ScrollPos := ScrollPos + 10;
  Label1.Caption := IntToStr(ScrollPos);
end;
```

OnSelectCell event

Applies to
TDrawGrid, *TStringGrid* component

Declaration

```
property OnSelectCell: TSelectCellEvent;
```

The *OnSelectCell* event occurs when the user selects a cell in a draw grid or string grid. Use the *OnSelectCell* event handler to write the code that handles the selecting of a cell. Using the *CanSelect* parameter of the event handler type, your code can determine whether the user can select a cell or not.

Example

The following code determines that the user cannot select a cell containing the text 'No'.

```
procedure TForm1.StringGrid1SelectCell(Sender: TObject; Col, Row: Longint;
  var CanSelect: Boolean);
begin
  CanSelect := not (StringGrid1.Cells[Col,Row]='No')
end;
```

OnSetEditText event

Applies to

TDrawGrid, *TStringGrid* component

Declaration

```
property OnSetEditText: TSetEditTextEvent;
```

The *OnSetEditText* event occurs when the user edits the text in the grid. The user can edit the text only if the *Options* property set contains the value *goEditing*. The *OnSetEditText* event makes the actual changes to the data. Use the *OnSetEditText* event handler to write the code to handle the changes to the text within a cell of the grid.

When the user edits data in a grid, the *OnSetEditText* event occurs to change the actual data, then the *OnGetEditText* event occurs to display the changed data in the grid.

See also

OnGetEditText event

OnSetText event

Applies to

TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
property OnSetText: TFieldSetTextEvent;
```

The *OnSetText* event is activated when the *Text* property is assigned a value. If *OnSetText* has been assigned a method, the default processing for *Text* does not occur. The event handler must store the text provided by *Text*.

By assigning a method to this property, you can take any special actions required by the event.

OnShow event

Applies to

TForm component

Declaration

```
property OnShow: TNotifyEvent;
```

The *OnShow* event occurs just before a form becomes visible. Use the *OnShow* event to specify any special processing you want to happen before the form appears.

Example

This example colors the form and changes its caption when it becomes visible:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  Color := clLime;
  Caption := 'I am showing';
end;
```

See also

OnHide event

OnShowHint event

Applies to

TApplication component

Declaration

```
property OnShowHint: TShowHintEvent;
```

The *OnShowHint* event occurs when the application is about to display a hint window for a Help Hint for a particular control. By writing an event handler for *OnShowHint*, you can change the appearance and behavior of the Help Hint. Use the *HintStr*, *CanShow*, and *HintInfo* parameters of the *TShowHintEvent* method pointer to modify the Help Hint and its window. The *HintInfo* parameter is of type *THintInfo*, a record.

Example

This example uses three speed buttons on a panel. The code changes the color, width, and position of the text in the Help Hint for the third speed button.

You must declare the *DoShow* method in the type declaration of the form. Once it is declared, write the code for the *DoShow* method in the **implementation** part of the unit. Finally, in the *OnCreate* event handler for the form, assign the method to the *OnShowHint* event of the application.

```
type
  TForm1 = class(TForm)
    Panel1: TPanel;
    SpeedButton1: TSpeedButton;
    SpeedButton2: TSpeedButton;
    SpeedButton3: TSpeedButton;
    procedure FormCreate(Sender: TObject);
   private
    { Private declarations }
  public
    procedure DoShowHint(var HintStr: string; var CanShow: Boolean;
      var HintInfo: THintInfo;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.DoShowHint(var HintStr: string; var CanShow: Boolean;
   var HintInfo: THintInfo);
begin
  if HintInfo.HintControl = SpeedButton3 then
  begin
    with HintInfo do
    begin
      HintColor := clAqua;                             { Changes only for this hint }
      MaxHintWidth := 120;       {Hint text word wraps if width is greater than 120 }
      Inc(HintPos.X, SpeedButton3.Width);                   { Move hint to right edge }
    end;
  end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnShowHint := DoShowHint;
end;

end.
```

See also

Hint property, *OnHint* event, *ParentShowHint* property, *ShowHint* property

OnSized event

Applies to

THeader component

Declaration

```
property OnSized: TSectionEvent;
```

An *OnSized* event is generated when a sizing operation of a header is complete. A user can resize the header section at run time if the *AllowResize* property is set to *True*. Your application can resize the header section at run time by assigning a new value to the *SectionWidth* property.

Example

The following code displays the new width of the sized header section in a label.

```
procedure TForm1.Header1Sized(Sender: TObject; ASection, AWidth: Integer);
begin
  Label1.Caption := IntToStr(AWidth);
end;
```

See also

OnSizing event

OnSizing event

Applies to

THeader component

Declaration

```
property OnSizing: TSectionEvent;
```

An *OnSizing* event is generated for each mouse movement when a user is resizing a header by clicking and dragging at run time.

Example

The following code displays the width of the header section that is being resized. As the user drags the mouse pointer, the label is continuously updated.

```
procedure TForm1.Header1Sized(Sender: TObject; ASection, AWidth: Integer);
begin
  Label1.Caption := IntToStr(AWidth);
end;
```

See also

AllowResize property, *OnSized* event

OnStateChange event

Applies to

TDataSource component

Declaration

```
property OnStateChange: TNotifyEvent;
```

OnStateChange is activated when the *State* property changes.

By assigning a method to this property, you can react programmatically to state changes. For example, this event is useful for enabling or disabling buttons (for example, enabling an edit button only when a table is in edit mode), or displaying processing messages.

Note *OnChangeState* can occur even for **nil** datasets, so it is important to protect any reference to the *DataSet* property with a **nil** test:

```
if DataSource1.Dataset <> nil then
⋮
```

See also

OnDataChange event

OnStatusLineEvent event

Applies to

TOLEContainer component

Declaration

```
property OnStatusLineEvent: TStatusLineEvent;
```

An *OnStatusLineEvent* event occurs if an OLE server application has a message to display in the status line of the OLE container application when an OLE object is activated in place. Typically, your OLE container application handles an *OnStatusLineEvent* event by displaying the message string in its own status bar.

Example

The following code displays the status line message from the OLE server in*Panel1*.

```
procedure TForm1.OleContainer1StatusLineEvent(Sender: TObject; Msg: string);
begin
  Panel1.Caption := Msg;
end;
```

OnTimer event

Applies to

TTimer component

Declaration

```
property OnTimer: TNotifyEvent;
```

The *OnTimer* event is used to execute code at regular intervals. Place the code you want to execute within the *OnTimer* event handler.

The *Interval property* of a timer component determines how frequently the *OnTimer* event occurs. Each time the specified interval passes, the *OnTimer* event occurs.

Example

Here is an example of an *OnTimer* event handler that moves a ball slowly across the screen:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Timer1.Interval := 100;
  Shape1.Left := Shape1.Left + 1;
end;
```

See also

Interval property

OnTopLeftChanged event

Applies to

TDrawGrid, TStringGrid components

Declaration

```
property OnTopLeftChanged: TNotifyEvent;
```

The *OnTopLeftChanged* event occurs whenever the value of either the *TopRow* property or *LeftCol* property changes.

Example

The following code displays the latest top row and left column of *StringGrid1*.

```
procedure TForm1.StringGrid1TopLeftChanged(Sender: TObject);
begin
  with StringGrid1 do
    MessageDlg('The top row is now '+IntToStr(TopRow)+
      ' and the left col is now '+IntToStr(LeftCol), mtInformation, [mbOK],0);
end;
```

OnUpdateData event

Applies to

TDataSource component

Declaration

```
property OnUpdateData: TNotifyEvent;
```

OnUpdateData is activated by the *Post* or *UpdateRecord* method of a dataset component when the current record is about to be updated in the database. It causes all data-aware controls connected to the data source to be notified of the pending update, allowing them to change their associated fields to the current values in the controls. By assigning a method to this property, you can react programmatically to updates.

See also
BeforePost event

OnValidate event

Applies to
TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
property OnValidate: TFieldNotifyEvent;
```

The *OnValidate* event is activated when a field is modified. If a data-aware control is linked to the field, changes in the control do not activate *OnValidate* until the control attempts to store the results of those changes into the current record.

By assigning a method to this property, you can perform any special validation required for the field.

Example

```
   Field1.OnValidate := ValidateFieldRange;
```

Open method

Applies to
TClipboard object; *TDatabase*, *TMediaPlayer*, *TQuery*, *TStoredProc*, *TTable* components

For the Clipboard

Declaration

```
procedure Open;
```

The *Open* method opens the Clipboard and prevents other applications from changing its contents until the Clipboard is closed. If you are adding a single item to the

Clipboard, your application doesn't have to call *Open*. If you want to add a series of items to the Clipboard, however, *Open* prevents the contents from being overwritten with each addition.

When your application has added all items to the Clipboard, it should call the *Close* method.

Example

The following code opens a Clipboard object before two items (text from an edit box and an OLE object from an OLE container) are copied to the Clipboard. Then the Clipboard is closed.

```
Clipboard.Open;
Edit1.CopyToClipboard;
OLEContainer1.CopyToClipboard;
Clipboard.Close;
```

See also

Clear method, *Clipboard* variable, *Close* method

For media player controls

Declaration

```
procedure Open;
```

The *Open* method opens a multimedia device. The multimedia device type must be specified in the *DeviceType* property before you can open a device.

Upon completion, *Open* stores a numerical error code in the *Error* property, and the corresponding error message in the *ErrorMessage* property.

The *Wait* property determines whether control is returned to the application before the *Open* method is completed. The *Notify* property determines whether *Open* generates an *OnNotify* event.

Example

This example begins playing an audio CD when the application begins running. When the application is closed, the CD automatically stops playing. For this example to run successfully, you must have an audio CD device installed correctly.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
with MediaPlayer1 do
  begin
    DeviceType := dtCDAudio;
    Visible := False;
    Open;
    Play;
  end;
end;
```

See also
AutoOpen property, *Close* method

For tables, queries, and stored procedures

Declaration

```
procedure Open;
```

The *Open* method opens the *dataset, putting it in Browse state*. It is equivalent to setting the *Active* property to *True*.

For *TQuery*, *Open* executes the SELECT statement in the *SQL* property. If the statement does not return a result set (for example, an INSERT or UPDATE statement), then use *ExecSQL* instead of *Open*.

For *TStoredProc*, use *Open* to execute the stored procedure if the procedure returns a result set. If the stored procedure returns a single row, use *ExecProc* instead.

Example

```
try
  Table1.Open;
except
  on EDataBaseError do { The dataset could not be opened };
end;
```

See also
Close method

For databases

Declaration

```
procedure Open;
```

The *Open* method connects the *TDatabase* component to the server (or BDE for Paradox and dBASE databases). This is the same as setting *Connected* to *True*.

Example

```
Database1.Open;
```

OpenCurrent method

Applies to
TDirectoryListBox component

Declaration

```
procedure OpenCurrent;
```

The *OpenCurrent* method opens the directory selected in the directory list box, as if the user had double-clicked the directory.

Example

This example uses a directory list box, a button, and a label on a form. When the user selects a directory in the directory list box and clicks the button, the selected directory opens, and the path of the second directory displayed in the list box appears as the caption of the label.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  DirectoryListBox1.OpenCurrent;
  Label1.Caption := DirectoryListBox1.GetItemPath(1);
end;
```

OpenDatabase method

Applies to

TSession component

Declaration

```
function OpenDatabase(const DatabaseName: string): TDatabase;
```

The *OpenDatabase* method attempts to find a *TDatabase* component with a *DatabaseName* property matching the *DatabaseName* parameter by calling the *FindDatabase* method. If no such database can be found, it creates a new database component. *OpenDatabase* returns either the found database component or the one created. The database returned will be opened during this process. *OpenDatabase* increments the *Session's* reference count of the number of open database connections.

Use *OpenDatabase* with *CloseDatabase* in a **try...finally** block to ensure that database connections are handled properly.

Example

```
Database := Session.OpenDatabase('DBDEMOS');
try
  begin
   {Do Something}
  finally
    Session.CloseDatabase('DBDEMOS');
end;
```

See also

Session variable

OpenLink method

Applies to

TDDEClientConv component

Declaration

```
function OpenLink: Boolean;
```

The *OpenLink* method initiates a new DDE conversation. If the conversation was successfully opened, an *OnOpen* event occurs and *OpenLink* returns *True*. If the conversation wasn't successfully opened, *OpenLink* returns *False*.

Example

The following code requests data if a link is open.

```
   if OpenLink then DDEClientConv1.RequestData(DDEClientItem1.DDEItem);
```

See also

CloseLink method

Options property

Applies to

TIndexDef object; *TColorDialog*, *TDBGrid*, *TDBLookupCombo*, *TDBLookupList*, *TDrawGrid*, *TFindDialog*, *TFontDialog*, *TOpenDialog*, *TOutline*, *TPrintDialog*, *TReplaceDialog*, *TSaveDialog*, *TStringGrid* components

The *Options* property is a set of options that affects how dialog boxes, outlines, and grids appear and behave. The possible values contained within the set vary depending on the type of dialog box or if the component is an outline or grid control.

For Color dialog boxes

Declaration

```
property Options: TColorDialogOptions;
```

These are the possible values that can be included in the *Options* set:

Value	Meaning
cdFullOpen	Displays the custom coloring options when the Color dialog opens
cdPreventFullOpen	Disables the Create Custom Colors button in the Color dialog box so the user cannot create their own custom colors.
cdShowHelp	Adds a Help button to the Color dialog box.

The default value is [ ], the empty set, meaning all of these values are *False* and none of the options are in effect.

Example

This example displays the Color dialog box with a Help button and the Create Custom Colors button dimmed. The form is colored whatever color the user chooses.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ColorDialog1.Options := [cdPreventFullOpen, cdShowHelp];
  if ColorDialog1.Execute then
    Color := ColorDialog1.Color;
end;
```

See also

Color property, *CustomColors* property

For Font dialog boxes

Declaration

```
property Options: TFontDialogOptions;
```

These are the possible values that can be included in the *Options* set for the Fonts dialog box:

Value	**Meaning**
fdAnsiOnly	If *True*, the user can select fonts that use the Windows character set only; that is, the user can't choose a font that contains only symbols because they aren't displayed in the Font combo box.
fdEffects	If *True*, the Effects check boxes and the Color list box appear in the Font dialog box. The user uses the Effects check boxes to specify strikeout or underlined text and the Color list box to select a color for the selected font. If *fdEffects* is False, the Effects check boxes and Color list box don't appear in the Font dialog box.
fdFixedPitchOnly	If *True*, only monospaced fonts are displayed in the Font combo box.
fdForceFontExist	If *True* and the user enters a font name in the Font combo box and chooses OK, a message dialog box appears informing the user the font name is invalid.
fdLimitSize	If *True*, the *MinFontSize* and *MaxFontSize* properties can limit the number of fonts available in the dialog box.
fdNoFaceSel	If *True*, when the dialog box appears, no font name is selected in the Font combo box.
fdNoOEMFonts	If *True*, only fonts that aren't vector fonts are displayed in the Font combo box.
fdScalableOnly	If *True*, only fonts that can be scaled are displayed in the Font combo box.
fdNoSimulations	If *True*, only fonts that aren't GDI font simulations are displayed in the Font combo box.
fdNoSizeSel	If *True*, when the dialog box appears, no size is selected in the Size combo box.
fdNoStyleSel	If *True*, when the dialog box appears, no style is selected in the Style combo box.
fdNoVectorFonts	Same as *fdNoOEMFonts*.
fdShowHelp	If *True*, a Help button appears in the dialog box.

Value	**Meaning**
fdTrueTypeOnly	If *True*, only TrueType fonts are displayed in the Font list box.
fdWysiwyg	If *True*, only fonts that are available to both the printer and the screen appear in the Font combo box.

The default value is [*fdEffects*], meaning that only the *fdEffects* option is in effect.

Example

This example sets the options of the Font dialog box so that when the dialog box displays, only TrueType fonts show in the list of fonts and no font size is selected:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  FontDialog1.Options := [fdTrueTypeOnly, fdNoSizeSel];
  if FontDialog1.Execute then
    Memo1.Font := FontDialog1.Font;
end;
```

See also

Font property

For Print dialog boxes

Declaration

```
property Options: TPrintDialogOptions;
```

These are the possible values that can be included in the *Options* set for the Print dialog box:

Value	**Meaning**
poHelp	If *True*, a Help button appears in the dialog box.
poPageNums	If *True*, the Pages radio button is enabled and the user can specify a range of pages to print.
poPrintToFile	If *True*, a Print to File check box appears in the dialog box, giving the user the option to print to a file rather than to a printer.
poSelection	If *True*, the Selection radio button is enabled and the user can choose to print selected text.
poWarning	If *True* and if no printer is installed, a warning message appears when the user chooses OK.
poDisablePrintToFile	If *True* and *poPrintToFile* is *True*, the Print to File check box is dimmed when the dialog box appears. If *poPrintToFile* is *False*, setting *poDisablePrintToFile* to *True* has no effect because the dialog box won't have a Print to File check box.

The default value is [ ], the empty set, meaning that none of the possible options are in effect.

Example

This example displays the Printer dialog box that includes a Help button. If users try to print when no printer is installed, they will see a warning message.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  PrinterDialog1.Options := [poHelp, poWarning];
  if PrinterDialog1.Execute then
    ...
end;
```

See also

PrintRange property, *PrintToFile* property

For Open and Save dialog boxes

Declaration

```
property Options: TOpenOptions;
```

These are the possible values that can be included in the *Options* set for the Open and Save dialog boxes:

Value	Meaning
ofAllowMultiSelect	When *True*, this option allows users to select more than one file in the File Name list box.
ofCreatePrompt	When *True*, this option displays a dialog box with a message if the user enters a file name that doesn't exist in the File Name edit box and chooses OK. The message tells the user the file doesn't exist and asks if the user wants to create a new file with that name.
ofExtensionDifferent	This option is set when the file name returned from the dialog box has an extension that differs from the default file extension, the value in the *DefaultExt* property. Your application can then use this information. Setting an *ofExtensionDifferent* value with the Object Inspector has no meaning.
ofFileMustExist	If *True*, this option displays a dialog box with a message if the user enters a file that doesn't exist in the File Name edit box and chooses OK. The message informs the user the file can't be found and asks the user to make sure they entered the correct path and file name.
ofHideReadOnly	If *True*, this option hides the Read Only check box in the dialog box.
ofNoChangeDir	If *True*, this option sets the current directory to whatever the current directory was when the dialog box first appeared and ignores any directory changes the user made while using the dialog box.
ofNoReadOnlyReturn	If *True*, a message box appears informing the user if the selected file is read-only.
ofNoTestFileCreate	This option applies only when the user wants to save a file on a create-no-modify network share point, which can't be opened again once it has been opened. If *ofNoTestFileCreate* is *True*, your application won't check for write protection, a full disk, an open drive door, or network protection when saving the file because doing so creates a test file. Your application will then have to handle file operations carefully so that a file isn't closed until you really want it to be.
ofNoValidate	If *True*, this option doesn't prevent the user from entering invalid characters in a file name. If *ofNoValidate* is *False* and the user enters invalid characters for a file name in the File Name edit box, a message dialog box appears informing the user the file name contains invalid characters.

Value	Meaning
ofOverwritePrompt	If *True*, this option displays a message dialog box if the user attempts to save a file that already exists. The message informs the user the file exists and lets the user choose to overwrite the existing file or not.
ofReadOnly	If *True*, the Read Only check box is checked when the dialog box is displayed.
ofPathMustExist	If this option is *True*, the user can type only existing path names as part of the file name in the File Name edit box. If the user enters a path name that doesn't exist, a message box appears informing the user that the path name is invalid.
ofShareAware	If *True*, the dialog box ignores all sharing errors and returns the name of the selected file even though a sharing violation occurred. If *ofShareAware* is *False*, a sharing violation results in a message box informing the user of the problem.
ofShowHelp	If *True*, this option displays a Help button in the dialog box.

The default value is [ ], the empty set, meaning that none of the options are in effect.

Example

This example uses an Open dialog box and a button on a form. The code forces the user to enter valid file name characters, prevents the read-only check box from appearing in the dialog box, and let's the user choose to overwrite a file if the user selects a file that doesn't exist; the selected file name appears in a label on the form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  OpenDialog1.Options := [ofNoValidate, ofHideReadOnly, ofCreatePrompt];
  if OpenDialog1.Execute then
    Label1.Caption := OpenDialog1.FileName;
end;
```

For Find and Replace dialog boxes

Declaration

```
property Options: TFindOptions
```

The value of the *Options* property is the selected set of options that determine how the Find and Replace dialog boxes appear and behave. These are the possible values that can be contained in the *Options* set:

Value	Meaning
frDisableMatchCase	When *True*, the Match Case check box is dimmed and users cannot check it. When it is *False*, users can check the Match Case check box.
frDisableUpDown	When *True*, the Direction Up and Down buttons are dimmed and the user cannot select either of them. When it is *False*, users can select one of the Direction Up and Down buttons.
frDisableWholeWord	When True, the Match Whole Word check box is dimmed and user cannot select it. When it is *False*, users can check the check box.
frDown	When *True*, the Down button is selected in the dialog box and the search direction is down. When *frDown* is *False*, the Up button is selected, and the search direction is up. *frDown* can be set a design time, or users can change its value at run time when they use the dialog box.

Value	Meaning
frFindNext	This is a flag that is set when the user chooses the Find Next button. When *frFindNext* is *True*, your application should search for the string in the *FindText* property.
frHideMatchCase	When *True*, the Match Case check box is not visible in the dialog box. When it is *False*, the Match Case check box is visible.
frHideWholeWord	When *True*, the Match Whole Word check box is not visible in the dialog box. When it is *False*, the Match Whole Word check box is visible.
frHideUpDown	When *True*, the Direction Up and Down buttons are not visible in the dialog box. When it is *False*, the Direction Up and Down buttons are visible.
frMatchCase	When *True*, the Match Case check box is checked. When it is *False*, the Match Case check box is unchecked. You can set *frMatchCase* at design time, or users can change the value at run time.
frReplace	*frReplace* is a flag set by the system that indicates your application should replace the current occurrence of the *FindText* string with the *ReplaceText* string. *frReplace* applies only to the Replace dialog box.
frReplaceAll	*frReplaceAll* is a flag set by the system that indicates your application should replace all occurrences of the *FindText* string with the *ReplaceText* string. *frReplaceAll* applies only to the Replace dialog box.
frShowHelp	When *True*, a Help button appears in the dialog box when the dialog box displays. When *frShowHelp* is *False*, no Help button is present.
frWholeWord	When *True*, the Match Whole Word check box is checked in the dialog box. You can set *frWholeWord* at design time, or users can change its value at run time as they use the dialog box.

The default value is [*frDown*], meaning that only the *frDown* option is in effect.

Example

The following code calls the user-defined routine SearchDown if the Down button is selected in *FindDialog1* or it calls the user-defined routine SearchUp if the Up button is selected.

```
if (FindDialog1.Options*[frDown])=[frDown] then SearchDown
else SearchUp;
```

For outlines

Declaration

```
property Options: TOutlineOptions;
```

The Options property determines how the items in an outline are drawn. These are the possible values that can be contained in the *Options* set:

Value	Meaning
ooDrawTreeRoot	The first item (*Index* value of 1) is connected to the root item by the outline tree. This means that the tree will extend from the top of the outline to all the first level items. Without *ooDrawTreeRoot*, all first level items appear leftmost in the outline, not connected by the tree.

Value	Meaning
ooDrawFocusRect	The outline draws a focus rectangle around the selected item.
ooStretchBitmaps	The outline stretches the standard bitmaps (*PictureLeaf*, *PictureOpen*, *PictureClosed*, *PicturePlus*, *PictureMinus*) to fit in the size of the item, determined by the size of the *Font* of the *Text*. Without *ooStretchBitmap*, the bitmaps won't be stretched. They will be cropped if larger than the height of the item text, or won't fill up the entire item space if smaller than the text.

Example

The following code draws the tree of the outline to the root (extending from the first level items to the top of the outline).

```
   Outline1.Options := [ooDrawTreeRoot];
```

See also

OutlineStyle property, *Style* property

For draw and string grids

Declaration

```
property Options: TGridOptions;
```

These are the possible values that can be included in the *Options* set for the draw and string grid controls:

Value	Meaning
goFixedHorzLine	When *True*, horizontal lines appear between the rows within nonscrolling regions.
goFixedVertLine	When *True*, vertical lines appear between the columns within nonscrolling regions.
goHorzLine	When *True*, horizontal lines appear between the rows.
goVertLine	When *True*, vertical lines appear between the columns.
goRangeSelect	When *True*, the user can select a range of cells at one time. When *goEditing* is *True*, the user can no longer select a range of cells.
goDrawFocusSelected	When *True*, the cell with the focus is colored the same as other cells in a selected block are colored. When *False*, the cell with the focus remains the color of all unselected cells, the color specified with the grid *Color* property.
goRowSizing	When *True*, rows can be resized individually except for fixed or nonscrolling rows.
goColSizing	When *True*, columns can be resized individually except for fixed or nonscrolling columns.
goRowMoving	When *True*, the user can move a row to a new location in the grid using the mouse.
goColMoving	When True, the user can move a column to a new location in the grid using the mouse.
goEditing	When *True*, the user can edit the text in the grid. When *goEditing* is *True*, the user cannot select a range of cells at one time.
goAlwaysShowEditor	When *True*, the grid is in automatic edit mode if *goEditing* is also *True*. When the grid is in automatic edit mode, the user does not have to press *Enter* or *F2* before editing the contents of a cell. When *goAlwaysShowEditor* is *False* and *goEditing* is *True*, the user must press *Enter* or *F2* before editing the contents of a cell. If *goEditing* is *False*, setting *goAlwaysShowEditor* to *True* has no effect.

Value	Meaning
goTabs	When *True*, the user can use the *Tab* and *Shift-Tab* keys to move from column to column in the grid.
goRowSelect	When *True*, the user can select only whole rows at a time instead of individual cells.
goThumbTracking	When *True*, the contents of the grid scrolls while the user is moving the thumb tab of the grid scroll bar. When *False*, the contents of the grid doesn't scroll until the user releases the thumb tab in its new position.

Example

This code changes the look of the grid; only horizontal lines appear in both the body of the grid and in the nonscrolling regions when the user clicks the *ChangeGridStyle* button:

```
procedure TForm1.ChangeGridStyleClick(Sender: TObject);
begin
  DrawGrid1.Options := [goFixedHorzLine, goHorzLine];
end;
```

For data grids

Declaration

```
property Options: TDBGridOptions;
```

These are the possible values that can be included in the *Options* set for the data grid control:

Value	Meaning
dgEditing	When *True*, allows the user to edit data in the data grid. When the *ReadOnly* property is *True* and *dgEditing* is *True*, users can still use the *Insert* key to insert a blank row, or press the *Down Arrow* key when positioned at the bottom of the grid to append a blank row, although they won't be able to enter text in the new row.
dgAlwaysShowEditor	When *True*, the grid is in automatic edit mode as long as *gdEditing* is also *True*. When the grid is in automatic edit mode, the user does not have to press *Enter* or *F2* before editing the contents of a cell. When *gdAlwaysShowEditor* is *False* and *gdEditing* is *True*, the user must press *Enter* or *F2* before editing the contents of a cell. If *gdEditing* is *False*, setting *gdAlwaysShowEditor* to *True* has no effect.
dgTitles	When *True*, the column titles are visible.
dgIndicator	When *True*, a small pointer is visible that indicates which column is the current one.
dgColumnResize	When *True*, the columns can be resized. A column can't be resized, however, until its field has been added to the grid. To add a field to the grid, choose Add from the Fields editor.
dgColLines	When *True*, lines between the columns appear.
dgRowLines	When *True*, lines between the rows appear.
dgTabs	When *True*, users press the *Tab* key and the *Shift-Tab* keys to move among the columns of the data grid.
dgRowSelect	When *True*, the user can select whole rows only instead of individual cells.
dgAlwaysShowSelection	When *True*, the cell selected in the grid continues to display as selected even if the data grid doesn't have the focus.

Value	Meaning
dgConfirmDelete	When *True*, a message box appears if the user uses *Ctrl+Delete* to delete a row in the grid. The message box asks for confirmation that the row should really be deleted.
dgCancelOnExit	When *True*, if an insert is pending and no modifications were made by the user, the insert will be cancelled when the user exits the grid. This prevents the inadvertent posting of partial or blank records.

Example

This line of code displays column titles, makes the column indicator visible, and permits the user to edit the data displayed in the data grid:

```
procedure TForm1.FormClick(Sender: TObject);
begin
  DBGrid1.Options := [dgIndicator, dgEditing, dgTitles];
end;
```

See also

ReadOnly property

For database lookup combo boxes and list boxes

Applies to

TDBLookupCombo, *TDBLookupList* components

Declaration

```
property Options: TDBLookupListOptions;
```

The *Options* property determines how multiple columns in database lookup combo boxes and database lookup list boxes appear. These are the possible values that can be part of the *Options* set:

Value	Meaning
loColLines	When *True*, lines separate the columns displayed in the control. When *False*, no lines appear between the columns.
loRowLines	When *True*, lines separate the rows displayed in the control. When *False*, no lines appear between the rows.
loTitles	When *True*, the field names appear as titles above the columns in the control. When *False*, no titles appear.

To display multiple columns, use the *LookupDisplay* property.

Example

This code displays three fields in a database lookup list box, displays the field names as titles for the columns, and separates the columns with lines:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DBLookupList1.LookupDisplay := 'Company;City;Country';
```

```
    DBLookupList1.Options := [loColLines,loTitles];
  end;
```

See also

LookupDisplay property

For TIndexDef objects

Declaration

```
property Options: TIndexOptions;
```

Run-time and read only. *Options* is the set of characteristics of the index. Possible elements are those of the *TIndexOptions* type: *ixPrimary*, *ixUnique*, *ixDescending*, *ixNonMaintained*, and *ixCaseInsensitive*.

Ord function **System**

Declaration

```
function Ord(X): Longint;
```

The *Ord* function returns the ordinal value of an ordinal-type expression.

X is an ordinal-type expression. The result is of type *Longint*, and its value is the ordinal position of *X*.

Example

```
  uses Dialogs;

  type
     Colors = (RED,BLUE,GREEN);

  var
    S: string;
   begin
     S := 'BLUE has an ordinal value of ' + IntToStr(Ord(BLUE)) + #13#10;
     S := 'The ASCII code for "c" is ' + IntToStr(Ord('c')) +  ' decimal';
     MessageDlg(S, mtInformation, [mbOk], 0);
   end;
```

See also

Chr function

Orientation property

Applies to

TPrinter object

Declaration

```
property Orientation: TPrinterOrientation;
```

Run-time only. The value of the *Orientation* property determines if the print job prints vertically or horizontally on a page. These are the possible values:

Value	Meaning
poPortrait	The print job prints vertically on the page.
poLandscape	The print job prints horizontally on the page.

Example

This example uses two radio buttons on a form named *Landscape* and *Portrait*. The form also includes a button. When the user selects an orientation by clicking one of the radio buttons and then clicks the button to print one line of text, the print job prints using the selected orientation:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Printer.BeginDoc;
  Printer.Canvas.TextOut(100,100,'Hi there');
  Printer.EndDoc;
end;

procedure TForm1.PortraitClick(Sender: TObject);
begin
  Printer.Orientation := poPortrait;
end;

procedure TForm1.LandscapeClick(Sender: TObject);
begin
  Printer.Orientation := poLandscape;
end;
```

See also

Printer variable

Origin typed constant — WinCrt

Declaration

```
const Origin: TPoint = (X: 0; Y: 0);
```

The *Origin* typed constant contains the virtual screen coordinates of the character cell displayed in the upper left corner of the CRT window.

Origin is a read-only variable; do not assign values to it.

OutlineStyle property

Applies to
TOutline component

Declaration

```
property OutlineStyle: TOutlineStyle;
```

The *OutlineStyle* property determines how the outline structure is displayed within the *TOutline* component. The following table describes the outline styles.

Style	Description
osPictureText	Displays open picture (specified in *PictureOpen*), closed picture (specified in *PictureClosed*), leaf picture (specified in *PictureLeaf*) and item text (specified in *Text*).
osPlusMinusPictureText	Displays plus picture (specified in *PicturePlus*), minus picture (specified in *PictureMinus*), open picture, closed picture, leaf picture, and item text.
osPlusMinusText	Displays plus picture, minus picture, and item text.
osText	Displays item text.
osTreePictureText	Displays outline tree, open picture, closed picture, leaf picture, and item text.
osTreeText	Displays outline tree and item text.

Example
The following code displays pictures only if they are monochrome. The first choice is Open and Closed pictures. If they aren't monochrome, the code tests Plus and Minus pictures. The final resort is to simply display text.

```
with Outline1 do
  if (PictureOpen.Monochrome and PictureClosed.Monochrome) then
    OutlineStyle := osPictureText
  else if (PicturePlus.Monochrome and PictureMinus.Monochrome) then
    OutlineStyle := osPlusMinusText
  else OutlineStyle := osText;
```

OutOfMemoryError procedure — SysUtils

Declaration

```
procedure OutOfMemoryError;
```

OutOfMemoryError raises the *EOutOfMemory* exception.

Output variable — System

Declaration

```
var Output: TextFile;
```

The *Output* variable is a write-only file associated with the operating system's standard output file, which is usually the display.

In many of Delphi's standard file-handling routines, the file variable parameter can be omitted. Instead the routine operates on the *Input* or *Output* file variable. The following standard file-handling routines operate on the *Output* file when no file parameter is specified:

- *Write*
- *Writeln*

Since Windows does not support text-oriented input and output, *Input* and *Output* files are unassigned by default in a Windows application. Any attempt to read or write to them will produce an I/O error.

If the application uses the *WinCrt* unit, *Input* and *Output* will refer to a scrollable text window.

See also

Input variable, *TextFile* type

Overload property

Applies to

TStoredProc component

Declaration

```
property Overload: Word;
```

Oracle servers allow overloading of stored procedures in an Oracle package; that is, different procedures with the same name.

Set the *Overload* property to specify the procedure to execute on an Oracle server. If *Overload* is zero (the default), there is assumed to be no overloading. If *Overload* is one (1), then Delphi will execute the first stored procedure with the overloaded name; if it is two (2), it will execute the second, and so on.

See also

StoredProcName property

Owner property

Applies to

All components

Declaration

```
property Owner: TComponent;
```

Run-time and read only. The *Owner* property indicates which component owns the component.

The form owns all components that are on it. In turn, the form is owned by the application.

When one component is owned by another, the memory for the owned component is freed when its owner's memory is freed. This means that when a form is destroyed, all the components on the form are destroyed also. Finally, when the memory for the application itself is freed, the memory for the form (and all its owned components) is also freed.

Don't confuse ownership of a component with being the parent of a component. A parent is a windowed control that contains a child window. The parent and the owner of a windowed component can be different components.

Example

The example assumes there are two edit box controls on the form. When the form displays, the code inserts the name of the *Edit1* control's owner (*TForm1*) into *Edit1* itself, and displays the size of the owner in bytes in the second edit box (*Edit2*).

```
procedure TForm1.FormCreate(Sender: TObject);
var
  TC: TComponent;
  Size: Word;
  SizeStr: string;

begin
  TC := Edit1.Owner;
  Edit1.Text := TC.ClassName;
  Size := TC.InstanceSize;
  Str(Size, SizeStr);
  Edit2.Text := SizeStr;
end;
```

P

See also

Components property, *Destroy* method, *Free* method, *Parent* property

Pack method

Applies to

TList object

Declaration

```
procedure Pack;
```

The *Pack* method deletes all **nil** items from the list of pointers kept by the *List* property of a list object. Items become **nil** when the *Delete* or *Remove*' method has been called to delete them from the list.

Example

This example assumes there are two edit box controls on the form. The code creates a list object and adds two strings to it. The second string in the list is a **nil** string. The code counts the number of strings in the list and displays the number in the *Edit1* control. The code then packs the list, removing the **nil** string, and counts the strings in the list again. The second count displays in the *Edit2* control:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  MyList: TList;
  I: Integer;
  Buffer: string;
begin
  MyList := TList.Create;              {Create a list of TList}
  MyList.Add(PChar('Another string')); {Add a string}
  MyList.Add(PChar(NIL));              {Add a Nil string}
  Str(MyList.Count, Buffer);
  Edit1.Text := Buffer;                {Put count into Edit1}
  Mylist.Pack;                         {Pack the list.}
  Str(MyList.Count, Buffer);
  Edit2.Text := Buffer;                {Put count into Edit2}
  MyList.Free;                         {Free memory for list}
end;
```

See also

Expand method, *Remove* method, *Capacity* property

PageHeight property

Applies to

TPrinter object

Declaration

```
property PageHeight: Integer;
```

Run-time and read only. The *PageHeight* property contains the height of the currently printing page in pixels.

Example

This code displays the page height of the currently printing page in an edit box.

To run this code successfully, you must add *Printers* to the **uses** clause of your unit.

```
Edit1.Text := IntToStr(Printer.PageHeight);
```

See also

PageNumber property, *PageWidth* property, *Printer* variable

PageIndex property

Applies to

TNotebook, *TTabbedNotebook* components

Declaration

```
property PageIndex: Integer;
```

The value of the *PageIndex* property determines which page displays in the notebook or tabbed notebook component. Changing the *PageIndex* value changes the page in the control.

Each string in the *Pages* property is automatically assigned a *PageIndex* value when the page is created. The first page receives a value of 0, the second has a value of 1, and so on. If you delete a string from the *Pages* property, the *PageIndex* values are reassigned so that the values always begin with 0 and continue to increase without any gaps between values.

Example

This example assumes that a notebook component and a tab set component are on a form. It demonstrates how you can use the tab set and notebook component together to allow the user to click on a tab to access a page in the notebook component.

This code assigns the strings in the *Pages* property of the notebook component to the *Tabs* property of the tab set component. Because the code is in the *OnCreate* event handler when the form first appears, the tab set component has one tab for each page in the notebook component.

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

Changing the *PageIndex* value of a notebook or component changes the page displayed. This code assigns the *TabIndex* value of the tab the user clicks on to the *PageIndex* property of the notebook component. When the user clicks the tab labeled with a page name, that page is displayed in the notebook component.

```
procedure TForm1.TabSet1Click(Sender: TObject);
begin
  Notebook1.PageIndex := TabSet1.TabIndex;
end;
```

See also

ActivePage property, *TTabSet* component

PageNumber property

Applies to

TPrinter object

Declaration

```
property PageNumber: Integer;
```

Run-time and read only. The *PageNumber* property contains the number of the current page. Each time an application calls the *NewPage* method, *NewPage* increments the value of *PageNumber*.

Example

This example uses a button on a form. When the user clicks the button, one line of text is printed on six separate pages. As each page is printed, a message indicating the number of the page being printed appears on the form.

To run this example successfully, you must add *Printers* to the **uses** clause of your unit.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I, X, Y: Integer;
begin
  Printer.BeginDoc;
  X := 10;
  Y := 10;
    for I := 1 to 6 do
    begin
      Printer.Canvas.TextOut(100, 100, 'Object Pascal is great');
      Canvas.TextOut(X, Y, 'Printing page ' + IntToStr(Printer.PageNumber));
      Printer.NewPage;
      Y := Y + 20;
    end;
  Printer.EndDoc;
end;
```

See also

NewPage method, *Printer* variable

Pages property

Applies to

TNotebook, *TTabbedNotebook* components

Declaration

```
property Pages: TStrings;
```

The *Pages* property contains the strings that identify the individual pages of the notebook or tabbed notebook control. Both these controls create a separate page for each string in the *Pages* property. For example, if *Pages* contains three strings, First, Second, and Third, the control has three separate pages.

You can access the various pages in a notebook or tabbed notebook control with either the *ActivePage* or *PageIndex* property.

Example

The following code ensures that the *Pages* of *Notebook1* correspond with the value of the *Tabs* property of *TabSet1*.

```
Notebook1.Pages := TabSet1.Tabs;
```

See also

TTabSet component

PageWidth property

Applies to

TPrinter object

Declaration

```
property PageWidth: Integer;
```

Run-time and read only. The *PageWidth* property contains the value of width of the currently printing page in pixels.

Example

The code uses an edit box on a form. The code creates a printer object and displays the current width of a page in pixels in the edit box when the form first appears.

To run this example, you must add the *Printers* unit to the **uses** clause of your unit.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Edit1.Text := IntToStr(Printer.PageWidth) + ' pixels';
end;
```

See also

PageHeight property, *PageNumber* property, *Printer* variable

Palette property

Applies to

TBitmap object

Declaration

```
property Palette: HPalette;
```

The *Palette* property controls a bitmap's color mapping. The *Palette* of a bitmap contains up to 256 colors that can be used to display the bitmap onscreen.

If the bitmap is drawn by an application running in the foreground, as many colors of *Palette* as will be added to the Windows system palette. Any additional colors will be mapped to the existing colors of the system palette. If the bitmap is drawn by an application running in the background and another application has loaded the system palette with its own colors, the bitmap's colors will be mapped to the system palette.

Example

The following code selects the *Palette* from *Form1* for *Form2*.

```
SelectPalette(Form2.Canvas.Handle, Form1.Canvas.Palette, True);
```

ParamBindMode property

Applies to

TStoredProc component

Declaration

```
property ParamBindMode; TParamBindMode;
```

ParamBindMode determines how the elements of the *Params* array will be matched with stored procedure parameters. If *ParamBindMode* is set to *pbByName* (the default), parameters will be bound based on their names in the stored procedure. If *ParamBindMode* is set to *pbByNumber,* parameters will be bound based on the order in which they are defined in the stored procedure. Use this setting if you are building your parameters list, and you don't want to use the parameter names defined in the stored procedure.

Example

```
ParamBindMode := pbByName;
```

ParamByName method

Applies to

TParams object; *TQuery*, *TStoredProc* component

For TParam objects

Declaration

```
function ParamByName(const Value: string): TParam;
```

The *ParamByName* method finds a parameter with the name passed in *Value*. If a match is found, *ParamByName* returns the parameter. Otherwise, an exception is raised. Use this method rather than a direct reference to the *Items* property if you need to get a specific parameter to avoid depending on the order of the entries.

Example

```
try
{ Assign a value of 999 to the CustNo parameter }
  Params.ParamByName('CustNo').AsInteger := 999;
except
{ If it doesn't exist, then }
  on EDatabaseError do
{ Create a new parameter for CustNo and assign a value of 999 to it }
  with Params.CreateParam(ftInteger, 'CustNo', ptInput) do
    AsInteger := 999;
  end;
```

For queries and stored procedures

Declaration

```
function ParamByName(const Value: string): TParam;
```

The *ParamByName* method returns the element of the *Params* property whose *Name* property matches *Value*. Use it to assign values to parameters in a dynamic query by name.

Example

```
Query1.ParamByName('CustNo').AsString := '1231';
```

ParamCount function **System**

Declaration

```
function ParamCount: Word;
```

The *ParamCount* function returns the number of parameters passed to the program on the command line. Separate parameters with spaces or tabs.

Example

```
begin
  if ParamCount = 0 then
```

```
    Canvas.TextOut(10, 10, 'No parameters on command line')
  else
    Canvas.TextOut(10, 10, IntToStr(ParamCount) + ' parameter(s)');
end;
```

See also
ParamStr function

ParamCount property

Applies to
TQuery, *TStoredProc* component

For query components

Declaration

```
property ParamCount: Word;
```

Run-time and read only. The *ParamCount* property specifies how many entries the *TQuery* has in its *Params* array, that is, how many parameters the query has. Adding a new item to *Params* will automatically increase the value; removing an item will automatically decrease the value.

Example

```
for I := 0 to Query1.ParamCount - 1 do
  Query1.Params[I].AsInteger := I;
```

See also
Params property

For stored procedures

Declaration

```
property ParamCount: Word;
```

Run-time and read only. *ParamCount* specifies the total number of input and output parameters to the stored procedure, and is automatically maintained by changes to the *Params* property. Use *ParamCount* to iterate over the *Params*.

Example

```
{ Set all parameters to an empty string }
with StoredProc1 do
  for I := 0 to ParamCount - 1 do
    Param[I].AsString := '';
```

Params property

Applies to
TDatabase, *TQuery*, *TStoredProc* component

For stored procedures

Declaration

```
property Params: TParams;
```

The *Params* property holds the parameters to be passed to the stored procedure.

Example

```
{ Copy all parameters from StoredProc1 to StoredProc2 }
StoredProc1.CopyParams(StoredProc2.Params);
```

See also
CopyParams method, *ParamCount* property

For queries

Declaration

```
property Params[Index: Word]: TParam;
```

When you enter a query, Delphi creates a *Params* array for the parameters of a dynamic SQL statement. *Params* is a zero-based array of *TParam* objects with an element for each parameter in the query; that is, the first parameter is *Params*[0], the second *Params*[1], and so on. The number of parameters is specified by *ParamCount*. Read-only and run time only.

Note Use the *ParamByName* method instead of *Params* to avoid dependencies on the order of the parameters.

Example
For example, suppose a *TQuery* component named *Query2* has the following statement for its *SQL* property:

```
INSERT
  INTO COUNTRY (NAME, CAPITAL, POPULATION)
  VALUES (:Name, :Capital, :Population)
```

An application could use *Params* to specify the values of the parameters as follows:

```
Query2.Params[0].AsString := 'Lichtenstein';
Query2.Params[1].AsString := 'Vaduz';
Query2.Params[2].AsInteger := 420000;
```

These statements would bind the value “Lichtenstein” to the :Name parameter, “Vaduz” to the :Capital parameter, and 420000 to the :Population parameter.

For database components

Declaration

```
property Params: TStrings;
```

The *Params* property holds the parameters required to open a database on an SQL server. By default, these parameters are specified in the BDE Configuration Utility. You can customize these parameters for an application-specific alias with the Database Parameters Editor.

For desktop databases, *Params* will specify only the directory path for the database. For server databases, *Params* will specify a variety of parameters, including the server name, database name, user name, and password.

ParamStr function **System**

Declaration

```
function ParamStr(Index): string;
```

The *ParamStr* function returns a specified command-line parameter.

Index is an expression of type *Word*. *ParamStr* returns the parameter from the command line that corresponds to *Index*, or an empty string if *Index* is greater than *ParamCount*. For example, an *Index* value of 2 returns the second command-line parameter.

ParamStr(0) returns the path and file name of the executing program (for example, C:\BP\MYPROG.EXE).

Example

```
var
  I: Word;
  Y: Integer;
begin
  Y := 10;
  for I := 1 to ParamCount do begin
    Canvas.TextOut(5, Y, ParamStr(I));
    Y := Y + Canvas.TextHeight(ParamStr(I)) + 5;
  end;
end;
```

See also

ParamCount function

ParamType property

Applies to

TParam object

Declaration

```
property ParamType: TParamType;
```

ParamType is used to identify the type of the parameter for a stored procedure. Possible values are those of the *TParamType* type: *ptUnknown*, *ptInput*, *ptOutput*, *ptInputOutput*, or *ptResult*. Normally Delphi will set this property, but if the server does not provide the necessary information, you may have to set it yourself.

Example

```
StoredProc1.Params.ParamByName('CustNo').ParamType := ptInput;
```

Parent property

Applies to

All controls; *TMenuItem* component; *TOutlineNode* object

For controls

Declaration

```
property Parent: TWinControl;
```

The *Parent* property contains the name of the parent of the control. The parent of a control is the windowed control that contains the control. If one control (parent) contains others, the contained controls are child controls of the parent. For example, if your application includes three radio buttons in a group box, the group box is the parent of the three radio buttons, and the radio buttons are the child controls of the group box.

Don't confuse the *Parent* property with the *Owner* property. A form is the owner of all the components on it, whether or not they are windowed controls. A child control is always a windowed control contained within another windowed control (its parent). If you put three radio buttons in a group box on a form, the owner of the radio buttons is still the form, while the parent is the group box.

If you are creating a new control, you must assign a *Parent* property value for the new control. Usually, this is a form, panel, group box, or some control that is designed to contain another. It is possible to assign any windowed control as the parent, but the contained control is likely to be painted over.

When the parent of a control is destroyed, all controls that are its children are also destroyed.

Example

To set up the form for this example, put a group box on the form and add a radio button to the group box. Put two labels and a button on the form. This code displays the name of the parent of the radio button and the class name of the owner of the radio button in the captions of the two labels when the user clicks the button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := RadioButton1.Parent.Name + ' is the parent';
  Label2.Caption := RadioButton1.Owner.ClassName +
    ' is the class name of the owner';
end;
```

This example uses a button and a group box on a form. When the user clicks the button, the button moves inside the group box, because the group box is now the parent of the button.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Parent := GroupBox1;
end;
```

See also

Controls property, *Owner* property

For menu items

Declaration

```
property Parent: TMenuItem;
```

Run-time and read only. The *Parent* property of a menu item identifies the parent menu item of this menu item.

Example

This example assumes there are two edit boxes on a form as well as a main menu that contains menu items. One of the menu items has Save as the value of its *Caption* property, so the value of its *Name* property is *Save1*. The code displays the name of the parent of the *Save1* menu item in the *Edit1* control, and it displays the class name of the parent in the *Edit2* control when the form first appears.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Edit1.Text := Save1.Parent.Name;
  Edit2.Text := Save1.Parent.ClassName;
end;
```

For outline nodes

Declaration

```
property Parent: TOutlineNode;
```

The *Parent* property of an outline node identifies the parent outline item of this outline node. A parent outline item is one level higher and contains the child outline node as a subitem.

Example

The following code tests to see if the currently selected item has a sibling. *True* will be assigned to *HasSibling* if so.

```
var
  HasSibling: Boolean;
begin
  with Outline1[Outline1.SelectedItem] do
    HasSibling := (Parent.GetPrevChild <> -1) or (Parent.GetNextChild <> -1);
end;
```

See also

TopItem property

ParentColor property

Applies to

TCheckBox, *TComboBox*, *TDBCheckBox*, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBText*, *TDBListBox*, *TDBLookupCombo*, *TDBLookupList*, *TDBMemo*, *TDBRadioGroup*, *TDirectoryListBox*, *TDrawGrid*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TFilterComboBox*, *TGroupBox*, *TLabel*, *TListBox*, *TMaskEdit*, *TMemo*, *TNotebook*, *TOutline*, *TPaintBox*, *TPanel*, *TRadioButton*, *TScrollBox*, *TStringGrid* components

Declaration

```
property ParentColor: Boolean;
```

The *ParentColor* property determines where a control looks for its color information. If *ParentColor* is *True*, the control uses the color in its parent component's *Color property.* If *ParentColor* is *False*, the control uses its own *Color* property. Except for the radio group, database radio group, label and database text controls, the default value is *False*.

By using *ParentColor*, you can ensure that all the controls on a form have a uniform appearance. For example, if you change the background color of your form to gray, by default, the controls on the form will also have a gray background.

To specify a different color for a particular control, specify the desired color as the value of that control's *Color* property, and *ParentColor* becomes *False* automatically.

Example

This code uses a label control and a timer component on the form. When the *OnTimer* event occurs, the label turns red if the label's *ParentColor* property is *True*. If the *ParentColor* property is *False*, *ParentColor* is set to *True*. The result is the label flashes red on and off. Every other time an *OnTimer* event occurs, the label turns red. The other times, the label assumes the color of its parent, *Form1*.

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
 if Label1.ParentColor then
   Label1.Color := clRed
 else
   Label1.ParentColor := True;
end;
```

See also

Color property, *Parent* property, *ParentFont* property

ParentCtl3D property

Applies to

TCheckBox, *TComboBox*, *TDBCheckBox*, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBLookupCombo*, *TDBLookupList*, *TDBListBox*, *TDBNavigator*, *TDBMemo*, *TDBRadioGroup*, *TDirectoryListBox*, *TDrawGrid*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TFilterComboBox*, *TGroupBox*, *TListBox*, *TMaskEdit*, *TMemo*, *TNotebook*, *TOLEContainer*, *TOutline*, *TPanel*, *TRadioButton*, *TScrollBox*, *TStringGrid* components

Declaration

```
property ParentCtl3D: Boolean;
```

The *ParentCtl3D* property determines where a component looks to determine if it should appear three dimensional. If *ParentCtl3D* is *True*, the component uses the dimensionality of its parent component's *Ctl3D* property. If *ParentCtl3D* is *False*, the control uses its own *Ctl3D* property. The default value is *True*.

By using *ParentCtl3D*, you can ensure that all the components on a form have a uniform appearance. For example, if you want all components on a form to appear three dimensional, set the form's *Ctl3D* property to *True* and each component's *ParentCtl3D* property to *True*. Not only will all components have a three-dimensional appearance, but if you decide you prefer a two-dimensional appearance, you only have to change the *Ctl3D* property of the form and all the components will become two dimensional.

To specify a different dimensionality for a particular component, specify the dimensionality (*True* for 3D or *False* for 2D) as the value of that control's *Ctl3D* property, and *ParentCtl3D* becomes *False* automatically.

Example

This code uses a group box and a button on a form. The code displays the group box in two dimensions when the user clicks the button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if GroupBox1.ParentCtl3d = True then
  begin
    GroupBox1.ParentCtl3d := False;
    GroupBox1.Ctl3d := False;
  end;
end;
```

See also

Ctl3D property, *Parent* property, *ParentColor* property, *ParentFont* property

ParentFont property

Applies to

TBitBtn, *TCheckBox*, *TComboBox*, *TDBCheckBox*, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBLookupCombo*, *TDBLookupList*, *TDBListBox*, *TDBMemo*, *TDBRadioGroup*, *TDBText*, *TDirectoryListBox*, *TDrawGrid*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TFilterComboBox*, *TForm*, *TGroupBox*, *THeader*, *TListBox*, *TMaskEdit*, *TMemo*, *TNotebook*, *TOLEContainer*, *TOutline*, *TPaintBox*, *TPanel*, *TRadioButton*, *TScrollBox*, *TSpeedButton*, *TStringGrid* components

Declaration

```
property ParentFont: Boolean;
```

The *ParentFont* property determines where a control looks for its font information. If *ParentFont* is *True*, the control uses the font in its parent component's *Font property*. If *ParentFont* is *False*, the control uses its own *Font* property.

By using *ParentFont*, you can ensure that all the controls on a form have a uniform appearance. For example, if you want all the controls in a form to use 12-point Courier for their font, you can set the form's *Font* property to that font. By default, all the controls on that form will use the same font.

To specify a different font for a particular control, specify the desired font as the value of the control's *Font* property, and *ParentFont* becomes *False* automatically.

When the *ParentFont* is *True* for a form, the form uses the value of the application's *Font* property.

Example

This example uses a timer component and a label control. When an *OnTimer* event occurs and the label uses its parent's font, the code changes the label's *ParentFont* property to *False* and changes the label's font size to 30 points. When an *OnTimer* event occurs and the label doesn't use its parent's font, the code sets its *ParentFont* to *True*. The result is that the label's font grows and shrinks alternately, each time an *OnTimer* event occurs.

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if Label1.ParentFont = True then
    Label1.Font.Size := 30
  else
    Label1.ParentFont := True;
end;
```

This example uses a button on a form. When the user clicks the button, the font type and color change for all components on all forms in the application.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ParentFont := True;
  if Application.Font.Name = 'System' then
  begin
    Application.Font.Color := clNavy;
    Application.Font.Name := 'New Times Roman';
  end
  else
  begin
    Application.Font.Color := clBlack;
    Application.Font.Name := 'System'
  end;
end;
```

See also

Application variable, *Font* property, *Parent* property, *ParentColor* property, *ParentCtl3D* property, *TApplication* component

ParentShowHint property

Applies to

All controls

Declaration

```
property ParentShowHint: Boolean;
```

The *ParentShowHint* property determines where a control looks to find out if Help Hint, specified as the value of the *Hint* property for the control, should be shown. If *ParentShowHint* is *True*, the control uses the *ShowHint* property value of its parent. If *ParentShowHint* is *False*, the control uses its own *ShowHint* property.

By using *ParentShowHint*, you can ensure that all the controls on a form either show their Help Hints or don't show them. By default, *ParentShowHint* is *True*.

If don't want all the controls to have Help Hints, set the *ShowHint* property for those controls you do want to have Help Hints to *True*, and *ParentShowHint* becomes *False* automatically.

You can enable or disable all Help Hints for the entire application using the *ShowHint* property of the application.

Example

This example uses an edit box, a memo, and a check box on a form. For each of these controls, the *ParentShowHint* property is *True*, the default value. When the code runs, the *ShowHint* property of the form is set to *True* and hints are assigned to each control. Because each control looks to its parent, the form, to find out whether to display a Help Hint, and because the form's *ShowHint* property is *True*, the Help Hints are available.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ShowHint := True;
  Edit1.Hint := 'Enter text';
  Memo1.Hint := 'Enter lots of text';
  CheckBox1.Hint := 'Check or uncheck me';
end;
```

See also

Hint property, *ParentColor* property, *ParentCtl3D* property, *ParentFont* property

PasswordChar property

Applies to

TDBEdit, *TEdit*, *TMaskEdit* components

P

Declaration

```
property PasswordChar: Char;
```

The *PasswordChar* property lets you create an edit box that displays special characters in place of the entered text. By default, *PasswordChar* is the null character (ANSI character zero), meaning that the control displays its text normally. If you set *PasswordChar* to any other character, the control displays that character in place of each character in the control's text.

Example

The following code displays asterisks for each character in an edit box called *PasswordField*:

```
PasswordField.PasswordChar := '*';
```

PasteFromClipboard method

Applies to

TDBEdit, *TDBImage*, *TDBMemo*, *TEdit*, *TMaskEdit*, *TMemo* components

Declaration

```
procedure PasteFromClipboard;
```

The *PasteFromClipboard* method copies the contents of the Clipboard to the control, inserting the contents where the cursor is positioned.

Example

This example uses two edit boxes and a button on a form. When the user clicks the button, text is cut from the *Edit1* edit box and pasted into the *Edit2* edit box:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.SelectAll;
  Edit1.CutToClipboard;
  Edit2.Clear;
  Edit2.PasteFromClipboard;
  Edit1.SetFocus;
end;
```

See also

Clear method, *ClearSelection* method, *CopyToClipboard* method, *CutToClipboard* method

PasteSpecialDlg function

ToCtrl

Declaration

```
function PasteSpecialDlg (Form: TForm; const Fmts: array of BOleFormat;
  HelpContext: THelpContext; var Format: Word; var Handle: THandle;
  var PInitInfo: Pointer) : Boolean;
```

PasteSpecialDlg displays the Paste Special dialog box. Use this function to paste an OLE object from the Windows Clipboard into a *TOLEContainer* component. Specify the OLE object initialization information by using the Paste Special dialog box.

PasteSpecialDlg returns *True* if the user specifies an OLE object and chooses OK in the Paste Special dialog box. *PasteSpecialDlg* returns *False* if the user doesn't specify an OLE object or chooses Cancel in the dialog box.

These are the parameters of *PasteSpecialDlg*:

Field	Description
Form	The form that will own the Paste Special dialog box
Fmts	This is the array of object formats to register for pasting. An object format is specified in a *BOLEFormat* record. Each type of data you want to allow to be pasted should be passed as an element of the Fmts array. To paste OLE objects, you should register a new Clipboard format for OLE objects with the Windows API function *RegisterClipboardFormat* before calling *PasteSpecialDlg*. Then, you should specify a *BOLEFormat* array element for OLE objects. To paste other data types, such as text or bitmaps, specify a *BOLEFormat* array element for each other type of data.
HelpContext	A help context identification number to be used if the user chooses Help from within the Paste Special dialog box. If you pass 0 for *HelpContext*, no Help button will appear in the Paste Special dialog box. Pass a number other than 0 if you want to provide context-sensitive online Help.
Format	Format is modified by *PasteSpecialDlg* to specify the Clipboard format of the data selected by the user in the Paste Special dialog box. If the object is an OLE object, *Format* specifies the Clipboard format registered with *RegisterClipboardFormat*, prior to the call to *PasteSpecialDlg*. If the object is a type other than an OLE object, *Format* specifies its Clipboard format (for example, if the data is text, format specifies *CF_TEXT*).
Handle	*Handle* is modified by *PasteSpecialDlg* to provide a handle to the data on the Clipboard. If the data is a type other than an OLE object, use the *THandle* returned in the *Handle* parameter to access the data.
PInitInfo	If *InsertOLEObject* returns *True*, *InsertOLEObjectDlg* modifies the *PInitInfo* pointer parameter to point to OLE initialization information. Initialize the OLE object by assigning this pointer to the *PInitInfo* property. When your application is finished with the *PInitInfo* pointer, it should be released with *ReleaseOLEInitInfo*.

Example

The following code registers a new Clipboard format for embedded OLE objects and creates a object formats array for *FEmbedClipFmt*. If an embedded OLE object is on the Clipboard, the Paste Special Dialog box is displayed. If the user selects the object and chooses OK, then *OLEContainer1* is initialized.

```
var
  FEmbedClipFmt: Word;
  Fmts: array[0..0] of BOLEFormat;
  TheFormat: Word;
  TheHandle: THandle;
  TheInfo: Pointer;

begin
  FEmbedClipFmt := RegisterClipboardFormat('Embedded Object');
  Fmts[0].fmtId := FEmbedClipFmt;
  Fmts[0].fmtMedium := BOLEMediumCalc(FEmbedClipFmt);
  Fmts[0].fmtIsLinkable := False;
  StrPCopy (Fmts[0].fmtName, '%s');
  StrPCopy (Fmts[0].fmtResultName, '%s');
  if PasteSpecialEnabled(Self, Fmts) then
    if PasteSpecialDlg(Form1, Fmts, 0, TheFormat, TheHandle, TheInfo) then
      OLEContainer1.PInitInfo := TheInfo;
end;
```

See also
InsertOLEObjectDlg function, *LinksDlg* procedure, *PasteSpecialEnabled* function

PasteSpecialEnabled function **ToCtrl**

Declaration

```
function PasteSpecialEnabled(Form: TForm; const Fmts: array of BOleFormat): Boolean;
```

PasteSpecialEnabled determines if the Paste Special dialog box is enabled. If so, *PasteSpecialEnabled* returns *True* and *PasteSpecialDlg* can be successfully called. If not, *PasteSpecialEnabled* returns *False* and nothing will happen if you call *PasteSpecialDlg*.

The Paste Special dialog box is enabled if any of the object formats specified by the *Fmts* parameter is on the Clipboard.

Example
The following code calls *PasteSpecialDlg* if the Paste Special dialog box is enabled or displays a message if it is not enabled.

```
var
  Pasted: Boolean;

begin
  if PasteSpecialEnabled(Self, Fmts) then
    Pasted := PasteSpecialDlg(Form1, Fmts, 0, TheFormat, TheHandle, TheInfo)
  else
    MessageDlg('There are no OLE objects on the Clipboard', mtInformation, [mbOK], 0);
end;
```

See also
HasFormat method

Pause method

Applies to
TMediaPlayer component

Declaration

```
procedure Pause;
```

The *Pause* method pauses the open multimedia device. If the device is already paused when *Pause* is called, the device resumes playing or recording by calling the *Resume* method. *Pause* is called when the Pause button on the media player control is clicked at run time.

Upon completion, *Pause* stores a numerical error code in the *Error* property and the corresponding error message in the *ErrorMessage* property.

The *Wait* property determines whether control is returned to the application before the *Pause* method has completed. The *Notify* property determines whether *Pause* generates an *OnNotify* event.

Example

This example uses a media player, a timer, and a button on a form. Only the button is visible when the application runs. When the user clicks the button, the .WAV file plays. When the user clicks the button again, the .WAV file pauses. The caption of the button changes, depending on whether the .WAV file is playing, paused, or stopped.

To run this example, you must have the CHIMES.WAV file in your Windows directory and have a device that plays WAV audio files:

```
procedure TForm1.FormActivate(Sender: TObject);
var
  WinDir: PChar;
begin
  MediaPlayer1.Visible := False;
  GetMem(WinDir, 144);
  GetWindowsDirectory(WinDir, 144);
  StrCat(WinDir, '\chimes.wav');
  MediaPlayer1.FileName := StrPas(WinDir);
  MediaPlayer1.Open;
  FreeMem(WinDir, 144);
  Button1.Caption := 'Play';
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if Button1.Caption = 'Play' then
  begin
    Button1.Caption := 'Pause';
    MediaPlayer1.Play;
  end
  else
  begin
    Button1.Caption := 'Play';
    MediaPlayer1.Pause;
  end;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if MediaPlayer1.Mode = mpStopped then
    Button1.Caption := ‘Play’;
end;
```

See also

PauseOnly method, *Play* method, *StartRecording* method, *Stop* method

PauseOnly method

Applies to

TMediaPlayer component

Declaration

```
procedure PauseOnly;
```

The *PauseOnly* method only pauses the open multimedia device. If the device is already paused when *PauseOnly* is called, the device will remain paused.

Upon completion, *PauseOnly* stores a numerical error code in the *Error* property and the corresponding error message in the *ErrorMessage* property.

The *Wait* property determines whether control is returned to the application before the *PauseOnly* method has completed. The *Notify* property determines whether *PauseOnly* generates an *OnNotify* event.

Example

The following code illustrates the difference between *Pause* and *PauseOnly*. After the second call to *Pause*, *MediaPlayer1* resumes playing. After the second call to *PauseOnly*, *MediaPlayer1* is still paused.

```
with MediaPlayer1 do begin
  MediaPlayer1.Play;
  MediaPlayer1.Pause;
    { Now its paused }
  MediaPlayer1.Pause;
    { Now its playing }
  MediaPlayer1.PauseOnly;
    { Now its paused }
  MediaPlayer1.PauseOnly;
    { Now its still paused }
end;
```

See also

Pause method, *Play* method, *Resume* method, *StartRecording* method, *Stop* method

Pen property

Applies to

TCanvas object; *TShape* component

Declaration

```
property Pen: TPen;
```

A canvas object's *Pen* property determines what kind of pen the canvas uses for drawing lines and shape outlines.

Example

The following code prints a rectangle that uses a pen 40 pixels wide when the user clicks the button on the form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Printer.Canvas.Pen.Width := 40;
  Printer.BeginDoc;
  Printer.Canvas.Rectangle(30, 30, 400, 600);
  Printer.EndDoc;
end;
```

Before running this code, you must add the *Printers* unit to the **uses** clause of your unit.

See also

TBrush object, *TFont* object

PenPos property

Applies to

TCanvas object

Declaration

```
property PenPos: TPoint;
```

The *PenPos* property is the current drawing position of the pen. You should use the *MoveTo method* to set the drawing position, rather than changing *PenPos* directly.

See also

MoveTo method

Pi function System

Declaration

```
function Pi: Real;
```

The *Pi* function returns the value of *Pi*, which is defined as 3.1415926535897932385.

Precision varies, depending on whether the compiler is in 80x87 or software-only mode.

Example

```
var
  S: string;
begin
   Str(Pi:10:11, S);
   Canvas.TextOut(10, 10, 'Pi = ' + S);
end;
```

Picture property

Applies to

TDBImage, *TImage* components

Declaration

```
property Picture: TPicture;
```

The *Picture* property determines the image that appears on the image control. The property value is a *TPicture* object which can contain an icon, metafile, or bitmap graphic.

Example

This example uses two picture components. When the form first appears, two bitmaps are loaded into the picture components and stretched to fit the size of the components. To try this code, substitute names of bitmaps you have available.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Image1.Stretch := True;
  Image2.Stretch := True;
  Image1.Picture.LoadFromFile('BITMAP1.BMP');
  Image2.Picture.LoadFromFile('BITMAP2.BMP');
end;
```

See also

Bitmap property, *Icon* property, *Metafile* property, *LoadFromFile* method, *SaveToFile* method

PictureClosed property

Applies to

TOutline component

Declaration

```
property PictureClosed: TBitmap;
```

The *PictureClosed* property determines the picture displayed in the *TOutline* component that represents an item, which contains subitems but is not expanded. By default, the *PictureClosed* property contains a picture of a closed file folder. The *OutlineStyle* property must be set to *osPictureText*, *osPlusMinusPictureText*, or *osTreePictureText* to display the *PictureClosed* picture.

Example

The following code loads a new bitmap for the *PictureClosed* property of *Outline1*.

```
Outline1.PictureClosed.LoadFromFile('C:\closed.bmp');
```

See also
PictureLeaf property, *PictureMinus* property, *PictureOpen* property, *PicturePlus* property

PictureLeaf property

Applies to
TOutline component

Declaration

```
property PictureLeaf: TBitmap;
```

The *PictureLeaf* property determines the picture displayed in the *TOutline* component that represents an item that contains no subitems. By default, the *PictureLeaf* property contains a bitmap of a document. The *OutlineStyle* property must be set to *osPictureText*, *osPlusMinusPictureText*, or *osTreePictureText* to display the *PictureLeaf* picture.

Example
The following code tests the *Width* of the leaf picture. If it is wider than ten pixels, the *OutlineStyle* is changed so that the leaf picture is not displayed.

```
if Outline1.PictureLeaf.Width > 10 then
  Outline1.OutlineStyle := osTreeText;
```

See also
PictureClosed property, *PictureMinus* property, *PictureOpen* property, *PicturePlus* property

PictureMinus property

Applies to
TOutline component

Declaration

```
property PictureMinus: TBitmap;
```

The *PictureMinus* property determines the picture displayed in the *TOutline* component that represents an item, which contains subitems and is expanded. By default, the *PictureMinus* property contains a bitmap of a minus sign. The *OutlineStyle* property must be set to *osPlusMinusPictureText* or *osPlusMinusText* to display the *PictureMinus* picture.

Example
The following code displays the same picture for the plus and minus states of *Outline1*. The same graphic appears whether an item is expanded or collapsed.

```
Outline1.PictureMinus := Outline1.PicturePlus;
```

See also
PictureClosed property, *PictureLeaf* property, *PictureOpen* property, *PicturePlus* property

PictureOpen property

Applies to
TOutline component

Declaration

```
property PictureOpen: TBitmap;
```

The *PictureOpen* property determines the picture displayed in the *TOutline* component that represents an item, which contains subitems and is expanded. By default, the *PictureOpen* property contains a bitmap of an open file folder. The *OutlineStyle* property must be set to *osPictureText*, *osPlusMinusPictureText*, or *osTreePictureText* to display the *PictureOpen* picture.

Example
The following code copies text ('Hello world') into the *PictureOpen* bitmap.

```
Outline1.PictureOpen.Canvas.TextOut(0, 0, 'Hello world');
```

See also
PictureClosed property, *PictureLeaf* property, *PictureMinus* property, *PicturePlus* property

PicturePlus property

Applies to
TOutline component

Declaration

```
property PicturePlus: TBitmap;
```

The *PicturePlus* property determines the bitmap displayed in the *TOutline* component that represents an item, which contains subitems but is not expanded. By default, the *PicturePlus* property contains a bitmap of a plus sign. The *OutlineStyle* property must be set to *osPlusMinusPictureText* or *osPlusMinusText* to display the *PicturePlus* picture.

Example
The following code allows the user to specify the graphic for the *PicturePlus* property of *Outline1* by using the Open dialog box .

```
if OpenDialog1.Execute then
  Outline1.PicturePlus.LoadFromFile(OpenDialog1.FileName);
```

See also
PictureClosed property, *PictureLeaf* property, *PictureMinus* property, *PictureOpen* property

Pie method

Applies to
TCanvas object

Declaration

```
procedure Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Longint);
```

The *Pie* method draws the section of an ellipse bounded by the rectangle (*X1*, *Y1*) and (*X2*, *Y2*) on the canvas. The section drawn is determined by two lines radiating from the center of the ellipse through the points (*X3*, *Y3*) and (*X4*, *Y4*).

Example
This code draws a section of an ellipse on the form's canvas when the user clicks the button on the form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Canvas.Pie(10, 10, 200, 200, 61, 3, 200, 61);
end;
```

See also
Ellipse method

PInitInfo property

Applies to
TOLEContainer component; *TOLEDropNotify* object

Declaration

```
property PInitInfo: Pointer;
```

PInitInfo specifies a pointer to the OLE object initialization information. Assigning a pointer, which points to valid OLE initialization information, to the *PInitInfo* property initializes the OLE object in the OLE container.

Typically, a valid *PInitInfo* pointer can be obtained by using the *InsertOLEObjectDlg* or *PasteSpecialDlg* functions, or as a property of the *TOLEDropNotify* object passed in the *Source* parameter of the *OnDragDrop* event when an OLE object is dropped on a form.

Example

The following code initializes *OLEContainer1* when an OLE object is dropped on the *Form1* at run time. Attach this code to the *OnDragDrop* event handler of *Form1*.

```
procedure TForm1.FormDragDrop(Sender, Source: TObject; X, Y: Integer);
begin
  if Source is TOLEDropNotify then
    with Source as TOLEDropNotify do
      OLEContainer1.PInitInfo := Source.PInitInfo;
end;
```

Pitch property

Applies to

TFont object

Declaration

```
property Pitch: TFontPitch;
```

The *Pitch* property specifies the pitch or width of the characters of a font. Characters with variable pitch can have varying widths. For example, the following characters are in a variable pitch font. Note that the width of ten 'i' characters is less than the width of ten 'M' characters.

iiiiiiiii
MMMMMMMMMM

The following characters are in a fixed-pitch font. Note that ten 'i' characters are the same width as ten 'M' characters:

```
iiiiiiiiii
MMMMMMMMMM
```

Here are the possible values for *Pitch*:

Value	Meaning
fpDefault	The font pitch is set to the default value, which depends on the font specified in the *Name* property.
fpFixed	The font pitch is set to fixed. All characters in the font have the same width.
fpVariable	The font pitch is set to variable. The characters in the font have different widths.

Note Setting the *Pitch* of a fixed-width font to *fpVariable* or a variable-width font to *fpFixed* might have no effect on the appearance of a font, or might cause another font to be substituted. For example, setting the pitch of MS Serif (a variable-pitch font, by default) to *fpFixed* causes *Courier* to be displayed.

Example

The following code toggles the pitch of the *Font* of *Label1* from variable to fixed or from fixed to variable.

```
if Label1.Font.Pitch = fpFixed then
  Label1.Font.Pitch := fpVariable
else
  if Label1.Font.Pitch = fpVariable then
    Label1.Font.Pitch := fpFixed;
```

See also

Font property

Pixels property

Applies to

TCanvas object

Declaration

```
property Pixels[X, Y: Longint]: TColor;
```

The *Pixels* array enables you to access any pixel on the canvas directly, to either set or read the color there. Each element in *Pixels* contains the color of the corresponding pixel in the canvas. The array indexes, *X* and *Y*, specify the horizontal and vertical coordinates of the pixel, respectively.

Example

This example draws a red line when the form becomes active. Attach the following code to the *OnActivate* event handler:

```
procedure TForm1.FormActivate(Sender: TObject);
var
  W: Word;
begin
  for W := 10 to 200 do
    Canvas.Pixels[W, 10] := clRed;
end;
```

PixelsPerInch property

Applies to

TFont object; *TForm*, *TScreen* components

Declaration

```
property PixelsPerInch: Integer;
```

There are three different properties called *PixelsPerInch*: one for forms, one for the screen, and one for fonts.

For forms

Declaration

```
property PixelsPerInch: Integer;
```

The *PixelsPerInch* property for a form determines how many pixels per inch are used to display a form. A higher value displays a smaller form at run time, and a lower value displays a larger form. This property is useful when your application runs on a computer system that uses a screen resolution different than the one you used to create the application. By specifying the pixels per inch used by the other computer system, you can be assured that the form appears as you designed it when your application runs.

Note Although you can change the *PixelsPerInch* value with the Object Inspector, you won't see the results until you run your application. Also, you must set the *Scaled* property to be *True*, or a change in the *PixelsPerInch* value has no effect.

Example

This example adds 30 to the form's *PixelsPerInch* property if the screen's *PixelsPerInch* property is greater than 100:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  Form1.Scaled := True;
  if Screen.PixelsPerInch > 100 then
    Form1.PixelsPerInch := Form1.PixelsPerInch + 30;
end;
```

See also

Scaled property

For the screen

Declaration

```
property PixelsPerInch: Integer;
```

Read and run-time only. The *PixelsPerInch* property determines how many pixels are in an inch using the current video driver. The value in *PixelsPerInch* is retrieved from Windows when Delphi loads.

Example

This example adds 30 to the form's *PixelsPerInch* property if the screen's *PixelsPerInch* property is greater than 100:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  Form1.Scaled := True;
  if PixelsPerInch > 100 then
```

```
    Form1.PixelsPerInch := Form1.PixelsPerInch + 30;
end;
```

See also

Screen variable

For fonts

Declaration

```
property PixelsPerInch: Integer;
```

The *PixelsPerInch* property affects printer fonts only and should not be modified. Delphi uses the *PixelsPerInch* property to ensure that when a font is copied from the form's canvas to the printer, the font is the same size in points. For example, if the font is 8 points on the screen, Delphi makes sure the font is 8 points when it is printed.

If you want to modify the size of a font, use the *Size* and *Height* properties.

See also

Height property, *Size* property

Play method

Applies to

TMediaPlayer component

Declaration

```
procedure Play;
```

The *Play* method plays the media loaded in the open multimedia device. *Play* is called when the Play button on the media player control is clicked at run time.

Upon completion, *Play* stores a numerical error code in the *Error* property and the corresponding error message in the *ErrorMessage* property.

The *Wait* property determines whether control is returned to the application before the *Play* method has completed. The *Notify* property determines whether *Play* generates an *OnNotify* event.

If the *StartPos* property is set, playing starts at the position specified in *StartPos*. Otherwise, playing starts at the current position, specified in the *Position* property. Similarly, if the *EndPos* property is set, playing stops at the position specified in *EndPos*. Otherwise, playing stops at the end of the medium.

Whether the medium (specified in the *Position* property) is rewound before playing starts depends on the *AutoRewind* property.

Example

This example uses a media player and a button on a form. When the application runs, only the button is visible. When the user clicks the button, the .WAV file plays.

To run this example, the file CHIMES.WAV must be in your Windows directory.

```
procedure TForm1.FormActivate(Sender: TObject);
var
  WinDir: PChar;
begin
  MediaPlayer1.Visible := False;
  GetMem(WinDir, 144);
  GetWindowsDirectory(WinDir, 144);
  StrCat(WinDir, '\CHIMES.WAV');
  MediaPlayer1.FileName := StrPas(WinDir);
  MediaPlayer1.Open;
  FreeMem(WinDir, 144);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  MediaPlayer1.Play;
end;
```

See also

Capabilities property, *Pause* method, *PauseOnly* method, *StartRecording* method, *Stop* method

Point function

Declaration

```
function Point(AX, AY: Integer): TPoint;
```

The *Point* function takes the x- and y-coordinates passed in *AX* and *AY* and returns a *TPoint* record. You'll most often use *Point* to construct a parameter for a function that requires one or more *TPoint*.

Example

The following code uses the *Polygon* method to draw a right triangle on a form called *Form1*:

```
Polygon([Point(10, 10), Point(10, 20), Point(20, 20)]);
```

See also

Rect function

PokeData method

Applies to
TDDEClientConv component

Declaration

```
function PokeData(Item: string; Data: PChar): Boolean;
```

The *PokeData* method sends data to a DDE server application. Text data from a linked control in the DDE client application is transferred to the linked section of the DDE server application. *Item* specifies the linked item in the DDE server. *Data* is a null-terminated string that specifies the text data to transfer to the DDE server.

The usual direction of data flow is from the DDE server to the DDE client application. Some DDE server applications won't accept poked data. *PokeData* returns *True* if the data was successfully transferred, or *False* if the data was not successfully transferred.

If you need to poke a string list rather than a single string, use the *PokeDataLines* method.

Note If either the *ExecuteMacro* or *ExecuteMacroLines* method was called with its *WaitFlg* parameter set to *True* prior to calling *PokeData*, you must wait until the server application has completed executing the macro before calling *PokeData*. Depending on the DDE server application, calling *PokeData* before the DDE server application has completed executing the macro might cause the macro to execute unsuccessfully or produce unpredictable results.

Example
The following code pokes the data that is in *Edit1* to the DDE server. The DDE item of the conversation is specified in the *DDEItem* property of *DDEClientItem1*. *TheData* is a *PChar* variable.

```
DDEClientConv1.PokeData(DDEClientItem1.DDEItem, StrPCopy(TheData, Edit1.Text));
```

See also
PokeDataLines method, *StrPCopy* function

PokeDataLines method

Applies to
TDDEClientConv component

Declaration

```
function PokeDataLines(Item: string; Data: TStrings): Boolean;
```

The *PokeDataLines* method sends data to a DDE server application. Text data from a linked control in the DDE client application is transferred to the linked section of the DDE server application. *Item* specifies the linked item in the DDE server. *Data* is a *TStrings* object that specifies the text data to transfer to the DDE server.

The usual direction of data flow is from the DDE server to the DDE client application. Some DDE server applications won't accept poked data. *PokeDataLines* returns *True* if the data was successfully transferred, or *False* if the data was not successfully transferred.

If you need to poke a single string rather than a string list, use the *PokeData* method.

Note If either the *ExecuteMacro* or *ExecuteMacroLines* method was called with its *WaitFlg* parameter set to *True* prior to calling *PokeDataLines*, you must wait until the server application has completed executing the macro before calling *PokeDataLines*. Depending on the DDE server application, calling *PokeDataLines* before the DDE server application has completed executing the macro might cause the macro to execute unsuccessfully or produce unpredictable results.

Example

The following code pokes the data that is in *Memo1* to the DDE server. The DDE item of the conversation is specified in the *DDEItem* property of *DDEClientItem1*. *TheData* is a *PChar* variable.

```
DDEClientConv1.PokeData(DDEClientItem1.DDEItem, Memo1.Lines));
```

See also

PokeData method

Polygon method

Applies to

TCanvas object

Declaration

```
procedure Polygon(Points: array of TPoint);
```

The *Polygon* method draws a series of lines on the canvas, connecting the points passed to it in *Points* (much as the *PolyLine* method would), then closes the shape by drawing a line from the last point to the first point. After drawing the complete shape, *Polygon* fills the shape using the current brush.

Example

This example draws a polygon in the specified shape, and fills it with the color teal:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  Canvas.Brush.Color := clTeal;
  Canvas.Polygon([Point(10, 10), Point(30, 10),
    Point(130, 30), Point(240, 120)]);
end;
```

See also

PolyLine method

PolyLine method

Applies to

TCanvas object

Declaration

```
procedure Polyline(Points: array of TPoint);
```

The *PolyLine* method draws a series of lines on the canvas with the current pen, connecting each of the points passed to it in *Points*.

Example

This example paints a series of connected lines in the color red:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Pen.Color := clRed;
  Canvas.PolyLine([Point(5, 5), Point(100, 40), Point(150, 120),
    Point(140, 200), Point(80, 100), Point(5, 5)]);
end;
```

See also

Pen property, *Polygon* method

Popup method

Applies to

TPopupMenu component

Declaration

```
procedure Popup(X, Y: Integer);
```

The *Popup* method displays a pop-up menu onscreen at the coordinates indicated by the values (in pixels) of *X* and *Y*.

Example

This example uses a pop-up menu. When the user presses the mouse button, the pop-up menu appears near the upper left corner of the form:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  PopupMenu1.AutoPopup := False;
  PopupMenu1.Popup(Form1.Left + 10, Form1.Top + 40);
end;
```

See also

AutoPopup property, *OnPopup* event, *PopupMenu* property

PopupComponent property

Applies to

TPopupMenu component

Declaration

```
property PopupComponent: TComponent;
```

Run-time only. The *PopupComponent* property contains the name of the component the user last clicked that displayed the pop-up menu. If your application has multiple controls that share the same pop-up menu, you can use *PopupComponent* to determine which of them last displayed the menu.

If you activate a pop-up menu by explicitly calling the *Popup* method, you should specify the name of the component you want to associate with the pop-up menu in the *PopupComponent* property

Example

This example uses two edit boxes, two memos, and one pop-up menu on a form. The pop-up menu contains Cut, Copy, and Paste commands. This code makes the pop-up menu available to both edit boxes and both memos:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  PopupMenu1.AutoPopup := True;
  Edit1.PopupMenu := PopupMenu1;
  Edit2.PopupMenu := PopupMenu1;
  Memo1.PopupMenu := PopupMenu1;
  Memo2.PopupMenu := PopupMenu1;
end;
```

These are the cut, copy, and paste *OnClick* events for the commands on the pop-up menu. The code only allows the user to cut and copy text from the edit boxes, and to paste text into the memo boxes.

```
procedure TForm1.Copy1Click(Sender: TObject);
begin
  if PopupMenu1.PopupComponent = Edit1 then
   Edit1.CopyToClipboard
  else
    if PopupMenu1.PopupComponent = Edit2 then
    Edit2.CopyToClipboard;
end;

procedure TForm1.Cut1Click(Sender: TObject);
begin
  if PopupMenu1.PopupComponent = Edit1 then
```

```
      Edit1.CutToClipboard
    else
      if PopupMenu1.PopupComponent = Edit2 then
      Edit2.CutToClipboard;
  end;

  procedure TForm1.Paste1Click(Sender: TObject);
  begin
    if PopupMenu1.PopupComponent = Memo1 then
      Memo1.PasteFromClipboard
    else
      if PopupMenu1.PopupComponent = Memo2 then
        Memo2.PasteFromClipboard;
  end;
```

See also
AutoPopup property, *OnPopup* event, *PopupMenu* property

PopupMenu property

Applies to
TBitBtn, *TButton*, *TCheckBox*, *TComboBox*, *TDBCheckBox*, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBLookupCombo*, *TDBLookupList*, *TDBListBox*, *TDBMemo*, *TDBNavigator*, *TDBText*, *TDBRadioGroup*, *TDirectoryListBox*, *TDrawGrid*, *TDriveComboBox*, *TEdit*, *TFileListBox*, *TForm*, *TGroupBox*, *TImage*, *TLabel*, *TListBox*, *TMaskEdit*, *TMemo*, *TNotebook*, *TPanel*, *TPaintBox*, *TRadioButton*, *TScrollBar*, *TScrollBox*, *TStringGrid* components

Declaration

```
property PopupMenu: TPopupMenu;
```

The *PopupMenu* property identifies the name of the pop-up menu that appears when the user selects the component and presses the right mouse button (if the pop-up menu's *AutoPopup* property is *True*), or when the *Popup* method of the pop-up menu executes.

Example
This example assigns the pop-up menu named *MyPopupMenu* to the form:

```
  procedure TForm1.FormActivate(Sender: TObject);
  begin
    PopupMenu := MyPopupMenu;
  end;
```

See also
OnPopup event

Pos function — System

Declaration

```
function Pos(Substr: string; S: string): Byte;
```

The *Pos* function searches for a substring in a string.

Substr and *S* are string-type expressions.

Pos searches for *Substr* within *S* and returns an integer value that is the index of the first character of *Substr* within *S*.

If *Substr* is not found, *Pos* returns zero.

Example

```
var S: string;
begin
  S := '   123.5';
  { Convert spaces to zeroes }
  while Pos(' ', S) > 0 do
    S[Pos(' ', S)] := '0';
end;
```

See also

Concat function, *Copy function*, *Delete procedure*, *Insert procedure*, *Length function*

Position property

Applies to

TControlScrollBar, *TForm*, *TMediaPlayer*, *TScrollBar* components

The *Position* property determines the visual position of a component or the current position within media loaded in a media player.

For forms

Declaration

```
property Position: TPosition;
```

The *Position* property determines the size and placement of the form when it appears in your application. These are the possible values:

Value	Meaning
poDesigned	The form appears positioned on the screen and with the same height and width as it had at design time.
poDefault	The form appears in a position on the screen and with a height and width determined by Delphi. Each time you run the application, the form moves slightly down and to the right. The right side of the form is always near the far right side of the screen, and the bottom of the form is always near the bottom of the screen, regardless of the screen's resolution.
poDefaultPosOnly	The form displays with the size you created it at design time, but Delphi chooses its position on the screen. Each time you run the application, the form moves slightly down and to the right. When the form can no longer move down and to the right and keep the same size while remaining entirely visible on the screen, the form displays at the top-left corner of the screen.
poDefaultSizeOnly	The form appears in the position you left it at design time, but Delphi chooses its size. The right side of the form is always near the far right side of the screen, and the bottom of the form is always near the bottom of the screen, regardless of the screen's resolution.
poScreenCenter	The form remains the size you left it at design time, but is positioned in the center of the screen.

The default value is *poDesigned*.

Example

This code assures that the first form will appear centered on the screen:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Position := poScreenCenter;
end;
```

For scroll bars

Declaration

```
property Position: Integer;
```

The *Position* property determines the position of the thumb tab on a scroll bar. When the user scrolls the scroll bar, the value of *Position* changes. You can also change where the thumb tab appears on the scroll bar by changing the value of *Position*.

For *TControlScrollBar* components, the value of the *Range* property determines the number of possible positions on a scroll bar that a thumb tab can assume. The default value is 0, which positions the thumb tab at the far left.

For *TScrollBar* components, the number of possible positions on the scroll bar is determined by the difference between the *Max* property and the *Min* property. If the *Min* and *Position* values are both 0, the thumb tab is positioned to the far left on a horizontal scroll bar and to the top of a vertical scroll bar. If *Min* is 10, *Position* can be no less than 10.

Example

This code places the thumb tab in the middle of the scroll bar:

```
ScrollBar1.Max := 1000;
ScrollBar1.Min := 500;
ScrollBar1.Position := 750;
```

See also

HorzScrollBar property, *Increment* property, *LargeChange* property, *SmallChange* property, *VertScrollBar* property

For media player controls

Declaration

```
property Position: Longint;
```

Run-time only. The *Position* property specifies the current position within the currently loaded medium. The value of *Position* is specified according to the current time format, which is specified in the *TimeFormat* property.

Position defaults to the beginning of the medium. If the medium supports multiple tracks, *Position* defaults to the beginning of the first track.

Example

The following code shows the position of the currently playing .WAV audio file (CARTOON.WAV in this example) in the *Caption* of a label. The current position is updated by *Timer1*.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  with MediaPlayer1 do begin
    DeviceType := dtWaveAudio;
    FileName := 'CARTOON.WAV';
    Open;
    TimeFormat := tfMilliseconds;
    Label1.Caption := IntToStr(Position);
    Play;
  end;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Label1.Caption := IntToStr(MediaPlayer1.Position);
end;
```

See also

Length property, *Start* property, *TrackPosition* property, *Tracks* property

For Find and Replace dialog boxes

Applies to
TFindDialog, *TReplaceDialog* component

Declaration

```
property Position: TPoint;
```

The *Position* property determines where the Find or Replace dialog box appears onscreen.

Example
This example uses a Find dialog box and a button on a form. When the user clicks the button, the Find dialog box appears on screen at location 100, 200.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  FindDialog1.Position := Point(100, 200);
  if FindDialog1.Execute then ;
end;
```

See also
CloseDialog method

Post method

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure Post;
```

The *Post* method writes the current record to the *database*. *Post* should be called after calling *Append* or *Insert* and making any desired changes to the fields of the current record.

Post behaves differently depending on a dataset's state.

- In Edit state, *Post* modifies the current record.
- In Insert state, *Post* inserts or appends a new record.
- In SetKey state, *Post* commits the changes to the search key buffer, and returns the dataset to Browse state.

Posting can be done explicitly, or implicitly as part of another procedure. When an application moves off the current record, Delphi calls *Post* implicitly. Calls to the *Next*, *MoveBy*, *Prior*, *First*, and *Last* methods perform a *Post* if the table is in Edit or Insert state.

The *Append*, *AppendRecord*, *Insert*, and *InsertRecord* methods also implicitly perform a *Post* of any pending data.

Note If the record can not be written to the database for some reason, the dataset will remain in Edit state.

Example

```
with Table1 do
  begin
  Append;
  FieldByName('CustNo').AsString := '9999';
  { Fill in other fields here }
  if { you are sure you want to do this} then Post
  else { if you changed your mind } Cancel;
  end.
```

See also

Cancel method

Precision property

Applies to

TBCDField, *TCurrencyField*, *TFloatField* components

Declaration

```
property Precision: Integer;
```

The *Precision* property is used in formatting numeric fields. The value of *Precision* is the number of decimal places to the right of the decimal point the numeric value should be formatted to before rounding begins. The default value is 15 decimal places.

Pred function System

Declaration

```
function Pred(X);
```

The *Pred* function returns the predecessor of the argument.

X is an ordinal-type expression. The result, of the same type as *X*, is the predecessor of *X*.

Example

```
uses Dialogs;

type
   Colors = (RED,BLUE,GREEN);
```

```
var
  S: string;
begin
  S := 'The predecessor of 5 is ' + IntToStr(Pred(5)) + #13#10;
  S := S + 'The successor of 10 is ' + IntToStr(Succ(10)) + #13#10;
  if Succ(RED) = BLUE then
    S := S + 'In the type Colors, RED is the predecessor of BLUE.';
  MessageDlg(S, mtInformation, [mbOk], 0);
end;
```

See also
Dec procedure, *Inc procedure*, *Succ function*

PrefixSeg variable — System

Declaration

```
var PrefixSeg: Word;
```

In a program, the *PrefixSeg* variable contains the selector (segment address) of the Program Segment Prefix (PSP) created by DOS and Windows when the application was executed.

In a library, *PrefixSeg* is always 0.

For a complete description of the PSP, refer to your Windows manuals.

Prepare method

Applies to
TQuery, *TStoredProc* components

For stored procedures

Declaration

```
procedure Prepare;
```

The *Prepare* method prepares the stored procedure to be executed. This allows the server to load the procedure and otherwise prepare for execution.

Example

```
StoredProc1.Prepare;
```

See also
Prepared property, *UnPrepare* method

For queries

Declaration

```
procedure Prepare;
```

The *Prepare* method sends a parameterized query to the database engine for parsing and optimization. A call to *Prepare* is not required to use a parameterized query. However, it is strongly recommended, because it will improve performance for dynamic queries that will be executed more than once. If a query is not explicitly prepared, each time it is executed, Delphi automatically prepares it.

Prepared is a *Boolean* property of *TQuery* that indicates if a query has been prepared.

If a query has been executed, an application must call *Close* before calling *Prepare* again. Generally, an application should call *Prepare* once—for example, in the *OnCreate* event of the form—then set parameters using the *Params* property, and finally call *Open* or *ExecSQL* to execute the query. Each time the query is to be executed with different parameter values, an application must call *Close*, set the parameter values, and then execute the query with *Open* or *ExecSQL*.

See also
Text property

Prepared property

Applies to
TQuery, *TStoredProc* components

For stored procedures

Declaration

```
property Prepared: Boolean;
```

Run-time only. The *Prepared* property is *True* if the stored procedure has been submitted to the server for optimization purposes. Setting *Prepared* to *True* will not execute the procedure; it simply advises the server that the procedure will need to be executed at some future time. Setting *Prepared* to *True* is equivalent to calling the *Prepare* method; setting it to *False* is equivalent to calling the *UnPrepare* method.

Example

```
{ Make sure that the server is aware that we will be executing the procedure }
with StoredProc1 do
  if not Prepared then Prepared := True;
```

See also
Prepare method, *UnPrepare* method

For queries

Declaration

```
property Prepared: Boolean;
```

Run-time only. The *Prepared* property specifies if the *Prepare* method has been called to prepare the *TQuery*. While preparing a query is not required, it is highly recommended in most cases.

Note Close the *TQuery* by setting the *Active* property to *False* before changing *Prepared*.

Example

```
if not Query1.Prepared then
  begin
  Query1.Close;
  Query1.Prepared := True;
  end;
```

See also

Params property, *UnPrepare* method

Preview property

Applies to

TReport component

Declaration

```
property Preview: Boolean;
```

The *Preview* property determines whether a report should be viewed onscreen or printed. If *Preview* is *True*, the report appears onscreen when the report is run. If *Preview* is *False*, the report is printed.

Example

This example uses a report component and a button on a form. When the user clicks the button, a message appears if the *Preview* property is *True*. If *Preview* is *True*, the *MyReport* report is sent to the screen; if *Preview* is *False*, the report prints on the printer.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Report1.ReportName := ‘MyReport’;
 if Report1.Preview then
    Application.MessageBox('Sending the report to the screen', 'Message box', MB_OK);
  Report1.Run;
end;
```

See also
ReportName property, *Run* method

Previous method

Applies to
TForm, *TMediaPlayer* components

The *Previous* method activates the previous form or media player track.

For forms

Declaration

```
procedure Previous;
```

The *Previous* method makes the previous child form in the form sequence the active form.

For example, if you have three child forms within a parent form in your MDI application and *Form4* is the active form, the *Previous* method makes *Form3* the active form. Calling *Previous* again makes *Form2* active. The next time your application calls *Previous*, the sequence starts over again and *Form4* becomes the active form once again.

The *Previous* method applies only to forms that are MDI parent forms (have a *FormStyle* property value of *fsMDIForm*).

Example
This code sample activates the previous child window of the parent (Form1) when the user selects a menu item named *Previous* on a menu.

```
procedure TForm1.Previous1Click(Sender: TObject);
begin
  Previous;
end;
```

See also
ArrangeIcons method, *Cascade* method, *Next* method, *Tile* method

For media players

Declaration

```
procedure Previous;
```

The *Previous* method sets the current position to the beginning of the previous track if the position was at the beginning of a track when *Previous* was called. If the position is at the first track or somewhere other than the beginning of a track when *Previous* was called, *Previous* sets the current position to the beginning of the current track. If the device doesn't use tracks, *Previous* sets the current position to the beginning of the medium, which is specified in the *Start* property. *Previous* is called when the Previous button on the media player control is clicked at run time.

Upon completion, *Previous* stores a numerical error code in the *Error* property and the corresponding error message in the *ErrorMessage* property.

The *Wait* property determines whether control is returned to the application before the *Previous* method has completed. The *Notify* property determines whether *Previous* generates an *OnNotify* event.

Example

The following code rewinds the media after playing has completed. Normally, setting *AutoRewind* to *True* would accomplish the same result, but if *EndPos* is set, *AutoRewind* has no effect. This code is essentially an *AutoRewind* for media with *EndPos* set.

```
with MediaPlayer1 do
begin
  EndPos := 3000;
  Play;
  Previous;
end;
```

See also

AutoRewind property, *Next* method, *Position* property, *Tracks* property

Print method

Applies to

TForm, *TReport* components

For forms

Declaration

```
procedure Print;
```

The *Print* method prints the form.

Example

This example uses a button named *PrintButton* on a form. When the user chooses the button, the form prints.

```
procedure TForm1.PrintButtonClick(Sender: TObject);
begin
  Print;
end;
```

See also
PrintScale property

For reports

Declaration

```
function Print: Boolean;
```

The *Print* method determines whether a ReportSmith report prints. *Print* sends a DDE message to ReportSmith Runtime and looks for a DDE message from ReportSmith Runtime in return. If *Print* returns *True*, ReportSmith Runtime received the message to print the report. If *Print* returns *False*, ReportSmith Runtime could not receive the DDE message at the current time.

Example
This example notifies the user if the report is being printed:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Report1.Print = True then
    MessageDlg('Printing the report', mtInformation, [mbOK], 0) ;
end;
```

See also
Preview property, *PrintCopies* property, *Run* method

PrintCopies property

Applies to
TReport component

Declaration

```
property PrintCopies: Word;
```

The value of the *PrintCopies* property determines how many copies of the report are printed when you run a report. Specify the number of copies you want printed when your report runs. The default value is 1.

Example
The following code reads the number of copies to print from an edit box.

```
Report1.PrintCopies := StrToInt(Edit1.Text);
```

See also
EndPage property, *StartPage* property

Printer variable

Printers

Declaration

```
Printer: TPrinter;
```

The *Printer* variable declares an instance of the *TPrinter* object. Use *Printer* when you want to print using the *TPrinter* object.

Printer is declared in the *Printers* unit. Whenever you use *Printer* and the *TPrinter* object, you must add *Printers* to the **uses** clause of your unit.

Example

This example prints a one-line print job when the user clicks the button on the form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Printer.BeginDoc;
  Printer.Canvas.TextOut(100,100, 'Programming is easy');
  Printer.EndDoc;
end;
```

PrinterIndex property

Applies to

TPrinter object

Declaration

```
property PrinterIndex: Integer;
```

Run-time only property. The *PrinterIndex* property specifies which printer listed in the *Printers* property is the currently selected printer.

To select the default printer, set the value of *PrinterIndex* to -1.

Example

The following code asks the user if they want to use the default printer. If they choose yes, *PrinterIndex* specifies the default printer. The code assumes that *Printer* is a *TPrinter* object.

```
if (MessageDlg('Do you want to use the default printer',
    mtInformation, mbYesNoCancel,0)=idYes) then
      Printer.PrinterIndex := -1;
```

See also

Printers property

Printers property

Applies to

TPrinter object

Declaration

```
property Printers: TStrings;
```

Run-time and read only. The *Printers* property is a list of all printers installed in Windows.

Example

The following code displays the names of all printers in *ListBox1*.

```
begin
  ListBox1.Items := Printer1.Printers;
end;
```

See also

Printer variable

Printing property

Applies to

TPrinter object

Declaration

```
property Printing: Boolean;
```

Run-time and read only. The *Printing* property determines whether a print job is printing. *Printing* is *True* when your application has called the *BeginDoc* method, but the *EndDoc* method (or the *Abort* method) hasn't been called yet.

Example

This code terminates the print job if the job is currently printing:

```
if Printer.Printing then
  Abort;
```

See also

Aborted property

PrintRange property

Applies to
TPrintDialog component

Declaration

```
property PrintRange: TPrintRange;
```

The *PrintRange* property determines the type of print range the application uses to print a file. These are the possible settings:

Value	Meaning
prAllPages	If set at run time, the user chose to print all pages of the print job. If you set the *PrintRange* value to *prAllPages* at design time, the All Pages radio button is selected when the Print dialog box first appears.
prSelection	If set at run time, the user chose to print only selected text. If you set the *PrintRange* value to *prSelection* at design time, the Selection radio button is selected when the Print dialog box first appears.
prPageNums	If set at run time, the user chose to specify a range of pages to print. If you set the *PrintRange* value to *prPageNum* at design time, the Pages radio button is selected when the Print dialog box first appears, and the user can specify a print range by page numbers. The page numbers are set through the *MinPage* and *MaxPage* properties.

The default value is *prAllPages*.

Note The *PrintRange* property can have the value *prSelection* only if the *Options property* set includes *poSelection*. Also, the *PrintRange* property can have the value *prPageNums* only if the *Options* property set includes *poPageNums*. If you select either of these *PrintRange* values at design time, but neglect to set the corresponding *Options* values to *True*, only the All Pages option will be enabled when your application displays the Print dialog box.

Example
The following code allows the printing of selected text.

```
PrintDialog1.Options := PrintDialog1.Options + [poSelection];
PrintDialog1.PrintRange := prSelection;
```

See also
Options property, *PrintToFile* property

PrintScale property

Applies to
TForm component

Declaration

```
property PrintScale: TPrintScale;
```

The *PrintScale* property determines the proportions of a printed form. These are the possible values:

Value	Meaning
poNone	No special scaling occurs; therefore, the printed form and how the form appears onscreen may have somewhat different proportions.
poProportional	The form is printed so that it maintains the same size that is has on the screen (the same number of pixels per inch is used).
poPrintToFit	The form is printed using the same screen proportions, but in a size that just fits the printed page.

The default value is *poProportional*.

Example

The following code maintains the proportions of the form when it is printed.

```
   Form1.PrintScale := poProportional;
   Form1.Print;
```

See also

Print method

PrintToFile property

Applies to

TPrintDialog component

Declaration

```
property PrintToFile: Boolean;
```

The *PrintToFile* property determines if the user has chosen to print the print job to a file rather than to a printer. If *True*, the user has checked the Print to File check box. If *False*, the user has unchecked the Print to File check box. If *PrintToFile* is set to *True* at design time, the Print to File check box is checked when the Print dialog box appears in your application. The default value is *False*.

Note The Print to File check box appears in the Print dialog box, only if the*Options property* set includes *poPrintToFile*. Otherwise, your users won't have the option of choosing to print to a file.

Example

This example displays a print dialog box with its Print to File check box checked:

```
   procedure TForm1.BitBtn1Click(Sender: TObject);
   begin
```

```
with PrintDialog1 do
  begin
    Options := [poPrintToFile];
    PrintToFile := True;
    if Execute then
      ...;
  end;
end;
```

Prior method

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure Prior;
```

The *Prior* method moves the current record position of the *dataset* backward by one record. If the dataset is in Insert or Edit state, *Prior* will perform an implicit *Post* of any pending data.

Example

```
{ Move to the previous record }
Table1.Prior;
if Table1.BOF then { No more records };
```

See also
First method, *Last* method, *MoveBy* method, *Next* method

P

PrivateDir property

Applies to
TSession component

Declaration

```
property PrivateDir: string;
```

Run-time only. *PrivateDir* specifies the path of the directory in which to store temporary files (for example, files used to process local SQL statements). You should set this property if there will be only one instance of the application running at a time. Otherwise, the temporary files from multiple application instances will interfere with each other.

See also
Session variable

ProblemCount property

Applies to
TBatchMove component

Declaration

```
property ProblemCount: Longint;
```

Run-time and read only. *ProblemCount* is the number of records which could not be added to *Destination* without loss of data due to field width constraints. If *AbortOnProblem* is *True*, then this number will be one, since the operation will be aborted when the problem occurs.

Example

```
   MessageDlg(IntToStr(BatchMove1.ProblemCount) + ' records had problems',
     mtInformation, [mbOK], 0);
```

See also
ProblemTableName property

ProblemTableName property

Applies to
TBatchMove component

Declaration

```
property ProblemTableName: TFileName;
```

If the *Execute* method is unable to move a record to *Destination* without data loss (caused by a field width conflict), the record will be placed in a new table with the name supplied in *ProblemTableName*. If *AbortOnProblem* is *True*, then there will be at most one record in this table since the operation will be aborted with that first record. *ProblemCount* will have the number of records placed in the new table. If *ProblemTableName* is not specified, the data in the record will still be trimmed and placed in the destination table.

Example

```
   BatchMove1.ProblemTableName := 'PROB.DB';
```

ProcessMessages method

Applies to
TApplication component

Declaration

```
procedure ProcessMessages;
```

The *ProcessMessages* method interrupts the execution of your application so that Windows can respond to events. For example, the user might want to move a form on the screen while your application is doing some complex processing that would ordinarily prevent Windows from responding to keyboard or mouse events. By calling *ProcessMessages*, your application permits Windows to process these events at the time *ProcessMessages* is called. The *ProcessMessages* method cycles the Windows message loop until it is empty and then returns control to your application.

Example

This example uses two buttons that are long enough to accommodate lengthy captions on a form. When the user clicks the button with the caption Ignore Messages, the code begins to generate a long series of random numbers. If the user tries to resize the form while the handler is running, nothing happens until the handler is finished. When the user clicks the button with the caption Process Messages, more random numbers are generated, but Windows can still respond to a series of mouse events, such as resizing the form.

Note How quickly these event handlers run depends on the microprocessor of your computer. A message appears on the form informing you when the handler has finished executing.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Button1.Caption := 'Ignore Messages';
  Button2.Caption := 'Process Messages';
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  I, J, X, Y: Word;
begin
  I := 0;
  J := 0;
  while I < 64000 do
  begin
    Randomize;
    while J < 64000 do
    begin
      Y := Random(J);
      Inc(J);
     end;
    X := Random(I);
    Inc(I);
  end;
  Canvas.TextOut(10, 10, 'The Button1Click handler is finished');
end;

procedure TForm1.Button2Click(Sender: TObject);
var
```

```
  I, J, X, Y: Word;
begin
  I := 0;
  J := 0;
  while I < 64000 do
  begin
    Randomize;
    while J < 64000 do
    begin
      Y := Random(J);
      Inc(J);
      Application.ProcessMessages;
    end;
  X := Random(I);
  Inc(I);
  end;
  Canvas.TextOut(10, 10, 'The Button2Click handler is finished');
end;
```

Ptr function **System**

Declaration

```
function Ptr(Seg, Ofs: Word): Pointer;
```

The *Ptr* function converts a segment base and an offset address to a pointer-type value. *Seg* and *Ofs* are expressions of type *Word*.

The result is a pointer that points to the address given by *Seg* and *Ofs*. Like **nil**, the result of *Ptr* is assignment compatible with all pointer types. The function result can be immediately dereferenced only if it is typecast:

```
if Byte(Ptr(Seg0040, $49)^) = 7 then
  Writeln('Video mode = mono');
```

Example

```
var P: ^Byte;
begin
  P := Ptr($40, $49);
  Canvas.TextOut(10, 10, 'Current video mode is ' + IntToStr(P^));
end;
```

See also

Addr function

PtrRec **SysUtils**

Declaration

```
PtrRec = record
  Ofs, Seg: Word;
end;
```

PtrRec declares a utility record that stores the offset and segment of a pointer as type *Word*.

See also

Ofs function, *Seg* function

Random function **System**

Declaration

```
function Random [ ( Range: Word) ];
```

The *Random* function returns a random number within the range 0 <= X < *Range*.

If *Range* is not specified, the result is a *real-type* random number within the range 0 <= X < 1.

To initialize the *Random* number generator, call *Randomize*, or assign a value to the *RandSeed* variable.

Example

```
var
   I: Integer;
 begin
   Randomize;
   for I := 1 to 50 do begin
     { Write to window at random locations }
     Canvas.TextOut(Random(Width), Random(Height), 'Boo!');
   end;
 end;
```

See also

Randomize procedure, *RandSeed* variable

Randomize procedure **System**

Declaration

```
procedure Randomize;
```

The *Randomize* procedure initializes the built-in random number generator with a random value (obtained from the system clock).

The random number generator should be initialized by making a call to *Randomize*, or by assigning a value to *RandSeed*.

Example

```
var
  I: Integer;
 begin
  Randomize;
  for I := 1 to 50 do begin
    { Write to window at random locations }
    Canvas.TextOut(Random(Width), Random(Height), 'Boo!');
  end;
 end;
```

See also

Random function, *RandSeed* variable

RandSeed variable **System**

Declaration

```
var RandSeed: LongInt;
```

The *RandSeed* variable stores the built-in random number generator's seed.

By assigning a specific value to *RandSeed*, the *Random* function can repetitively generate a specific sequence of random numbers.

This is useful for applications that deal with data encryption, statistics, and simulations.

See also

Random function, *Randomize* procedure

Range property

Applies to

TControlScrollBar component

Declaration

```
property Range: Integer;
```

The value of the *Range* property determines how far a horizontal or vertical form scroll bar can be scrolled. It also represents the virtual size of the form. For example, if the *Range* value of a horizontal scroll bar is 500, and the client width of the form is 200, the scroll bar position can range from 0 to 300. While the client width of the form is 200, the virtual client width of the form is 500, because the user can scroll the form that far.

If the value of *Range* for a horizontal scroll bar is less than the client width of the form or scroll box, no horizontal scroll bar appears on the form. Likewise, if the value of *Range* for a vertical scroll bar is less than the client height of the form or scroll box, no vertical scroll bar appears.

For a horizontal scroll bar, the *Range* is calculated to be the distance of the right edge of the control that is the farthest to the right in the scroll bar or form from the left edge of the scroll box or form plus an amount specified as the value of the *Margin* property. If the form or scroll box contains one or more controls that are right-aligned (their *Align* value is *alRight*), the width of these controls is also added to the *Range* calculation.

For a vertical scroll bar, the *Range* is calculated to be the distance of the bottom edge of the control farthest away from the top edge of the scroll box or form from the top of the scroll box or form plus an amount specified as the value of the *Margin* property. If the form or scroll box contains one or more controls that are bottom-aligned (their *Align* value is *alBottom*), the height of these controls is also added to the *Range* calculation.

Example

This example uses a button on a form. When the user clicks the button, a vertical scroll bar alternately appears and disappears on the form, because the value of the *Range* property changes with each click.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ClientHeight := 300;
  VertScrollBar.Visible := True;
  if VertScrollBar.Range = 290 then
    VertScrollBar.Range := 500
  else
    VertScrollBar.Range := 290;
end;
```

See also

ClientHeight property, *ClientWidth* property, *HorzScrollBar* property, *VertScrollBar* property, *Visible* property

Read method

Applies to

TBlobStream object

Declaration

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

The *Read* method copies up to *Count* bytes from the current position in the field to *Buffer*. *Buffer* must have at least *Count* bytes allocated for it. *Read* returns the number of bytes transferred (which may be less than the number requested in *Count*.) Transfers which require crossing a selector boundary in the destination will be handled correctly.

Example

```
BlobStream1.Read(MyBuf, 4096);
```

See also

TBlobField component, *TBytesField* component, *TVarBytesField* component

Read procedure System

Declaration

Typed files:

```
procedure Read(F , V1 [, V2,...,Vn ] );
```

Text files:

```
procedure Read( [ var F: Text; ] V1 [, V2,...,Vn ] );
```

The *Read* procedure can be used in the following ways.

- For typed files, it reads a file component into a variable.
- For text files, it reads one or more values into one or more variables.

With a type string variable

- *Read* reads all characters up to, but not including, the next end-of-line marker or until *Eof*(*F*) becomes *True*; it does not skip to the next line after reading. If the resulting string is longer than the maximum length of the string variable, it is truncated.
- After the first *Read*, each subsequent *Read* sees the end-of-line marker and returns a zero-length string.
- Use multiple *Readln* calls to read successive string values.

When the extended syntax is enabled, *Read* can read null-terminated strings into zero-based character arrays.

With type integer or type real variables

- *Read* skips any blanks, tabs, or end-of-line markers preceding the numeric string.
- If the numeric string does not conform to the expected format, an I/O error occurs; otherwise, the value is assigned to the variable.
- The next *Read* starts with the blank, tab, or end-of-line marker that terminated the numeric string.

See also

Eof function, *ReadKey function*, *Readln procedure*, *Write procedure*, *Writeln procedure*

ReadBool method

Applies to

TIniFile object

Declaration

```
function ReadBool(const Section, Ident: string; Default: Boolean): Boolean;
```

The *ReadBool* method retrieves a Boolean value in an .INI file.

The *Section* constant identifies the section of the .INI file in which to search for the value. For example, the WIN.INI for Windows contains a [Desktop] section.

The *Ident* parameter is the name of the identifier of which you want the value.

The *Default* parameter is the default value.

Example

This example reads the DELPHI.INI file and displays on the form the status of your auto save options.

To run this application, you must add the *IniFiles* unit to the **uses** clause of your unit.

```
procedure TForm1.FormActivate(Sender: TObject);
var
  DelphiIni: TIniFile;
begin
  DelphiIni := TIniFile.Create('Delphi.Ini');
  with DelphiIni do
  begin
    if ReadBool('AutoSave', 'EditorFiles', True) = True then
      Canvas.TextOut(10, 10, 'Auto saving editor files.')
    else
      Canvas.TextOut(10, 10, 'Not auto saving editor files.');
    if ReadBool('AutoSave', 'DesktopFile', True) = True then
      Canvas.TextOut(10, 50, 'Auto saving desktop file.')
    else
      Canvas.TextOut(10, 50, 'Not auto saving desktop file.');
  end;
  DelphiIni.Free;
end;
```

See also

ReadInteger method, *ReadSection* method, *ReadString* method, *WriteBool* method

ReadBuf function **WinCrt**

Declaration

```
function ReadBuf(Buffer: PChar; Count: Word): Word;
```

The *ReadBuf* function inputs a line from the CRT window.

Buffer points to a line buffer that can store up to *Count* characters. *Count* contains the number of characters to read.

Only *Count*–2 characters can be input because an end-of-line marker (a #13 followed by a #10) is automatically appended to the line when the user presses Enter.

If *CheckEof* is *True*, the user can terminate the input line by pressing *Ctrl+Z*, and the line will have an end-of-line marker (#26) appended to it.

ReadBuf returns the number of characters read, including the end-of-line or end-of-file marker.

Example

```
uses WinCrt;

var
  C: PChar;

begin
  GetMem(C, 20);
  C := #0#0#0#0#0#0#0#0#0#0#0#0#0#0#0#0#0#0#0#0;
  Writeln('Type a phrase up to 20 characters long:');
  ReadBuf(C, 20);
  Writeln(' You typed: ');
  Writeln(C);
end;
```

See also

ReadKey function

ReadFrom method

Applies to

TBitmap, *TGraphic*, *TIcon*, *TMetafile*, *TPicture* objects

Declaration

```
procedure ReadFrom(const Filename: string); virtual;
```

The *ReadFrom* method reads an image from the file named in *FileName*.

Example

To read an image into a bitmap object called *MyBitmap* from the file MYBITMAP.BMP,

```
MyBitmap.ReadFrom('MYBITMAP.BMP');
```

See also

SaveToFile method

Readln procedure System

Declaration

```
procedure Readln([ var F: Text; ] V1 [, V2, ...,Vn ]);
```

The *Readln* procedure reads a line of text and then skips to the next line of the file.

Readln(*F*) with no parameters causes the current file position to advance to the beginning of the next line if there is one; otherwise, it goes to the end of the file.

{$I+} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using **{$I–}**, you must use *IOResult* to check for I/O errors.

Example

```
uses WinCrt;

var
   s : string;
 begin
   Write('Enter a line of text: ');
   Readln(s);
   Writeln('You typed: ',s);
   Writeln('Hit <Enter> to exit');
   Readln;
 end;
```

See also

Read procedure, *Writeln procedure*

ReadInteger method

R

Applies to

TIniFile object

Declaration

```
function ReadInteger(const Section, Ident: string; Default: Longint): Longint;
```

The *ReadInteger* method retrieves an integer value in an .INI file.

The *Section* constant identifies the section of the .INI file in which to search for the value. For example, the WIN.INI for Windows contains a [Desktop] section.

The *Ident* parameter is the name of the identifier of which you want the value.

The *Default* parameter is the default value.

Example

This example reads settings in the WIN.INI file and displays on the form the value of a few settings.

Before you run this example, you must add the *IniFiles* unit to the **uses** clause of your unit.

```
procedure TForm1.FormActivate(Sender: TObject);
var
  WinIni: TIniFile;
begin
  Canvas.TextOut(20, 10, 'VARIOUS WINDOWS SETTINGS');
  WinIni := TIniFile.Create('Win.Ini');
  with WinIni do
  begin
    Canvas.TextOut(10, 45, 'Border Width = ' +
      IntToStr(ReadInteger('Windows', 'BorderWidth', -1)));
    Canvas.TextOut(10, 65, 'Icon Spacing = ' +
       IntToStr(ReadInteger('Desktop', 'IconSpacing', -1)));
    Canvas.TextOut(10, 85, 'Grid Granularity = ' +
       IntToStr(ReadInteger('Desktop', 'GridGranularity', -1)));
    Canvas.TextOut(10, 105, 'Cursor Blink Rate = ' +
       IntToStr(ReadInteger('Windows', 'CursorBlinkRate', -1)));
    Canvas.TextOut(10, 125, 'Double Click Speed = ' +
       IntToStr(ReadInteger('Windows', 'DoubleClickSpeed', -1)));
  end;
  WinIni.Free;
end;
```

See also

ReadBool method, *ReadSection* method, *ReadString* method, *WriteBool* method

ReadKey function — WinCrt

Declaration

```
function ReadKey: Char;
```

The *ReadKey* function reads a character from the keyboard.

ReadKey supports only standard ASCII key codes. It does not support extended key codes, such as function and cursor keys codes.

Example

```
uses WinCrt;

var
  C: Char;

begin
  Writeln('Please press a key');
```

```
    C := Readkey;
    Writeln(' You pressed ', C, ', whose ASCII value is ', Ord(C), '.');
  end;
```

See also
KeyPressed function, *ReadBuf function*

ReadOnly property

Applies to
TBCDField, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TDBCheckBox*, *TDBComboBox*, *TDBEdit*, *TDBGrid*, *TDBImage*, *TDBListBox*, *TDBLookupCombo*, *TDBLookupList*, *TDBMemo*, *TDBRadioGroup*, *TEdit*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMaskEdit*, *TMemo*, *TMemoField*, *TSmallintField*, *TStringField*, *TTable*, *TTimeField*, *TVarBytesField*, *TWordField* components

For controls

Declaration

```
property ReadOnly: Boolean;
```

The *ReadOnly* property determines if the user can change the contents of the control. If *ReadOnly* is *True*, the user can't change the contents. If *ReadOnly* is *False*, the user can modify the contents. The default value is *False*.

For data-aware controls, the *ReadOnly* property determines whether the user can use the data-aware control to change the value of the field of the current record, or if the user can use the control only to display data. If *ReadOnly* is *False*, the user can change the field's value as long as the dataset is in edit mode.

When the *ReadOnly* property of a data grid is *True*, the user can no longer use the *Insert* key to insert a new row in the grid, nor can the user append a new row at the end of the data grid with the *Down Arrow* key.

R

Example
This code toggles the read-only state of an edit box each time the user double-clicks the form:

```
  procedure TForm1.FormActivate(Sender: TObject);
  begin
    Edit1.Left := 2;
    Edit1.Top := 2;
    Edit1.ReadOnly := True;
    Edit1.Text := 'Change Me';
    Canvas.TextOut(10, 40, 'Double-click form to toggle read-only state');
  end;

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

```
    Edit1.ReadOnly := not Edit1.ReadOnly;
  end;
```

See also

Alignment property, *EditMask* property, *Options* property, *Title* property, *Visible* property

For tables

Declaration

```
property ReadOnly: Boolean;
```

Use the *ReadOnly* property to prevent users from changing data in the table.

Note Set the *Active* property to *False* before changing *ReadOnly*.

Example

```
  Table1.Active := False;
  Table1.ReadOnly := True;
  Table1.Active := True;
```

See also

Exclusive property

For field components

Declaration

```
property ReadOnly: Boolean;
```

ReadOnly enables or disables modification of a field. If set to *False*, the default, a field can be modified. To prevent a field from being modified, set *ReadOnly* to *True*. In a *TDBGrid*, tabbing from field to field skips over *ReadOnly* fields.

ReadSection method

Applies to

TIniFile object

Declaration

```
procedure ReadSection (const Section: string; Strings: TStrings);
```

The *ReadSection* method reads all the variables of a section of an .INI file into a string object. The *Strings* parameter specifies the string list object. If you want to use a string list that is maintained by a component such as a list box, *Strings* should specify the property of the component that contains the string list. If you want to maintain the string list independent of any components, use a *TStringList* object.

The *Section* constant identifies the section of the .INI file that is read. For example, the WIN.INI for Windows contains a [Desktop] section.

Example

This example uses a list box on a form. When the application runs, all the entries in the Windows section of the WINI.INI file appear as items in the list box.

Before you run this example, you must put the *IniFiles* unit in the **uses** clause of your unit.

```
procedure TForm1.FormActivate(Sender: TObject);
var
  WinIni: TIniFile;
begin
  WinIni := TIniFile.Create('WIN.INI');
  WinIni.ReadSection('Windows', ListBox1.Items);
  WinIni.Free;
end;
```

See also

EraseSection method, *ReadBool* method, *ReadInteger* method, *ReadSectionValues* method, *ReadString* method, *WriteBool* method, *WriteInteger* method, *WriteString* method

ReadSectionValues method

Applies to

TIniFile object

Declaration

```
procedure ReadSectionValues(const Section: string; Strings: TStrings);
```

The *ReadSectionValues* method reads all the variables and their values of an entire section of an .INI file into a string object. You can then use the *Values* property of string and string list objects to access a specific string in the list of strings.

Example

This example reads the Transfer section of the DELPHI.INI file into a memo and changes one of the strings:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  DelphiIni: TIniFile;
begin
  DelphiIni := TIniFile.Create('c:\windows\delphi.ini');
  Memo1.Clear;
  DelphiIni.ReadSectionValues('Transfer', Memo1.Lines);
  Memo1.Lines.Values['Title1'] := 'Picture Painter';
  DelphiIni.Free;
end;
```

Although this example doesn't do so, your code could then write the new value to the DELPHI.INI file.

See also

EraseSection method, *ReadBool* method, *ReadInteger* method, *ReadSection* method, *ReadString* method, *Values* property, *WriteBool* method, *WriteInteger* method, *WriteString* method

ReadString method

Applies to

TIniFile object

Declaration

```
function ReadString(const Section, Ident, Default: string): string;
```

The *ReadString* method retrieves a string in an .INI file.

The *Section* constant identifies the section of the .INI file in which to search for the value. For example, the WIN.INI for Windows contains a [Desktop] section.

The *Ident* constant is the name of the identifier of which you want the value.

The *Default* constant is the default string value.

Example

This example reads strings in the DELPHI.INI file and displays them on the form.

Before you run this application, you must add the *IniFiles* unit to the **uses** clause of your unit.

```
procedure TForm1.FormActivate(Sender: TObject);
var
  DelphiIni: TIniFile;
begin
  Canvas.TextOut(20, 10, 'VARIOUS DELPHI SETTINGS');
  DelphiIni := TIniFile.Create('Delphi.Ini');
  with DelphiIni do
  begin
    with Canvas do
    begin
      TextOut(10, 50, 'Editor Font = ' +
        ReadString('Editor', 'FontName', 'ERROR'));
      TextOut(10, 70, 'Search Path = ' +
        ReadString('Library', 'SearchPath', 'ERROR'));
      TextOut(10, 90, 'Component Library = ' +
        ReadString('Library', 'ComponentLibrary', 'ERROR'));
      TextOut(10, 110, 'VBX Directory = ' +
        ReadString('VBX', 'VBXDir', 'ERROR'));
      TextOut(10, 130, 'VBX Unit Directory = ' +
        ReadString('VBX', 'UnitDir', 'ERROR'));
    end;
  end;
  DelphiIni.Free;
end;
```

See also

ReadBool method, *ReadInteger* method, *ReadSection* method, *WriteBool* method, *WriteInteger* method, *WriteString* method

ReAllocMem function **SysUtils**

Declaration

```
function ReAllocMem(P: Pointer; CurSize, NewSize: Cardinal): Pointer;
```

ReAllocMem re-allocates a block. On entry, *P* points to an existing heap block, *CurSize* gives the current size of the heap block, and NewSize specifies the requested new size of the block.

If *CurSize* is less than *NewSize*, the additional bytes in the new buffer are set to zero. The returned value is a pointer to the new block; this value is always different from the original pointer.

See also

AllocMem function

RecalcReport method

Applies to

TReport component

R

Declaration

```
function RecalcReport: Boolean;
```

The *RecalcReport* method recalculates and reprints the report with the new value for the report variable previously changed with the *SetVariable* method.

RecalcReport sends a DDE message to ReportSmith Runtime and looks for a DDE message in return. If *RecalcReport* returns *True*, the DDE message to recalculate the report was sent successfully to ReportSmith Runtime. If it returns *False*, ReportSmith Runtime could not receive the message at the current time.

For more information about report variables, see your ReportSmith documentation.

Example

The following code sets the 'FirstName' report variable to 'Marty', then recalculates the report.

```
Report1.SetVariable('FirstName', 'Marty');
if not (Report1.RecalcReport) then
  MessageDlg('Unable to recalculate', mtInformation, [mbOK] 0);
```

See also
Preview property, *Print* method, *Run* method, *SetVariable* method, *SetVariableLines* method

Rect function

Declaration

```
function Rect(ALeft, ATop, ARight, ABottom: Integer): TRect;
```

The *Rect* function returns a *TRect* record built from the individual coordinates passed in *ALeft*, *ATop*, *ARight*, and *ABottom*. You'll usually use *Rect* to construct parameters for functions that require *TRect*, rather than setting up local variables for each one.

Example
The following code defines the display rectangle for a media player component to be 100 pixels wide, 200 pixels tall, with a top-left corner at coordinates (10, 10);

```
MediaPlayer1.DisplayRect := Rect(10, 10, 110, 210);
```

See also
Point function

RecordCount property

Applies to
TBatchMove, *TQuery*, *TStoredProc*, *TTable* components

For batch move components

Declaration

```
property RecordCount: Longint;
```

The *RecordCount* property is used to control the maximum number of records that will be moved. If zero, all records are moved, beginning with the first record in *Source*. If *RecordCount* is not zero, a maximum of *RecordCount* records will be moved, beginning with the current record. If *RecordCount* exceeds the number of records remaining in *Source*, no wraparound occurs; the operation is terminated.

Example

```
{ Limit the move to the first 1000 records }
```

```
BatchMove1.RecordCount := 1000;
```

For tables, queries, and stored procedures

Declaration

```
property RecordCount: Longint;
```

Run-time and read only. The *RecordCount* property specifies the number of records in the *dataset*. The number of records reported may depend on the server and whether a range limitation is in effect.

Rectangle method

Applies to

TCanvas object

Declaration

```
procedure Rectangle(X1, Y1, X2, Y2: Integer);
```

The *Rectangle* method draws a rectangle on the canvas with its upper left corner at the point (*X1, Y1*) and its lower right corner at the point (*X2, Y2*). Rectangle draws the rectangle using the current brush (*TBrush*) and pen (*TPen*) attributes.

Example

This example draws many rectangles of various sizes and colors on a form maximized to fill the entire screen:

```
var
  X, Y: Integer;

procedure TForm1.FormActivate(Sender: TObject);
begin
  WindowState := wsMaximized;
  Canvas.Pen.Width := 5;
  Canvas.Pen.Style := psDot;
  Timer1.Interval := 50;
  Randomize;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  X := X + 4;
  Y := Y + 4;
  Canvas.Pen.Color := Random(65535);
  Canvas.Rectangle(X, Y, X + Random(400), Y + Random(400));
  if X > 700 then
    Timer1.Enabled := False;
end;
```

See also
RoundRect method

Refresh method

Applies to
All controls; *TTable*, *TQuery*, *TStoredProc* components

For all controls

Declaration

```
procedure Refresh;
```

The *Refresh* method erases whatever image is on the screen and then repaints the entire control. Within the implementation of *Refresh*, the *Invalidate* and then the *Update* methods are called.

Example
The following code refreshes all windowed controls of *Form1*, then refreshes *Form1*.

```
var
  I: Integer;

begin
  for I := 0 to Form1.ComponentCount-1 do
    if Form1.Components[i] is TWinControl then
      with Form1.Components[i] as TWinControl do
        Refresh;
  Form1.Refresh;
end;
```

See also
Repaint method

For tables, queries, and stored procedures

Declaration

```
procedure Refresh;
```

The *Refresh* method rereads all records from the *dataset*. Use *Refresh* to be certain that data controls display the latest information from the dataset. Calling *Refresh* may unexpectedly change the displayed data, potentially confusing the user.

RegisterFormAsOLEDropTarget procedure **ToCtrl**

Declaration

```
procedure RegisterFormAsOleDropTarget(Form: TForm; const Fmts: array of BOleFormat);
```

RegisterFormAsOLEDropTarget registers a form as a drag-and-drop target for OLE objects. The object formats in the *Fmts* array are registered so the objects can be dropped on the form. To register an OLE object format, you must declare a new Clipboard format for OLE objects with the Windows API function *RegisterClipboardFormat* prior to the call to *RegisterFormAsOLEDropTarget*.

Once a form is registered, the object formats which can be dropped can be modified with the *SetFormOLEDropFormats* procedure or deleted with the *ClearFormOLEDropFormats* procedure.

Example

The following code registers OLE formats for linked and embedded OLE objects. Then it creates a formats array for linked and embedded objects, as well as text. Finally,*Form1* is registered as an OLE drop target.

```
var
  FEmbedClipFmt, FLinkClipFmt: Word;
  Fmts: array[0..2] of BOLEFormat;

begin
  FEmbedClipFmt := RegisterClipboardFormat('Embedded Object');
  FLinkClipFmt := RegisterClipboardFormat('Link Source');
  Fmts[0].fmtId := FEmbedClipFmt;
  Fmts[0].fmtMedium := BOLEMediumCalc(FEmbedClipFmt);
  Fmts[0].fmtIsLinkable := False;
  StrPCopy (Fmts[0].fmtName, '%s');
  StrPCopy (Fmts[0].fmtResultName, '%s');
  Fmts[1].fmtId := FLinkClipFmt;
  Fmts[1].fmtMedium := BOLEMediumCalc(FLinkClipFmt);
  Fmts[1].fmtIsLinkable := True;
  StrPCopy (Fmts[1].fmtName, '%s');
  StrPCopy (Fmts[1].fmtResultName, '%s');
  Fmts[2].fmtId := CT_TEXT;
  Fmts[2].fmtMedium := BOLEMediumCalc(CF_TEXT);
  Fmts[2].fmtIsLinkable := False;
  StrPCopy (Fmts[2].fmtName, 'Text');
  StrPCopy (Fmts[2].fmtResultName, 'Text');
  RegisterFormAsOLEDropTarget(Self, Fmts);
end;
```

See also

TOLEDropNotify object

Release method

Applies to

TForm component

Declaration

```
procedure Release;
```

The *Release* method destroys the form and releases its associated memory. It is much like the *Free* method except that it does not destroy the form until all event handlers of the form or event handlers of components on the form have finished executing.

Example

This example displays a message box about the form going away, calls *Release*, and terminates the application.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MessageDlg('This form is going away forever', mtInformation, [mbOK], 0);
  Release;
  Application.Terminate;
end;
```

See also

Free method, *Destroy* method

Release procedure **System**

Declaration

```
procedure Release(var p: pointer);
```

The *Release* procedure returns the heap to a given state.

Release should not be used with *FreeMem* or *Dispose*.

Note *Release* is obsolete for Delphi applications.

Example

```
uses Crt;

var
   p : pointer;
   p1,p2,p3 : ^Integer;
 begin
   ClrScr;
   New(p1); { Allocate an Integer }
   Mark(p); { Save heap state }
   New(p2); { Allocate two more Integers }
```

```
    New(p3);
    Release(p); { Memory reserved for p2^ and p3^ has been released; p1^ may still be used}
  end;
```

See also
Dispose procedure, FreeMem procedure, GetMem procedure, Mark procedure, New procedure

ReleaseHandle method

Applies to
TBitmap object

Declaration

```
function ReleaseHandle: HBitmap;
```

The *ReleaseHandle* method returns the handle to the bitmap so that the *TBitmap* object no longer knows about the handle.

Example
The following code release the handle to the bitmap in *MyBitmap*.

```
MyBitmap.ReleaseHandle;
```

See also
ReleasePalette method

ReleaseOLEInitInfo procedure **ToCtrl**

Declaration

```
procedure ReleaseOleInitInfo(PInitInfo: Pointer);
```

ReleaseOLEInitInfo frees the memory allocated for OLE object initialization information. *ReleaseOLEInitInfo* should be called after calling the *InsertOLEObjectDlg* or *PasteSpecialDlg* functions to initialize a pointer to an OLE initialization information data structure. Pass the pointer initialized by *InsertOLEObjectDlg* or *PasteSpecialDlg* in the *PInitInfo* parameter of *ReleaseOLEInitInfo*.

Example
The following code uses *PasteSpecialDlg* to specify OLE initialization information. After *OLEContainer1* is initialized, the information is released. Fmts is assumed to be a valid array of *BOLEFormat* records.

```
var
  ClipFmt: Word;
  DataHand: THandle;
  Info: Pointer;
```

```
begin
  if PasteSpecialDlg(Form1, Fmts, 0, ClipFmt, DataHand, Info) then
  begin
    OLEContainer.PInitInfo := Info;
    ReleaseOLEInitInfo(Info);
  end;
end;
```

See also

PInitInfo property

ReleasePalette method

Applies to

TBitmap object

Declaration

```
function ReleasePalette: HPalette;
```

The *ReleasePalette* method returns the handle to the bitmap's palette so that the *TBitmap* object no longer knows about the palette.

Example

The following code release the palette of the bitmap in *MyBitmap*.

```
MyBitmap.ReleasePalette;
```

See also

ReleaseHandle method

Remove method

Applies to

TList object

Declaration

```
function Remove(Item: Pointer): Integer;
```

The *Remove* method deletes the item referenced in the *Item* parameter from the list of pointers stored in the *List* property of a list object. The value returned is the position of the item in the list of pointers before it was removed. After an item is removed, its position in the list is **nil**.

Example

The following code adds a new object to a list in a list object and then removes it:

```
type
  TMyClass = class
    MyString: string;
    constructor Create(S: string);
  end;

constructor TMyClass.Create(S: string);
begin
  MyString := S;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  MyList: TList;
  MyObject, SameObject: TMyClass;
begin
  MyList := TList.Create;                             { create the list }
  try
    MyObject := TMyClass.Create('Semper Fidelis!');  { create a class instance }
    try
      MyList.Add(MyObject);                           { add instance to list }
      SameObject := TMyClass(MyList.Items[0]);  { get first element in list }
      MessageDlg(SameObject.MyString, mtInformation, [mbOk], 0); { show it }
      MyList.Remove(MyObject);
      MessageDlg('Removing the object', mtInformation, [mbOk], 0);
    finally
      MyObject.Free;
    end;                                     { don't forget to clean up! }
  finally
    MyList.Free;
  end;
end;
```

See also

Delete method

RemoveAllPasswords method

Applies to

TSession component

Declaration

```
procedure RemoveAllPasswords;
```

The *RemoveAllPasswords* method causes all previously entered password information to be discarded. Any future access will require that new password information be supplied before the table can be opened. This method affects Paradox databases only.

Example

```
Session.RemoveAllPasswords;
```

See also

RemovePassword method, *Session* variable

RemoveComponent method

Applies to

All components

Declaration

```
procedure RemoveComponent(AComponent: TComponent);
```

The *RemoveComponent* method removes the component specified in the *AComponent* parameter from the component's *Components* list. That position in the list becomes **nil**.

Example

The following code removes *Button2* from the *Components* list of *Form1*.

```
Form1.RemoveComponent(Button2);
```

See also

InsertComponent method

RemoveControl method

Applies to

All controls

Declaration

```
procedure RemoveControl(AControl: TControl);
```

The *RemoveControl* method removes the control specified with the *AControl* parameter from the *Controls* array of this control. The result is that this control is no longer the parent of the removed control.

Example

This example uses a button placed alongside a group box. When the user clicks the button, the group box becomes the parent of the button, so the button moves inside the group box:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  RemoveControl(Button1);
  GroupBox1.InsertControl(Button1);
end;
```

Note that it was necessary to remove the button from the *Controls* property of the form before the button actually appears to move into the group box.

This code accomplishes the same thing:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Parent := GroupBox1;
end;
```

See also

Controls property, *InsertControl* method

RemoveParam method

Applies to

TParams object

Declaration

```
procedure RemoveParam(Value: TParam);
```

RemoveParam removes *Value* from the *Items* property.

Example

```
{ Move all parameter info from Params2 to Params1 }
while Params2.Count <> 0 do
  begin
{ Grab the first parameter from Params2 }
  TempParam := Params2[0];
{ Remove it from Params2 }
  Params2.RemoveParam(TempParam);
{ And add it to Params1 }
  Params1.AddParam(TempParam);
  end;
```

See also

AddParam method

RemovePassword method

Applies to

TSession component

Declaration

```
procedure RemovePassword(const Password: string);
```

The *RemovePassword* method removes *Password* from the known set of authorizations. Any future access will require that new password information be supplied before the table can be opened. This method affects Paradox databases only.

Example

```
Session.RemovePassword('MySecret');
```

See also

RemoveAllPasswords method, *Session* variable

Rename procedure — System

Declaration

```
procedure Rename(var F; Newname);
```

The *Rename* procedure changes the name of an external file.

F is a variable of any file type. *Newname* is a string-type expression or an expression of type *PChar* if the extended syntax is enabled.

The external file associated with *F* is renamed *Newname*. Further operations on *F* operate on the external file with the new name.

{$I+} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using **{$I–}**, you must use *IOResult* to check for I/O errors.

Example

```
uses Dialogs;

var
   f : file;
 begin
   OpenDialog1.Title := 'Choose a file... ';
   if OpenDialog1.Execute then begin
     SaveDialog1.Title := 'Rename to...';
     if SaveDialog1.Execute then begin
       AssignFile(f, OpenDialog1.FileName);
       Canvas.TextOut(5, 10, 'Renaming ' + OpenDialog1.FileName + ' to ' +
         SaveDialog1.FileName);
       Rename(f, SaveDialog1.FileName);
     end;
   end;
 end;
```

See also
Erase procedure

RenameFile function **SysUtils**

Declaration

```
function RenameFile(const OldName, NewName: string): Boolean;
```

The *RenameFile* function attempts to change the name of the file specified by *OldFile* to *NewFile*. If the operation succeeds, *RenameFile* returns *True*. If it cannot rename the file (for example, if a file called *NewName* already exists), it returns *False*.

Example
The following code renames a file:

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

See also
DeleteFile function

Repaint method

Applies to
All controls

Declaration

```
procedure Repaint;
```

The *Repaint* method forces the control to repaint its image on the screen, but without erasing what already appears there. To erase before repainting, call the *Refresh method* instead of *Repaint*.

Example
The following code repaints all windowed controls of *Form1*, then repaints *Form1*.

```
var
  I: Integer;

begin
  for I := 0 to Form1.ComponentCount-1 do
    if Form1.Components[I] is TWinControl then
      with Form1.Components[I] as TWinControl do
        Repaint;
  Form1.Repaint;
end;
```

See also
Refresh method

ReplaceText property

Applies to
TReplaceDialog component

Declaration

```
property ReplaceText: string;
```

The *ReplaceText* property contains the string your application can use to replace the string specified in the *FindText* property when the *FindText* value is found during a search.

Example
The following code replaces the selected text in *Memo1* with the value of *ReplaceText*.

```
   Memo1.SelText := ReplaceDialog1.ReplaceText;
```

See also
FindText property

ReportDir property

Applies to
TReport component

Declaration

```
property ReportDir: string;
```

The value of the *ReportDir* is the directory where ReportSmith stores its reports and expects to find saved reports. By specifying a report directory, you won't have to include a path when specifying a report name.

Example

The following text lets the user use the Save dialog box component to specify where ReportSmith saves its reports.

```
   if SaveDialog1.Execute then
     Report1.ReportDir := SaveDialog1.FileName;
```

See also
ReportName property

ReportHandle property

Applies to

TReport component

Declaration

```
property ReportHandle: HWND;
```

Run-time and read only. The value of the *ReportHandle* property is a Windows handle to ReportSmith.

Example

The following code retrieves the window placement information for ReportSmith, assuming *Report1* is a valid *TReport* component.

```
var
  RSWinPlacement: PWindowPlacement;

begin
  GetWindowPlacement(Report1.ReportHandle, RSWinPlacement);
end;
```

ReportName property

Applies to

TReport component

Declaration

```
property ReportName: string;
```

The value of the *ReportName* property determines which report you want to run. You can include a full path name as part of the report name if you have not specified a *ReportDir* property value or want to run a report that is stored elsewhere. If you have specified a *ReportDir* value, omit the path name and simply specify the name of the report.

Example

The following code lets users use the Open dialog box component to specify the report they want to run.

```
if OpenDialog1.Execute then
  Report1.ReportName := OpenDialog1.FileName;
```

See also

ReportDir property

RequestData method

Applies to

TDDEClientConv component

Declaration

```
function RequestData(const Item: string): PChar;
```

The *RequestData* method requests data from a DDE server. Call *RequestData* when you want your DDE client application to receive data from the server once, instead of being updated continually. Another reason to use *RequestData* is that some DDE servers contain DDE items that can't be continually updated; the only way for your client to access these items is to explicitly request the data.

Item specifies the DDE server item you want data from. The value of the DDE item depends on the linked DDE server application. *Item* is typically a selectable portion of text, such as a spreadsheet cell or a database field in an edit box. If the DDE server is an Delphi application, *Item* is the name of the linked DDE server component.

Note See the documentation for the DDE server application for the specific information about specifying DDEItem.

RequestData returns a null-terminated *PChar* string which contains the value of the item requested of the DDE server. *RequestData* automatically allocates memory to store this data, but you must dispose of the *PChar* string returned by *RequestData* after you have finished processing it. This is done with the *StrDispose* function.

Example

The following code requests data from the DDE server and displays it in *Label1*. The DDE item of the conversation is specified in the *DDEItem* property of *DDEClientItem1*.

```
var
  TheData: PChar;
begin
  TheData := DDEClientConv1.RequestData(DDEClientItem1.DDEItem);
  Label1.Caption := StrPas(TheData);
end;
```

See also

StrPas function

RequestLive property

Applies to

TQuery component

Declaration

```
property RequestLive: Boolean;
```

By default, a *TQuery* always returns a read-only result set. Set *RequestLive* to *True* to request a live result set. The BDE will then return a live result set if the SELECT syntax of the query conforms to the syntax requirements for a live result set. If *RequestLive* is *True*, but the syntax does not conform to the requirements, the BDE returns a read-only result set (for local SQL) or an error return code (for passthrough SQL). If a query returns a live result set, Delphi will set the *CanModify* property to *True*.

RequestLive	CanModify	Type of result set
False	*False*	Read-only result set
True—SELECT syntax meets requirements	*True*	Live result set
True—SELECT syntax does not meet requirements	*False*	Read-only result set

See also
Local property

Required property

Applies to
TFieldDef object; *TBCDField*, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

For field components

Declaration

```
property Required: Boolean;
```

Specifies whether a non-**nil** value for a field is required. The default value is *False*, meaning a field does not require a value. If a field is created with the Fields Editor, then this property is set based on the underlying table. Set *Required* to *True* for fields that must get values (for example, a password or part number), and write an *OnValidate* event handler for the field. Before a record is posted, exceptions are raised for any required fields that have **nil** values.

For TFieldDef objects

Declaration

```
property Required: Boolean;
```

Run-time and read only. Reports whether or not a value for a physical field in an underlying table is required.

Example

```
{ Is field required? }
if FieldDef1.Required) then
  MessageDlg(Name 'is a required field' , mtInformation, [mbOK], 0);
```

See also

TField component

Reset procedure **System**

Declaration

```
procedure Reset(var F [: File; RecSize: Word ] );
```

The *Reset* procedure opens an existing file.

F is a variable of any file type associated with an external file using *AssignFile*. *RecSize* is an optional expression, which can be specified only if *F* is an untyped file. If *F* is an untyped file, *RecSize* specifies the record size to be used in data transfers. If *RecSize* is omitted, a default record size of 128 bytes is assumed.

Reset opens the existing external file with the name assigned to *F*. An error results if no existing external file of the given name exists. If *F* is already open, it is first closed and then reopened. The current file position is set to the beginning of the file.

If *F* is assigned an empty name, such as *AssignFile*(*F*, ''), then after the call to *Reset*, *F* refers to the standard input file (standard handle number 0).

If *F* is a text file, *F* becomes read-only.

After a call to *Reset*, *Eof*(*F*) is *True* if the file is empty; otherwise, *Eof*(*F*) is *False*.

{$I+} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using **{$I–}**, you must use *IOResult* to check for I/O errors.

Example

```
function FileExists(FileName: string): Boolean;
{ Boolean function that returns True if the file exists; otherwise,
  it returns False. Closes the file if it exists. }
 var
  F: file;
begin
  {$I-}
  AssignFile(F, FileName);
  FileMode := 0;  ( Set file access to read only }
  Reset(F);
  CloseFile(F);
  {$I+}
  FileExists := (IOResult = 0) and (FileName <> '');
end;  { FileExists }

begin
  if FileExists(ParamStr(1)) then {Get file name from command line}
    Canvas.TextOut(10, 10, 'File exists')
  else
    Canvas.TextOut(10, 10, 'File not found');
end;
```

See also

Append procedure, AssignFile procedure, *FileClose procedure, Rewrite procedure, Truncate procedure*

Restore method

Applies to

TApplication component

Declaration

```
procedure Restore;
```

The *Restore* method returns your application to its previous size before it was maximized or minimized.

Don't confuse the *Restore* method with restoring a form or window to its original size. To minimize, maximize, and restore a window or form, you change the value of the *WindowState* property.

Example

This example uses a timer on a form. When the application runs and the user minimizes the application, the application returns to its normal size when an *OnTimer* event occurs:

```
  procedure TForm1.Timer1Timer(Sender: TObject);
  begin
    Application.Restore;
  end;
```

See also

BorderIcons property, *BorderStyle* property, *Minimize* method

RestoreTopMosts method

Applies to

TApplication component

Declaration

```
procedure RestoreTopMosts;
```

The *RestoreTopMosts* method restores forms that were originally designated as topmost forms (*FormStyle* is *fsStayOnTop*) and then temporarily changed to be non-topmost forms with the *NormalizeTopMosts* method call. After a call to *RestoreTopMosts*, the topmost forms move on top of other forms again.

Example

The following code normalizes topmost forms before calling the *MessageBox* function in the *WinProcs* unit. After the message box is closed, the topmost forms are restored.

```
begin
  Application.NormalizeTopMosts;
  MessageBox(Form1.Handle, 'This should be on top.', 'Message Box', MB_OK);
  Application.RestoreTopMosts;
end;
```

Resume method

Applies to

TMediaPlayer component

Declaration

```
procedure Resume;
```

The *Resume* method resumes playing or recording the currently paused multimedia device. *Resume* is called when the Pause button on the media player control is clicked at run time, when the device is paused.

Upon completion, *Resume* stores a numerical error code in the *Error* property, and the corresponding error message in the *ErrorMessage* property.

The *Wait* property determines whether control is returned to the application before the *Resume* method has completed. The *Notify* property determines whether *Resume* generates an *OnNotify* event.

Example

The following code resumes the playing or recording of *MediaPlayer1*.

```
MediaPlayer1.Resume;
```

See also

Pause method, *PauseOnly* method

Rewind method

Applies to

TMediaPlayer component

Declaration

```
procedure Rewind;
```

The *Rewind* method sets the current position to the beginning of the medium, which is stored in the *Start* property.

Upon completion, *Rewind* stores a numerical error code in the *Error* property, and the corresponding error message in the *ErrorMessage* property.

The *Wait* property determines whether control is returned to the application before the *Rewind* method has completed. The *Notify* property determines whether *Rewind* generates an *OnNotify* event.

Example

This example uses a media player and a button on a form. When the user clicks the button, the WAV audio media rewinds and begins playing. To run this example successfully, you must have installed a WAV audio device correctly.

```
procedure TForm1.FormClick(Sender: TObject);
begin
  MediaPlayer1.DeviceType := dtWaveAudio;
  FileName := ‘CHIMES.WAV’;
  Button1.Caption := 'Rewind and Play';
  Button1.Width := 130;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  MediaPlayer1.Rewind;
  MediaPlayer1.Play;
end;
```

See also

AutoRewind property, *Back* method

Rewrite procedure **System**

Declaration

```
procedure Rewrite(var F: File [; Recsize: Word ] );
```

The *Rewrite* procedure creates and opens a new file.

F is a variable of any file type associated with an external file using *AssignFile*. *RecSize* is an optional expression, which can be specified only if *F* is an untyped file. If *F* is an untyped file, *RecSize* specifies the record size to be used in data transfers. If *RecSize* is omitted, a default record size of 128 bytes is assumed.

Rewrite creates a new external file with the name assigned to *F*.

If an external file with the same name already exists, it is deleted and a new empty file is created in its place.

If *F* is already open, it is first closed and then re-created. The current file position is set to the beginning of the empty file.

If *F* was assigned an empty name, such as *AssignFile*(*F*,''), then after the call to *Rewrite*, *F* refers to the standard output file (standard handle number 1).

If *F* is a text file, *F* becomes write-only.

After calling *Rewrite*, *Eof*(*F*) is always *True*.

{$I+} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using **{$I–}**, you must use *IOResult* to check for I/O errors.

Example

```
var F: TextFile;
begin
  AssignFile(F, 'NEWFILE.$$$');
  Rewrite(F);
  Writeln(F, 'Just created file with this text in it...');
  CloseFile(F);
end;
```

See also

Append procedure, *AssignFile* procedure, *Reset procedure*, *Truncate procedure*

RmDir procedure — System

Declaration

```
procedure RmDir(S: string);
```

The *RmDir* procedure deletes an empty subdirectory.

RmDir removes the subdirectory with the path specified by *S*. If the path does not exist, is non-empty, or is the currently logged directory, an I/O error occurs.

{$I+} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using **{$I–}**, you must use *IOResult* to check for I/O errors.

Example

```
uses Dialogs;

begin
   {$I-}
   { Get directory name from TEdit control }
   RmDir(Edit1.Text);
   if IOResult <> 0 then
     MessageDlg('Cannot remove directory', mtWarning, [mbOk], 0)
   else
     MessageDlg('Directory removed', mtInformation, [mbOk], 0);
 end;
```

See also

ChDir procedure, GetDir procedure, MkDir procedure

Rollback method

Applies to

TDataBase component

Declaration

```
procedure Rollback;
```

The *Rollback* method rolls back the current transaction and thus cancels all modifications made to the database since the last call to *StartTransaction*. Use this method only when connected to a server database.

Example

```
with Database1 do
  begin
  StartTransaction;
{ Update one or more records in tables linked to Database1 }
...
  Rollback;
  end;
```

See also

Commit method

Round function System

Declaration

```
function Round(X: Real): Longint;
```

The *Round* function rounds a real-type value to an integer-type value.

X is a real-type expression. *Round* returns a *Longint* value that is the value of *X* rounded to the nearest whole number. If *X* is exactly halfway between two whole numbers, the result is the number with the greatest absolute magnitude.

If the rounded value of *X* is not within the *Longint* range, you will generate a run-time error, which you can handle using the *EInvalidOp* exception.

Example

```
var
   S, T: string;

begin
   Str(1.4:2:1, T);
   S := T + ' rounds to ' + IntToStr(Round(1.4)) + #13#10;
   Str(1.5:2:1, T);
   S := S + T + ' rounds to ' + IntToStr(Round(1.5)) + #13#10;
   Str(-1.4:2:1, T);
   S := S + T + ' rounds to ' + IntToStr(Round(-1.4)) + #13#10;
   Str(-1.5:2:1, T);
   S := S + T + ' rounds to ' + IntToStr(Round(-1.5));
   MessageDlg(S, mtInformation, [mbOk], 0);
end;
```

See also
Int function, *Trunc function*

RoundRect method

Applies to
TCanvas object

Declaration

```
procedure RoundRect(X1, Y1, X2, Y2, X3, Y3: Integer);
```

The *RoundRect* method draws a rectangle on a canvas with the upper left corner at (*X1*, *Y1*) and the lower right corner at (*X2*, *Y2*), much as the *Rectangle method* does. However, *RoundRect* draws the corners as quarters of an ellipse with the width of *X3* and a height of *Y3*.

Example
This example draws many rectangles of various sizes and colors on a form maximized to fill the entire screen:

```
var
  X, Y: Integer;

procedure TForm1.FormActivate(Sender: TObject);
```

```
begin
  WindowState := wsMaximized;
  Canvas.Pen.Width := 5;
  Canvas.Pen.Style := psDot;
  Timer1.Interval := 50;
  Randomize;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  X := X + 4;
  Y := Y + 4;
  Canvas.Pen.Color := Random(65535);
  Canvas.RoundRect(X, Y, X + Random(400), Y + Random(400), 10, 10);
  if X > 700 then
    Timer1.Enabled := False;
end;
```

See also
Ellipse method, *Rectangle* method

Row property

Applies to
TDrawGrid, *TOutline*, *TStringGrid* components

Declaration

```
property Row: Longint;
```

Run-time only. The value of the *Row* property indicates which row of the control has focus. For outlines, you can use the *Row* property to determine which item is selected at run time. For the grid components, you can use *Row* along with the *Col* property to determine which cell is selected.

Example
This examples uses a string grid with a label above it on a form. When the user clicks a cell in the grid, the location of the cursor is displayed in the caption of the label.

```
procedure TForm1.StringGrid1Click(Sender: TObject);
begin
  Label1.Caption := 'The cursor is in column ' + IntToStr(StringGrid1.Col + 1)
     +  ', row ' + IntToStr(StringGrid1.Row + 1);
end;
```

See also
DefaultRowHeight property, *RowCount* property, *RowHeights* property

RowCount property

Applies to
TDrawGrid, *TStringGrid* components

Declaration

```
property RowCount: Longint;
```

The value of the *RowCount* property determines the number of rows that appear in the grid.

Example
This example uses a string grid and a button. When the user clicks the button, the number of columns and rows change:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  StringGrid1.ColCount := 7;
  StringGrid1.RowCount := 11;
end;
```

See also
ColCount property, *Row* property, *RowHeights* property

RowHeights property

Applies to
TDrawGrid, *TStringGrid* components

Declaration

```
property RowHeights[Index: Longint]: Integer;
```

Run-time only. The *RowHeights* property determines the height in pixels of all the cells within the row referenced by the *Index* parameter.

By default, all the rows are the same height, the value found in the *DefaultRowHeight* property. To change the height of all rows within a grid, change the *DefaultRowHeight* property value.

To change the height of one row without affecting any others, change the *RowHeights* property. Specify the row you want to change as the value of the *Index* parameter. Remember, the first row always has an *Index* value of 0.

Example
This example uses a string grid and a button. When the user clicks the button, the number of columns and rows change, and the first column and row in the grid are sized differently from the rest of the columns and rows:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
with StringGrid1 do
  begin
    ColCount := 7;
    RowCount := 11;
    RowHeights[0] := 35;
    ColWidths[0] := 90;
  end;
end;
```

See also
ColWidths property, *DefaultColWidth* property, *Row* property

Rows property

Applies to
TStringGrid component

Declaration

```
property Rows[Index: Integer]: TStrings;
```

Run-time only. The *Rows* property is an array of the strings and their associated objects in a row. The number of strings and associated objects is always the value of the *RowCount* property, the number of rows in the grid. Use the *Rows* property to access the strings and their associated objects within a particular row in the grid. The *Index* parameter is the number of the row you want to access, with the first row having an *Index* value of zero.

Example
This example displays a string grid, a list box, and a button on a form. When the application runs, strings are put in the cells of row 1 of the string grid. When the user clicks the button, each string in row 1 appears as an item in the list box.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items := StringGrid1.Rows[1];
  Button1.Enabled := False;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  with StringGrid1 do
  begin
    Cells[1,1] := 'Object';
    Cells[2,1] := 'Pascal';
    Cells[3,1] := 'is';
    Cells[4,1] := 'excellent';
  end;
```

```
end;
```

Note that the first position of the list box is empty, because there is no string in the cell referenced by *Cells[0, 1]*.

See also

Cells property, *Cols* property, *Objects* property, *TStrings* object

Run method

Applies to

TApplication, *TReport* components

For an application

Declaration

```
procedure Run;
```

The *Run* method executes the application.

When you create a new project, Delphi automatically creates a main program block in the project file that calls the *Run* method.

Example

The main program block of a Delphi project always looks like this, by default:

```
begin
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

See also

CreateForm method, *MainForm* property

For reports

Declaration

```
procedure Run;
```

The *Run* method loads ReportSmith Runtime, runs the report specified as the value of the *ReportName* property, and prints the report.

Example

The following code allows the user to specify the report with the Open dialog box and then runs the report.

```
if OpenDialog1.Execute then
begin
  Report1.ReportName := OpenDialog1.FileName;
  Report1.Run;
end;
```

See also

Preview property, *Print* method, *ReportName* property

RunError procedure **System**

Declaration

```
procedure RunError [ ( Errorcode: Byte ) ];
```

The *RunError* procedure stops program execution by generating a run-time error with the given number at the current statement.

Errorcode is the run-time error number (0 if omitted). If you compile the current module with debug information on, and run the program within Delphi, Delphi automatically takes your *RunError* call.

Example

```
begin
{$IFDEF Debug}
  if P = nil then
    RunError(204);
{$ENDIF}
end;
```

See also

Exit procedure, *Halt* procedure

RunMacro method

Applies to

TReport component

Declaration

```
function RunMacro(Macro: string): Boolean;
```

The *RunMacro* method runs the ReportBasic macro specified as the value of the*Macro* parameter.

The *RunMacro* method sends a DDE message to ReportSmith Runtime to run the specified macro and looks for a DDE message from ReportSmith Runtime in return. If *RunMacro* returns *True*, the message to run the macro was sent successfully to

ReportSmith Runtime. If it returns *False*, ReportSmith Runtime could not receive the DDE message at the current time.

For information about ReportBasic macros, refer to your ReportSmith documentation.

Example

The following code runs the macro “SELALL.MAC”.

```
Report1.RunMacro('SELALL.MAC');
```

See also

Run method

Save method

Applies to

TMediaPlayer component

Declaration

```
procedure Save;
```

The *Save* method saves the currently loaded medium to the file specified in the *FileName* property. *Save* is ignored for devices that don’t use media stored in files (videodiscs, for example).

Upon completion, *Save* stores a numerical error code in the *Error* property, and the corresponding error message in the *ErrorMessage* property.

The *Wait* property determines whether control is returned to the application before the *Save* method has completed. The *Notify* property determines whether *Save* generates an *OnNotify* event.

Example

The following code saves to a file when the *SaveButton* is clicked, assuming that *FileName* has been specified.

```
procedure TForm1.SaveButtonClick(Sender: TObject);
begin
  MediaPlayer1.Save;
end;
```

See also

Close method, *Open* method

SaveToFile method

Applies to

TBitmap, *TBlobField*, *TGraphic*, *TGraphicField*, *TIcon*, *TMemoField*, *TMetafile*, *TPicture*, *TStringList*, *TStrings* objects; *TOLEContainer*, *TOutline* components

Declaration

```
procedure SaveToFile(const FileName: string);
```

The *SaveToFile* method saves an object to the file specified in *FileName.* The graphic objects save a graphic to the file, the OLE container saves an OLE object to the file, the outline and string objects save text to the file, and the field components save the contents of the field to the file.

Example

This example uses a memo control and two buttons on a form. When the application runs, the code attempts to load text in a text file named SOMETEXT.TXT into the memo control. If the attempt fails, a message box appear, prompting the user to enter text in the memo.

The user must choose one of the two buttons, as there is no System menu (also called a Control menu) on the form to close the form. If the user chooses the button labeled Save, the contents of the memo control is saved in a file named SOMETEXT.TXT, and the form closes. If the user chooses the button labeled Discard, the lines in the memo control are not saved, and the form closes.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  BorderIcons := [];
  Button1.Caption := 'Save';
  Button2.Caption := 'Discard';
  try
    Memo1.Lines.LoadFromFile('SOMETEXT.TXT');
  except
    Memo1.Lines.Clear;
    MessageDlg('When form appears, type in the memo control',
      mtInformation, [mbOk], 0);
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.Lines.SaveToFile('SOMETEXT.TXT');
  Close;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;
```

This example stored a blob field into a file:

```
{ Store a blob field into a temporary file }
BlobField1.SaveToFile('c:\windows\temp\myblob.blb');
```

See also

LoadFromFile method, *LoadFromStream* method, *SaveToStream* method

SaveToStream method

Applies to

TBlobField, *TGraphicField*, *TMemoField* components

Declaration

```
procedure SaveToStream(Stream: TStream);
```

The *SaveToStream* method writes a stream with the name passed in *Stream* with the contents of *TBlobField*, *TMemoField*, or *TGraphicField*.

Example

```
{ Store a blob field into a stream }
BlobField1.SaveToStream(Stream1);
```

See also

LoadFromFile method, *LoadFromStream* method, *SaveToFile* method

ScaleBy method

Applies to

All controls

Declaration

```
procedure ScaleBy(M, D: Integer);
```

Description

The *ScaleBy* method scales a control to a percentage of its former size. The *M* parameter is the multiplier and the *D* parameter is the divisor. For example, if you want a control to be 75% of its original size, specify the value of *M* as 75, and the value of *D* as 100 (75/100). You could also obtain the same results by specifying the value of *M* as 3, and the value of *D* as 4 (3/4). Both fractions are equal and result in the control being scaled by the same amount, 75%.

If you want the control to be 33% larger than its previous size, specify the value of *M* as 133, and the value of *D* as 100 (133/100). You can also obtain the same results by specifying the value of *M* as 4, and the value of *D* as 3 (4/3), as the fraction 133/100 is approximately equal to 4/3.

Example

This example makes your form 50% larger:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ScaleBy(150, 100);
end;
```

See also

Scaled property

Scaled property

Applies to

TForm component

Declaration

```
property Scaled: Boolean;
```

The *Scaled* property determines if the form is scaled to the value in the *PixelsPerInch* property. If *Scaled* is *True* and the value of *PixelsPerInch* differs from its default value, the form is scaled to a new size. If *Scaled* is *False*, no scaling occurs, regardless of the *PixelsPerInch* value. The default value is *True*.

Example

This code ensures that the form is always scaled to whatever value is in the *PixelsPerInch* property:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Scaled := True;
end;
```

See also

PixelsPerInch property, *ScaleBy* method

Screen variable **Forms**

Declaration

```
Screen: TScreen;
```

The *Screen* variable is a *TScreen component* that normally represents your screen device. By default, your application creates a screen component based on information from Windows about the current screen device and assigns it to *Screen*.

Example

The following code sets the width of a form called *Form1* to half the width of the screen:

```
Form1.Width := Screen.Width div 2;
```

ScreenSize typed constant — WinCrt

Declaration

```
const ScreenSize: TPoint = (X: 80; Y: 25);
```

The *ScreenSize* typed constant determines the width and height in characters of the virtual screen within the CRT window.

The default screen size is 80 columns by 25 lines.

You can change the size of the virtual screen by assigning new values to the x- and y-coordinates of *ScreenSize* before the CRT window is created.

The value given by *ScreenSize.X* multiplied by *ScreenSize.Y* must not exceed 65,520.

ScreenToClient method

Applies to

All controls

Declaration

```
function ScreenToClient(Point: TPoint): TPoint;
```

The *ScreenToClient* method is used to determine the control coordinates in pixels of a point on the screen. *ScreenToClient* returns X and Y coordinates in a record of type *TPoint*.

Example

The following code converts the origin of the screen (0, 0) to the client coordinates of *Button2*.

```
var
  ScreenOrgin, ClientPoint: TPoint;
begin
  ScreenOrgin.X := 0;
  ScreenOrgin.Y := 0;
  ClientPoint := Button2.ScreenToClient(ScreenOrgin);
end;
```

See also

ClientToScreen method

ScrollBars property

Applies to

TDBMemo, *TDrawGrid*, *TMemo*, *TStringGrid* components

Declaration

```
property ScrollBars: TScrollStyle;
```

The *ScrollBars* property controls whether a memo control or a grid control has any scroll bars. You can set *ScrollBars* to any of the following values:

Value	Meaning
ssNone	No scroll bar
ssHorizontal	Puts a scroll bar on the right edge
ssVertical	Puts a scroll bar on the bottom edge
ssBoth	Puts a scroll bar on both the right and bottom edges

By default, grids have both vertical and horizontal scroll bars, while memo controls have none.

Example

The following example adds a scroll bar to the bottom of memo control *Memo1*:

```
  Memo1.ScrollBars := scHorizontal;
```

See also

HorzScrollBar property, *VertScrollBar* property

ScrollBy method

Applies to

All controls; *TForm* component

Declaration

```
procedure ScrollBy(DeltaX, DeltaY: Integer);
```

The *ScrollBy* method scrolls the contents of a form or windowed control. You will seldom need to call the *ScrollBy* method unless you want to write your own scrolling logic rather than use a scroll bar.

The *DeltaX* parameter is the change in pixels along the X axis. A positive *DeltaX* value scrolls the contents to the right; a negative value scrolls the contents to the left. The

DeltaY parameter is the change in pixels along the Y axis. A positive *DeltaY* value scrolls the contents down; a negative value scrolls the contents up.

Example

This example uses a timer and several controls of your choosing on a form. When the application runs, the controls on the form appear to slide down and off to the right. This is because the contents of the form are scrolling both down and to the right by one pixel each time a timer event occurs:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  Timer1.Interval := 1;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  ScrollBy(1,1);
end;
```

See also

HorzScrollBar property, *ScrollBars* property, *ScrollInView* method, *TScrollBox* component, *VertScrollBar* property

ScrollInView method

Applies to

TForm, *TScrollBox* components

Declaration

```
procedure ScrollInView(AControl: TControl);
```

The *ScrollInView* method scrolls the form or scroll box so that at least part of the control specified as the *AControl* parameter is in view.

Example

This example uses two buttons on a form. Place each button on opposite sides of the form. When the user runs the application and resizes the form so that it is smaller, clicking either of the buttons scrolls the form so that at least part of the other button is visible:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ScrollInView(Button2);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  ScrollInView(Button1);
end;
```

See also

HorzScrollBar property, *VertScrollBar* property

ScrollPos property

Applies to

TControlScrollBar component

Declaration

```
property ScrollPos: Integer
```

Run-time and read only. The value of the *ScrollPos* property is the current *Position* value of a horizontal or vertical scroll bar on a form or scroll box.

Example

This example uses a label on a form. When the application runs, resize the form so a horizontal scroll bar appears. You can use the scroll bar to scroll the form. Each time you click the label, it reports the current position of the scroll bar:

```
procedure TForm1.Label1Click(Sender: TObject);
begin
  Label1.Caption := 'Scroll bar position is ' + IntToStr(HorzScrollBar.ScrollPos);
end;
```

See also

HorzScrollBar property, *Position* property, *VertScrollBar* property

ScrollTo procedure — WinCrt

Declaration

```
procedure ScrollTo(X, Y: Integer);
```

The *ScrollTo* procedure scrolls the CRT window to show the virtual screen location (*X*,*Y*) in the upper left corner.

The upper left corner of the virtual screen corresponds to (0,0).

Example

```
uses WinCrt;

begin
  GotoXY(1,10);
  Writeln('Hello');
  Writeln('Type in a line and press Enter.');
  Readln;
  ScrollTo(0,10);
end;
```

See also

GoToXY procedure

Sections property

Applies to

THeader component

Declaration

```
property Sections: TStrings
```

The *Sections* property is a list of strings that contain the text for the sections of a header. The number of lines of the string list determines the number of sections of the header. If the string list is empty, the header will have one blank section. If this string list contains one or more lines, the text of each line will be in its own section. The first line will be in the leftmost section, the second line will be in the next section to the right, and so on.

Example

The following code adds ‘Schaeferle’ to the header sections list.

```
Header1.Sections.Add(‘Schaeferle’);
```

See also

SectionWidth property

SectionWidth property

Applies to

THeader component

Declaration

```
property SectionWidth[X: Integer]: Integer;
```

Run-time only. The *SectionWidth* array property determines the width in pixels of the sections of a header. X is an index into the sections, from 0 to the number of sections - 1. For example, the index of the first section is 0, the second section is 1, and so on.

Example

The following code doubles the width of all the sections of a header.

```
var
  I: Integer;
begin
  with Header1 do
    for I := Sections.Count-1 do
    SectionWidth[I] := SectionWidth[I] * 2;
end;
```

See also

Sections property

Seek method

Applies to

TBlobStream object

Declaration

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

The *Seek* function resets the current position within the *TBlobStream*. If *Origin* is 0, the new position is *Offset* (seek absolute). If *Origin* is 1, the new position is *Position* + *Offset* (seek relative). If *Origin* is 2, the new position is *Size* + *Offset* (seek absolute from end of data). *Seek* returns the new position, relative to the beginning of the BLOB stream.

Note When *Origin* is 0, *Offset* must be >= 0. When *Origin* is 2, *Offset* must be <= 0.

Example

```
{ Move to the end of the data so we can add more to it }
BlobStream1.Seek(0, 2);
```

See also

TBlobField component, *TBytesField* component, *TVarBytesField* component

Seek procedure **System**

Declaration

```
procedure Seek(var F; N: Longint);
```

The *Seek* procedure moves the current position of a file to a specified component. You can use *Seek* only on open typed or untyped files.

In the above syntax, *F* is a typed or untyped file variable, and *N* is an expression of type *Longint*.

The current file position of *F* moves to component number *N*. The number of the first component of a file is 0.

To expand a file, you can seek one component beyond the last component; that is, the statement *Seek*(*F*, *FileSize*(*F*)) moves the current file position to the end of the file.

{$I+} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using **{$I–}**, you must use *IOResult* to check for I/O errors.

Example

```
uses Dialogs;

var
   f: file of Byte;
   size : Longint;
   S: string;
   y: integer;
 begin
   if OpenDialog1.Execute then begin
     AssignFile(f, OpenDialog1.FileName);
     Reset(f);
     size := FileSize(f);
     S := 'File size in bytes: ' + IntToStr(size);
     y := 10;
     Canvas.TextOut(5, y, S);
     y := y + Canvas.TextHeight(S) + 5;
     S := 'Seeking halfway into file...';
     Canvas.TextOut(5, y, S);
     y := y + Canvas.TextHeight(S) + 5;
     Seek(f,size div 2);
     S := 'Position is now ' + IntToStr(FilePos(f));
     Canvas.TextOut(5, y, S);
     CloseFile(f);
   end;
 end;
```

See also
FilePos function

SeekEof function — System

Declaration

```
function SeekEof [ (var F: Text) ]: Boolean;
```

The *SeekEof* function returns the end-of-file status of a file.

SeekEof can only be used on open text files.

{$I+} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using **{$I–}**, you must use *IOResult* to check for I/O errors.

Example

```
var
   f : System.TextFile;
```

```
    i, j, Y : Integer;
  begin
    AssignFile(f,'TEST.TXT');
    Rewrite(f);
    { Create a file with 8 numbers and some
      whitespace at the ends of the lines }
    Writeln(f,'1 2 3 4 ');
    Writeln(f,'5 6 7 8 ');
    Reset(f);
    { Read the numbers back. SeekEoln returns TRUE if there are no more numbers on the
      current line; SeekEof returns TRUE if there is no more text (other than whitespace)
      in the file. }
    Y := 5;
    while not SeekEof(f) do
    begin
      if SeekEoln(f) then
        Readln; { Go to next line }
      Read(f,j);
      Canvas.TextOut(5, Y, IntToStr(j));
      Y := Y + Canvas.TextHeight(IntToStr(j)) + 5;
    end;
  end;
```

See also

Eof function, *SeekEoln* function

SeekEoln function — System

Declaration

```
function SeekEoln [ (var F: Text) ]: Boolean;
```

The *SeekEoln* function returns the end-of-line status of a file.

SeekEoln can be used only on open text files.

{$I+} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using **{$I–}**, you must use *IOResult* to check for I/O errors.

Example

```
 var
   f : System.TextFile;
   i, j, Y : Integer;
  begin
   AssignFile(f,'TEST.TXT');
   Rewrite(f);
   { Create a file with 8 numbers and some
     whitespace at the ends of the lines }
   Writeln(f,'1 2 3 4 ');
```

```
    Writeln(f,'5 6 7 8 ');
    Reset(f);
    { Read the numbers back. SeekEoln returns TRUE if there are no more numbers on the
      current line; SeekEof returns TRUE if there is no more text (other than whitespace)
      in the file. }
    Y := 5;
    while not SeekEof(f) do
    begin
      if SeekEoln(f) then
        Readln; { Go to next line }
      Read(f,j);
      Canvas.TextOut(5, Y, IntToStr(j));
      Y := Y + Canvas.TextHeight(IntToStr(j)) + 5;
    end;
  end;
```

See also

Eoln function, *SeekEof* function

Seg function — System

Declaration

```
function Seg(X): Word;
```

The *Seg* function returns the segment of a specified object.

X is any variable, or a procedure or function identifier. The result is the segment part of the address of *X*.

Example

```
function MakeHexWord(w: Word): string;
  const
    hexChars: array [0..$F] of Char =
      '0123456789ABCDEF';
var
    HexStr : string;
  begin
    HexStr := '';
    HexStr := HexStr + hexChars[Hi(w) shr 4];
    HexStr := HexStr + hexChars[Hi(w) and $F];
    HexStr := HexStr + hexChars[Lo(w) shr 4];
    HexStr := HexStr + hexChars[Lo(w) and $F];
    MakeHexWord := HexStr;
  end;

var
  i: Integer;
  Y: Integer;
  S: string;
```

```
begin
  Y := 10;
  S := 'The current code segment is $' + MakeHexWord(CSeg);
  Canvas.TextOut(5, Y, S);

Y := Y + Canvas.TextHeight(S) + 5;
  S := 'The global data segment is $' + MakeHexWord(DSeg);
  Canvas.TextOut(5, Y, S);

Y := Y + Canvas.TextHeight(S) + 5;
  S := 'The stack segment is $' + MakeHexWord(SSeg);
  Canvas.TextOut(5, Y, S);

Y := Y + Canvas.TextHeight(S) + 5;
  S := 'The stack pointer is at $' + MakeHexWord(SPtr);
  Canvas.TextOut(5, Y, S);

Y := Y + Canvas.TextHeight(S) + 5;
  S := 'i is at offset $' + MakeHexWord(Ofs(i));
  Canvas.TextOut(5, Y, S);

Y := Y + Canvas.TextHeight(S) + 5;
  S := 'in segment $' + MakeHexWord(Seg(i));
  Canvas.TextOut(5, Y, S);
end;
```

See also
Addr function, *Ofs* function

SelCount property

Applies to
TDBListBox, *TDirectoryListBox*, *TFileListBox*, *TListBox* components

Declaration

```
property SelCount: Integer;
```

Run-time and read only. The *SelCount* property reports the number of items that are selected in a list box when the value of the *MultiSelect* property is *True*. When *MultiSelect* property is *False*, only one item can be selected. If no items are selected, the value of *SelCount* is 0.

Example
This example uses a list box, a label, and a button on a form. Enter several strings in the list box as the value of the *Items* property. When the user selects items in the list box and clicks the button, the number of items selected in the list box is displayed in the caption of the label:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := IntToStr(ListBox1.SelCount) + ' items are selected';
  end;
```

See also
ExtendedSelect property, *MultiSelect* property, *Selected* property

SelectAll method

Applies to
TComboBox, *TDBComboBox*, *TDBEdit*, *TDBMemo*, *TDriveComboBox*, *TEdit*, *TFilterComboBox*, *TMaskEdit*, *TMemo* components

Declaration

```
procedure SelectAll;
```

The *SelectAll* method selects the entire block of text in the control. If you want to select only part of the text, use the *SelStart* and *SelLength* properties.

Example
The following code selects all the text in *Memo1*.

```
  Memo1.SelectAll;
```

See also
SelLength property, *SelStart* property, *SelText* property, *Text* property

SelectDirectory function **FileCtrl**

Declaration

```
function SelectDirectory(var Directory: string; Options: TSelectDirOpts; HelpCtx: Longint):
Boolean;
```

The *SelectDirectory* function lets the user enter a directory name into the application using a Select Directory dialog box. Calling the *SelectDirectory* function displays the Select Directory dialog box. The directory passed to the function with the *Directory* parameter is the currently selected directory when the dialog box appears. The name of the directory the user selects becomes the value of *Directory* when the function returns.

The *Options* parameter is a set of values. These are the possible values of the Options set:

Value	Meaning
[ ], the empty set	The user can select a directory that currently exists only. The user cannot specify a directory that does not exist as there is no edit box to type in a new directory name.
sdAllowCreate	An edit box appears in the dialog box so that the user can type in the name of a directory that does not exist. This option does not create a directory, but the application can access the *Directory* parameter to create the directory selected if desired.

Value	Meaning
sdPerformCreate	Used only when the *Options* set contains *sdAllowCreate*. If the user enters a directory name that does not exist, the function creates the directory.
sdPrompt	Used when the *Options* set contains *sdAllowCreate*. Displays a message box the informs the user the entered directory does not exist and asks the user if the directory should be created. If the user chooses OK, the directory is created only if the *Options* set contains the *sdPerformCreate* value. If only *sdAllowCreate* is *True*, the directory is not actually created, but the application can create it if desired.

The function returns *True* if the user selected a directory and chose OK, and *False* if the user chose Cancel or closed the dialog box without selecting a directory.

The *HelpCtx* parameter is the help context ID number.

Example

This example uses a button on a form. When the user clicks the button, a Select Directory dialog box appears. The current directory displayed in the dialog box is C:\LINDA. The user can select a directory from the directory list, or enter a new directory in the edit box. If the user enters a new directory, a message box asks the user if the directory should be created. If the user chooses Yes, the directory is created. If the user chooses No, the message box goes away and the user can use the dialog box again to select a directory. The name of the directory the user selects appears as the caption of the label:

```
uses FileCtrl;

procedure TForm1.Button1Click(Sender: TObject);
var
  Dir: string;
begin
  Dir := 'C:\LINDA';
  if SelectDirectory(Dir, [sdAllowCreate, sdPerformCreate, sdPrompt]) then
    Label1.Caption := Dir;
end;
```

See also

DirectoryExists function, *ForceDirectories* procedure

Selected property

Applies to

TDBListBox, *TDirectoryListBox*, *TFileListBox*, *TListBox* components

Declaration

```
property Selected[X: Integer]: Boolean;
```

The *Selected* property determines whether a particular item is selected in the list box. The *X* parameter is the item referenced by its position in the list box, with the first item having an *X* value of 0. If the specified item is selected in the list box, the value of the *Selected* property is *True*. If the specified item is not selected, *Selected* is *False*.

If you want the user to be able to select more than one item in the list box, use the *MultiSelect* property

Example

This example uses a list box on a form. When the form is first created, 3 items are added to the list box. When the user selects an item in the list box, the list box color changes to reflect the item selected:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Items.Add('Blue');
  ListBox1.Items.Add('Yellow');
  ListBox1.Items.Add('Red');
end;

procedure TForm1.ListBox1Click(Sender: TObject);
begin
  if ListBox1.Selected[0] then
    ListBox1.Color := clBlue;
  if ListBox1.Selected[1] then
    ListBox1.Color := clYellow;
  if ListBox1.Selected[2] then
    ListBox1.Color := clRed;
end;
```

See also

ExtendedSelect property, *MultiSelect* property, *SelCount* property

SelectedColor property

Applies to

TTabSet component

Declaration

```
property SelectedColor: TColor;
```

The *SelectedColor* property determines the color of the selected tab in the tab set control. To view a list of available color values, see the *Color* property.

Example

This code changes the color of selected tabs:

```
TabSet11.SelectedColor := clPurple;
```

See also
UnselectedColor property

SelectedField property

Applies to
TDBGrid, *TDBLookupList* components

Declaration

```
property SelectedField: TField;
```

Run-time and read only. The value of the *SelectedField* property indicates which field is selected in the data grid.

Example
The following code displays the name of the selected field in a label if the selected field is 'CustNo'.

```
if DBGrid1.SelectedField.FieldName = 'CustNo' then
  Label1.Caption := DBGrid1.SelectedField.FieldName;
```

See also
SelectedIndex property

SelectedIndex property

Applies to
TDBGrid, *TDBLookupList* components

Declaration

```
property SelectedIndex: Integer;
```

Run-time only. The value of the *SelectedIndex* property returns the index value of the currently selected field in the displayed dataset. A value of 0 indicates the first field of the displayed dataset, 1 is the second field, and so on.

SelectedIndex can be used as an index to *Fields* property array to access a field in the dataset.

Example
The following code makes all the fields up to the selected field of the dataset of*DBGrid1* read-only. *I* is an integer variable.

```
for I := 0 to DBGrid1.SelectedIndex do
  DBGrid1.Fields[I].ReadOnly := True;
```

See also
SelectedField property

SelectedItem property

Applies to
TOutline component

Declaration

```
property SelectedItem: Longint;
```

Run-time only. The *SelectedItem* property determines which item of the outline currently has focus. *SelectedItem* contains the *Index* value of the selected item. If no item is selected, *SelectedItem* contains 0.

Example
The following code expands the selected item of *Outline1*.

```
Outline1.Items[Outline1.SelectedItem].FullExpand;
```

See also
Items property

Selection property

Applies to
TDrawGrid, TStringGrid components

Declaration

```
property Selection: TGridRect;
```

The *Selection* property contains the column and row coordinates of the cell or cells selected in the grid.

Example
The following code selects the rectangle containing rows 1 and 2, and columns 3 and 4.

```
var
  SRect: TGridRect;
begin
  SRect.Top := 1;
  SRect.Left := 3;
  SRect.Bottom := 2;
  SRect.Right := 4;
  StringGrid1.Selection := SRect;
end;
```

SelectNext method

Applies to

TTabSet component

Declaration

```
procedure SelectNext(Direction: Boolean);
```

The *SelectNext* method selects the next tab in a tab set control, and scrolls that tab set control if necessary to bring the selected tab into view.

The value of the *Direction* parameters determines if the tab to the left or right is selected. If *Direction* is *True*, the tab to the right is selected. If *Direction* is *False*, the tab to the left is selected. When the last tab in either direction is selected, calling *SelectNext* using the same direction wraps around to the beginning of the tab order. For example, if your application has three tabs, First, Second, and Third, and Third is the current tab, calling *SelectNext*(*True*) selects First. Likewise, if First is the current tab, *SelectNext*(*False*) selects Third.

When *SelectNext* is called, the *OnClick* event of the tab set occurs, followed by the *OnChange* event, just as if the user had clicked on a new tab.

Example

The following code selects the next tab to the left of the current tab (or selects the last tab if the first tab is currently selected).

```
  TabSet1.SelectNext(False);
```

See also

TabIndex property

SelectorInc variable **System**

Declaration

```
var SelectorInc: Word;
```

SelectorInc contains the value that must be added to or subtracted from the selector part of a pointer to increment or decrement the pointer by 64K bytes.

SelLength property

Applies to

TComboBox, *TDBComboBox*, *TDBEdit*, *TDBMemo*, *TDriveComboBox*, *TEdit*, *TFilterComboBox*, *TMaskEdit*, *TMemo* components

Declaration

```
property SelLength: Integer;
```

The *SelLength* property returns the length (in characters) of the control's selected text. By using *SelLength* along with the *SelStart* property, you specify which part of the text in the control is selected. You can change the number of selected characters by changing the value of *SelLength*. When the *SelStart* value changes, the *SelLength* value changes accordingly.

The edit box or memo must be the active control when you change the value of *SelLength*, or nothing appears to happen.

Example

This example uses an edit box and a label on a form. When the user selects text in the edit box, the number of selected characters is reported in the caption of the label:

```
procedure TForm1.Edit1MouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Label1.Caption := 'Selected length = ' +
    IntToStr(Edit1.SelLength);
end;
```

See also

SelStart property, *SelText* property, *Text* property

SelStart property

Applies to

TComboBox, *TDBComboBox*, *TDBEdit*, *TDBMemo*, *TDriveComboBox*, *TEdit*, *TFilterComboBox*, *TMaskEdit*, *TMemo* components

Declaration

```
property SelStart: Integer;
```

The *SelStart* property returns the starting position of the selected part of the control's text, with the first character in the text having a value of 0. You can use *SelStart* along with the *SelLength* property to select a portion of the text. Specify the character you want the selected text to start with by its position in the text as the value of *SelStart*.

When the *SelStart* value changes, the *SelLength* value changes accordingly.

The edit box or memo must be the active control when you change the value of *SelStart*, or nothing appears to happen.

Example

This example uses an edit box and a label on a form. When the user selects text in the edit box, the starting position of the selected text is reported in the caption of the label:

```
procedure TForm1.Edit1MouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Label1.Caption := 'Selected text starts at character ' +
    IntToStr(Edit1.SelStart + 1);
end;
```

Note that if all the text in the edit box is selected, the value of the *SelStart* property is 0, the index value of the first item in the string array. Therefore, this example uses the expression *SelStart + 1* to make the result reported more understandable to most people.

See also

SelectAll method, *SelLength* property, *SelText* property, *Text* property

SelText property

Applies to

TComboBox, *TDBComboBox*, *TDBEdit*, *TDriveComboBox*, *TEdit*, *TFilterComboBox*, *TMaskEdit* components

Declaration

```
property SelText: string;
```

The *SelText* property contains the selected part of the control's text. You can use it to determine what the selected text is, or you can change the contents of the selected text by specifying a new string. If no text is currently selected, the *SelText* string is inserted in the text at the cursor.

Example

This example uses an edit box and a label on a form. When the user selects text in the edit box, the selected text is reported in the caption of the label.

```
procedure TForm1.Edit1MouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Label1.Caption := 'Selected string =  ' + Edit1.SelText;
end;
```

S

See also

SelLength property, *SelStart* property, *Text* property

SendToBack method

Applies to

All controls; *TForm* component

Declaration

```
procedure SendToBack;
```

The *SendToBack* method puts a windowed component behind all other windowed components within its parent component or form, or it puts a non-windowed component behind all other non-windowed components within its parent component or form. If the component has the input focus when the *SendToBack* method executes, it loses the input focus.

SendToBack is useful for changing the order of overlapping controls or forms.

The order in which controls stack on top of each order (also called the Z order) depends on whether the controls are windowed or non-windowed. For example, if you put a label and an image on a form so that one is on top of the other, the first one you placed on the form is the one on the bottom. Because both the label and the image are non-windowed controls, they "stack" as you would expect them to. Suppose that the image is on the top. If you call the *SendToBack* method for the image, the label then appears on top of the image.

The stacking order of windowed controls is the same. For example, if you put a memo on a form, then put a check box on top of it, the memo remains on the bottom. If you call *SendToBack* for the check box, the memo appears on top.

The stacking order of windowed and non-windowed controls cannot be mingled. For example, if you put a memo, a windowed control, on a form, and then put a label, a non-windowed control, on top of it, the label disappears behind the memo. Windowed controls always "stack" on top of non-windowed controls. In this example, if you call the *SendToBack* method of the memo, it remains on top of the label.

Example

This example uses two forms. When the user clicks the button on *Form2*, it moves *Form2* behind the other form and is no longer the active form:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  SendToBack;
end;
```

In this example, the parent of the two forms is the application itself.

See also

BringToFront method

ServerConv property

Applies to

TDDEServerItem component

Declaration

```
property ServerConv: TDdeServerConv;
```

The *ServerConv* property specifies the DDE server conversation component to associate with the DDE server item component. The value of *ServerConv* is the name of the DDE server conversation component that defines the DDE conversation.

Example

The following code sets *DDEServerConv1* to be the server conversation for *DDEServerItem1*.

```
   DDEServerItem1.ServerConv := DDEServerConv1;
```

See also

Name property

ServiceApplication property

Applies to

TDDEClientConv component

Declaration

```
property ServiceApplication: string;
```

The *ServiceApplication* property specifies the main executable file name (and path, if necessary) of a DDE server application, without the .EXE extension. Typically, this is the same value as the *DDEService* property. Sometimes, however, *DDEService* is a value other than the DDE server application's executable file name. In either case, *ServiceApplication* must be specified for Delphi to run an inactive DDE server to establish a DDE conversation.

Example

The following code sets the service application to 'ReportSmith'.

```
   DDEClientConv1.ServiceApplication := 'ReportSmith';
```

Session variable **DB**

Declaration

```
Session: TSession;
```

The *Session* variable is responsible for maintaining all of the database components used by your application. It is created automatically as part of your application's initialization and destroyed as part of your application's termination. The *Session* variable must remain active at all times; it can not be destroyed and recreated.

SetAsHandle method

Applies to

TClipboard object

Declaration

```
function SetAsHandle (Format: Word): THandle;
```

The *SetAsHandle* method places the data in the given format as a Windows handle. Once your application gives the handle to the Clipboard, it should not delete the handle. Instead, the Clipboard will delete the handle.

See the Windows API Help file for information about the available formats for *Format* parameter.

Example

The following code gives the handle of bitmap graphic data to the Clipboard.

```
Clipboard.SetAsHandle(CF_BITMAP);
```

See also

FormatCount property, *Formats* property, *GetAsHandle* method, *HasFormat* method

SetBounds method

Applies to

All controls

Declaration

```
procedure Setbounds(ALeft, ATop, AWidth, AHeight: Integer);
```

The *SetBounds* method sets the component's boundary properties, *Left*, *Top*, *Width*, and *Height*, to the values passed in *ALeft*, *ATop*, *AWidth*, and *AHeight*, respectively.

SetBounds enables you to set more than one of the component's boundary properties at a time. Although you can always set the individual boundaries, using *SetBounds* enables you to make several changes at once without repainting the control for each change.

Example

The following code doubles the size of a button control when the user clicks it:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.SetBounds(Left, Top, Height * 2, Width * 2);
end;
```

Note that you could use the following code instead, but each click would result in the button being redrawn twice: once to change the height, and once to change the width:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Height := Button1.Height * 2;
  Button1.Width := Button1.Width * 2;
end;
```

See also

Height property, *Left* property, *Top* property, *Width* property

SetComponent method

Applies to

TClipboard object

Declaration

```
procedure SetComponent(Component: TComponent);
```

The *SetComponent* method copies a component to the Clipboard. Specify the component you want copied as the value of the *Component* parameter.

Example

This example uses a button and a group box on a form. When the user clicks the button, the button is copied to the Clipboard and then retrieved from the Clipboard and placed in the new parent of the button, the group box. The name of the original button is changed to an empty string to avoid having two components with the same name at the same time.

```
implementation

uses Clipbrd;

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  Clipboard.SetComponent(Button1);                { copies button to the Clipboard }
  Button1.Name := '';          { prevents having two components with the same name }
  Clipboard.GetComponent(Self, GroupBox1);   { retrieves button from Clipboard and }
end;                                                    { places it in the group box }

initialization
  RegisterClasses([TButton]);                             { registers the TButton class }
end.
```

See also

AsText property, *GetComponent* method, *Owner* property, *Parent* property

SetData method

Applies to

TParam object; *TBCDField*, *TBlobField*, *TBooleanField*, *TBytesField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

For TParam objects

Declaration

```
procedure SetData(Buffer: Pointer);
```

The *SetData* method copies a new value for the parameter in native format from *Buffer*.

Example

```
  var I: Longint;
  I := 1221;
  { Set the data }
  Query1.ParamByName('CustNo').SetData(@I);
```

For field components

Declaration

```
procedure SetData(Buffer: Pointer);
```

SetData is the method used to assign "raw" data to the field. Unlike the *AsString* or *Text* properties, *SetData* performs no translation or interpretation of the data. *Buffer* must have sufficient space allocated for the data. Use the *DataSize* property to determine the space required. To set the data to NULL, pass **nil** for the *Buffer* parameter.

Example

```
  { Assign "raw" data to Field1 }
  with Field1 do
    begin
  { Allocate space }
    GetMem(Buffer, DataSize);
  { Fill Buffer with the desired data }
    ...
  { Do the assignment }
    Field1.SetData(Buffer)
  { Free the space }
    FreeMem(Buffer, DataSize);
    end;
```

See also
FieldDefs property, *TFieldDef* object

SetFields method

Applies to
TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure SetFields(const Values: array of const);
```

SetFields assigns the values specified in the *Values* array parameter to the fields in the dataset. If *Values* has fewer elements than there are fields, the remaining elements are unchanged. To assign a null value to a field, use the keyword **null**. To not assign any value to a field, use **nil**; the field will then get its default value.

Before calling this method, an application must first call *Edit* to put the dataset in Edit state. To then modify the current record in the database, it must then call *Post*.

Because this method depends explicitly on the structure of the underlying table, an application should use it only if the table structure will not change.

Example

```
  Table1.SetFields([208, 23.1]);
```

See also
Fields property

SetFocus method

Applies to
All controls; *TForm* component

Declaration

```
procedure SetFocus;
```

The *SetFocus* method gives the input focus to the control. If the control is a form, the form calls the *SetFocus* method of its active control.

Example
When the user clicks the button on this form, the list box becomes the active control and receives the input focus:

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

```
    ListBox1.SetFocus;
  end;
```

See also

ActiveControl property, *OnEnter* event

SetFormOLEDropFormats procedure **ToCtrl**

Declaration

```
procedure SetFormOleDropFormats(Form: TForm; const Fmts: array of BOleFormat);
```

SetFormOLEDropFormats specifies which object formats can be dropped on a form that was registered for drag-and-drop by the *RegisterFormAsOLEDropTarget* procedure. Use *SetFormOLEDropFormats* to modify which objects can be dropped. The formats in the *Fmts* array are registered so objects can be dropped on the form. Drop formats can be deleted with the *ClearFormOLEDropFormats* procedure.

Example

The following code resets the OLE object drop format for *Form1*, assuming *Fmts* is an array of *BOLEFormat* records.

```
  SetFormOLEDropFormats(Form1, Fmts);
```

See also

TOLEDropNotify object

SetKey method

Applies to

TTable component

Declaration

```
procedure SetKey;
```

The *SetKey* method puts the *TTable* in *SetKey* state (the *State* property is set to *dsSetKey*). This enables an application to search for values in database tables. In *SetKey* state, you can set the values of the search key buffer. The search key buffer is a set of fields corresponding to the table's key fields. After setting the values of the search key buffer fields, call *GotoKey*, *GotoNearest*, *FindKey*, or *FindNearest* to move the cursor to the matching record.

SetKey differs from *EditKey* in that the former clears all the elements of the search key buffer. *EditKey* leaves the elements of the search key buffer with their current values, but enables you to edit them.

Example

```
with Table1 do
  begin
  SetKey;
  FieldByName('State').AsString := 'CA';
  FieldByName('City').AsString := 'Scotts Valley';
  GotoKey;
  end;
```

See also

IndexFields property, *State* property, *TDataSetState* type

SetLink method

Applies to

TDDEClientConv component

Declaration

```
function SetLink(Service: string; Topic: string): Boolean;
```

The *SetLink* method specifies the service and topic of a DDE conversation and attempts to open the link if *ConnectMode* is *ddeAutomatic*. The *Service* parameter defines the DDE service and is assigned to the *DDEService* property. The *Topic* parameter defines the DDE topic and is assigned to the *DDETopic* property.

If *ConnectMode* is *ddeManual*, you must call *OpenLink* to initiate the conversation after calling *SetLink*.

Example

The following code establishes a link with a DDE server. The service is specified in the *DDEService* property of *DDEClientConv1*, and the topic is specified in the *DDETopic* property of *DDEClientConv1*. If the link is established, a message is displayed.

```
with DDEClientConv1 do
  if SetLink(DDEService, DDETopic) then
    MessageDlg('Link established.', mtInformation, [mbOK], 0);
```

See also

CloseLink method, *OpenLink* method

SetParams method

Applies to

TScrollBar component

Declaration

```
procedure SetParams(APosition, AMin, AMax: Integer);
```

The *SetParams* method sets the *Position*, *Min*, and *Max* property values of a scroll bar all at once.

Example

This example uses a scroll bar and a button on a form. When the user clicks the button, the minimum and maximum values of scroll bar are set, and the thumb tab moves to one-fifth of the distance to the right:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ScrollBar1.SetParams(100, 0, 500)
end;
```

SetPrinter method

Applies to

TPrinter object

Declaration

```
procedure SetPrinter(ADevvice, ADriver, APort: PChar; ADeviceMode: THandle);
```

The *SetPrinter* method specifies a printer as the current printer. You should seldom, if ever, need to call this method, but instead should access the printer you want in the *Printers* property array. For more information, see the Windows API *CreateDC* function.

SetRange method

Applies to

TTable component

Declaration

```
procedure SetRange(const StartValues, EndValues: array of const);
```

The *SetRange* method combines the functionality of the *SetRangeStart*, *SetRangeEnd*, and *ApplyRange* methods. *SetRange* assigns the elements of *StartValues* to the beginning index key, the elements of *EndValues* to the ending index key, and then calls *ApplyRange*. This enables an application to filter the data visible to the dataset.

If either *StartValues* or *EndValues* has fewer elements than the number of fields in the current index, then the remaining entries are set to NULL.

Note With Paradox or dBASE tables, these methods work only with indexed fields. With SQL databases, they can work with any columns specified in the *IndexFieldNames* property.

Example

```
Table1.SetRange([1000], [2000]);
```

See also

KeyExclusive property

SetRangeEnd method

Applies to

TTable component

Declaration

```
procedure SetRangeEnd;
```

SetRangeEnd indicates that subsequent assignments to field values will specify the end of the range of rows to include in the dataset. This enables an application to filter the data that is visible to it. Any column values not specified are not considered. The corresponding method *EditRangeEnd* indicates to keep existing range values and update with the succeeding assignments.

Call *ApplyRange* to apply the range filter defined with *SetRangeEnd* and *SetRangeStart*.

SetRangeEnd differs from *EditRangeEnd* in that it clears all the elements of the range filter to the default values (or NULL). *EditRangeEnd* leaves the elements of the range filter with their current values.

Note With Paradox or dBASE tables, these methods work only with indexed fields. With SQL databases, they can work with any columns specified in the *IndexFieldNames* property.

Example

```
with Table1 do
  begin
  SetRangeStart; { Set the beginning key }
  FieldByName('City').AsString := 'Felton';
  SetRangeEnd; { Set the ending key }
  FieldByName('City').AsString := 'Scotts Valley';
  ApplyRange; { Tell the dataset to establish the range }
  end;
```

See also

CancelRange method, *KeyExclusive* property, *EditRangeStart* method, *SetRange* method, *SetRangeStart* method

SetRangeStart method

Applies to
TTable component

Declaration

```
procedure SetRangeStart;
```

SetRangeStart indicates that subsequent assignments to field values will specify the start of the range of rows to include in the dataset. This enables an application to filter the data that is visible to it. Any column values not specified are not considered. The corresponding method *EditRangeStart* indicates to keep existing range values and update with the succeeding assignments.

Call *ApplyRange* to apply the range filter defined with *SetRangeEnd* and *SetRangeStart*.

SetRangeStart differs from *EditRangeStart* in that it clears all the elements of range filter to the default values (or NULL). *EditRangeStart* leaves the elements of range filter with their current values.

Note With Paradox or dBASE tables, these methods work only with indexed fields. With SQL databases, they can work with any columns specified in the *IndexFieldNames* property.

Example

```
with Table1 do
  begin
  SetRangeStart; { Set the beginning key }
  FieldByName('City').AsString := 'Ben Lomond';
  SetRangeEnd; { Set the ending key }
  FieldByName('City').AsString := 'Scotts Valley';
  ApplyRange; { Tell the dataset to establish the range }
  end;
```

See also
EditRangeEnd method, *KeyExclusive* property, *SetRangeEnd* method

SetSelTextBuf method

Applies to
TComboBox, *TDBComboBox*, *TDBEdit*, *TDBMemo*, *TEdit*, *TMaskEdit*, *TMemo* components

Declaration

```
procedure SetSelTextBuf(Buffer: PChar);
```

The *SetSelTextBuf* method sets the selected text in the edit box or memo control to the text in the null-terminated string pointed to by *Buffer*.

You should have no need to use the *SetSelTextBuf* method unless you are working with strings longer than 255 characters. Because an Object Pascal string has a limit of 255 characters, such properties as *Text* for an edit box, *Items* for a list box, and *Lines* for a memo control do not allow you to work with strings longer than 255 characters. *SetSelTextBuf* and the corresponding *GetSelTextBuf* methods use null-terminated strings that are up to 64K in length.

Example

This example uses a button and an edit box on a form. When the user selects text in the edit box and clicks the button, new text replaces the selected text. The string specified as the parameter of the *SetSelTextBuf* is a null-terminated string of type *PChar*.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Button1.Caption := 'Click me';
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.SetSelTextBuf('You clicked the button');
end;
```

See also

GetSelTextBuf method, *SetTextBuf* method

SetTabFocus method

Applies to

TTabbedNotebook component

Declaration

```
procedure SetTabFocus(Index: Integer);
```

The *SetTabFocus* changes the active page in the tabbed notebook control. The *Index* parameter is the *PageIndex* value of the page, which indicates the page's position in the *Pages* property array. For example, the first page in the control has an index value of 0, the second page has an index value of 1, and so on.

Example

The following code sets focus to the first page in *TabbedNotebook1*.

```
TabbedNoteBook1.SetTabFocus(0);
```

See also

ActivePage property, *GetIndexForPage* method

SetText method

Applies to
TStringList, *TStrings* objects

Declaration

```
procedure SetText(Text: PChar);
```

The *SetText* method writes an entire list of strings at one time. It is meant to be used with components that contain multiple strings where you would find it convenient to treat all the strings as one block. For example, *SetText* would be useful with a memo component, which can hold multiple strings.

Specify the text you want write as the value of the *Text* parameter, making sure the block of text to which you are referring is a null-terminated string.

Example
The following code uses *SetText* to write the contents of an edit box to *Memo1*.

```
var
  TheText: array[0..255] of Char;
begin
  StrPCopy(TheText, Edit1,Text);
  Memo1.SetText(TheText);
end;
```

See also
GetText method

SetTextBuf method

Applies to
All controls; *TClipboard* object

Declaration

```
procedure SetTextBuf(Buffer: PChar);
```

The *SetTextBuf* method sets the control's text to the text in the buffer pointed to by *Buffer*. *Buffer* must point to a null-terminated string.

Usually, you use *SetTextBuf* and the corresponding *GetTextBuf* only when you need to work with strings that are longer than 255 characters. Because an Object Pascal style string has a limit of 255 characters, such properties as *Text* for an edit box, *Items* for a list box, and *Lines* for a memo control do not allow you to work with strings longer than 255 characters. *GetTextBuf* and *SetTextBuf* use null-terminated strings that are up to 64K in length.

Example

This example uses a button and an edit box on a form. When the user clicks the button, text appears in the edit box. The string specified as the parameter of the *SetTextBuf* is a null-terminated string, as it is of type *PChar*.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Button1.Caption := 'Click me';
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.SetTextBuf('You clicked the button');
end;
```

See also

GetTextBuf method, *SetSelTextBuf* method

SetTextBuf procedure **System**

Declaration

```
procedure SetTextBuf(var F: Text; var Buf [ ; Size:   Word ] );
```

The *SetTextBuf* procedure assigns an I/O buffer to a text file.

F is a text file variable, *Buf* is any variable, and *Size* is a optional expression.

Each *Text* file variable has an internal 128-byte buffer that buffers *Read* and *Write* operations. This buffer is adequate for most operations. However, heavily I/O-bound programs benefit from a larger buffer to reduce disk head movement and file system overhead.

SetTextBuf changes the text file *F* to use the buffer specified by *Buf* instead of *F's* internal buffer. *Size* specifies the size of the buffer in bytes. If *Size* is omitted, *SizeOf*(*Buf*) is assumed. The new buffer remains in effect until *F* is next passed to *AssignFile*.

SetTextBuf can be called immediately after *Reset*, *Rewrite*, and *Append*, but never apply it to an open file.

If you call *SetTextBuf* on an open file once I/O operations have taken place, you could lose data because of the change of buffer.

Delphi does not ensure that the buffer exists for the entire duration of I/O operations on the file. A common error is to install a local variable as a buffer, then use the file outside the procedure that declared the buffer.

Example

```
uses Dialogs;

var
  F, FTwo: System.TextFile;
  Ch: Char;
  Buf: array[1..4095] of Char;  { 4K buffer }
begin
  if OpenDialog1.Execute then begin
    AssignFile(F, ParamStr(1));
    { Bigger buffer for faster reads }
    SetTextBuf(F, Buf);
    Reset(F);
    { Dump text file into another file }
    AssignFile(FTwo, 'WOOF.DOG');
    Rewrite(FTwo);
    while not Eof(f) do
    begin
      Read(F, Ch);
      Write(FTwoCh);
    end;
    System.CloseFile(F);
    System.CloseFile(FTwo);
  end;
end;
```

See also

Append procedure, *AssignFile* procedure, *Read* procedure, *Reset* procedure, *Rewrite* procedure, *SizeOf* function, *Write* procedure

SetUpdateState method

Applies to

TOutline component

Declaration

```
procedure SetUpdateState(Value: Boolean);
```

The *SetUpdateState* method sets the update state of the outline component. If you add or delete an item from the outline, by default the outline component reindexes the subsequent items that have indexes changed by the addition or deletion. For a large outline, this can slow processing and consume a large amount of processing time. If you have a large outline, or plan to add many items, you can turn off automatic reindexing to speed up processing. You can quickly add items, but the indexes of all subsequent items will no longer be valid. When you finish adding items, you should turn on reindexing so the index values of the subsequent items are made valid again.

By passing *True* in the *Value* property of *SetUpdateState*, you turn off automatic reindexing. This is functionally the same as calling the *BeginUpdate* method. By passing

False in the *Value* property of *SetUpdateState*, you turn on automatic indexing. This is functionally the same as calling the *EndUpdate* method.

Example

The following code turns off automatic reindexing on *Outline1*.

```
Outline1.SetUpdateState(True);
```

SetVariable method

Applies to

TReport component

Declaration

```
function SetVariable(Name, Value: string): Boolean;
```

The *SetVariable* method changes the value of a report variable. The *Name* parameter specifies which report variable changes, and the *Value* parameter specifies the new value. Once the *SetVariable* method has been called, your application can call the *RecalcReport* method, which recalculates the report using the new value for the specified report variable.

The *SetVariable* method sends a DDE message to ReportSmith Runtime to change the specified report variable with the new value, and looks for a DDE message from ReportSmith Runtime in return. If *SetVariable* returns *True*, the message was sent to ReportSmith Runtime successfully. If it returns *False*, ReportSmith Runtime could not receive the message at the current time.

To learn more about report variables, see your ReportSmith documentation.

Note Before calling *SetVariable*, you must load a report by specifying the *ReportName* property.

Example

The following code attempts to set the 'LastName' report variable to 'Schaeferle'. If successful, it then recalculates the report.

```
if Report1.SetVariable('LastName', 'Schaeferle') then
  Report1.RecalcReport;
```

See also

RecalcReport method, *SetVariableLines* method

SetVariableLines method

Applies to

TReport component

Declaration

```
function SetVariableLines(Name, Value: TStrings): Boolean;
```

The *SetVariableLines* method changes the value of a report variable. The *Name* parameter specifies which report variable changes, and the *Value* parameter specifies the new value, which is a list of strings. Once the *SetVariableLines* method has been called, your application can call the *RecalcReport* method, which recalculates the report using the new value for the specified report variable.

The *SetVariableLines* method sends a DDE message to ReportSmith Runtime to change the specified report variable with the new value, and looks for a DDE message from ReportSmith Runtime in return. If *SetVariableLines* returns *True*, the message was sent to ReportSmith Runtime successfully. If it returns *False*, ReportSmith Runtime could not receive the message at the current time.

To learn more about report variables, see your ReportSmith documentation.

Note Before calling *SetVariable*, you must load a report by specifying the *ReportName* property.

See also
RecalcReport method, *SetVariable* method

Shape property

Applies to
TBevel, *TShape* components

The *Shape* property determines the visual shape of the component.

For bevels

Declaration

```
property Shape: TBevelShape;
```

The *Shape* property determines the shape the bevel control assumes. These are the possible values:

Value	Meaning
bsBox	The bevel assumes a box shape.
bsFrame	The bevel assumes a frame shape.
bsTopLine	The bevel becomes a line at the top of the bevel control.
bsBottomLine	The bevel becomes a line at the bottom of the bevel control.
bsLeftLine	The bevel becomes a line at the left side of the bevel control.
bsRightLine	The bevel becomes a line at the right side of the bevel control.

Example

This code uses a bevel control and a button. When the user clicks the button, the bevel becomes a raised frame:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Bevel1.Shape := bsFrame;
  Bevel1.Style := bsRaised;
end;
```

For shape controls

Declaration

```
property Shape: TShapeType;
```

The *Shape* property determines how a *TShape* component appears on a form. These are the possible values and their meanings:

Value	Meaning
stEllipse	The shape is an ellipse.
stRectangle	The shape is a rectangle.
stRoundRect	The shape is a rectangle with rounded corners.
stRoundSquare	The shape is a square with rounded corners.
stSquare	The shape is a square.
stCircle	The shape is a circle.

Example

This example uses a shape component on a form. When the user clicks the shape, it becomes a ball with red stripes:

```
procedure TForm1.Shape1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
with Shape1 do
  begin
    Shape := stCircle;
    Brush.Color := clRed;
    Brush.Style := bsVertical;
  end;
end;
```

Shareable property

Applies to

TMediaPlayer component

Declaration

```
property Shareable: Boolean;
```

The *Shareable* property determines whether more than one application can share a multimedia device. If *Shareable* is *False*, no other components or applications can access the device. If *True*, more than one component or application can access the device. *Shareable* defaults to *False*.

You should set *Shareable* before opening a device. Some devices aren't shareable. If you set *Shareable* to *True* and try to open a device that isn't shareable by more than one application, the *Open* method fails and the error code is returned to the *Error* property.

Example

The following code sets the *Shareable* property of a media player named *MediaPlayer1* to *True* before attempting to open the Microsoft Video for Windows device. Attach this code to the *OnClick* event handler of a bitmap button named *BitBtn1*. If an exception occurs when the *Open* method is called, a message dialog box displays the error. Note that this example assumes that C:\KA-BAR.AVI is a valid video file name.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  with MediaPlayer1 do
  begin
    try
      FileName := 'c:\Ka-Bar.AVI';
      Shareable := True;
     Open;
    except
      MessageDlg(MediaPlayer1.ErrorMessage, mtError, [mbOk], 0);
    end;
  end;
end;
```

ShortCut function

Menus

Declaration

```
function ShortCut(Key: Word; Shift: TShiftState): TShortCut;
```

The *ShortCut* function creates a menu shortcut at run time. Specify a *Key* value by using a virtual key code.You can find a table of virtual key codes in the Help system; search for the topic Virtual Key Codes. Specify a Shift value by using a set of type *TShiftState*. For example, to specify the Shift key, use the set [ssShift]. To specify a Shift and Ctrl key combination, use the set [ssShift, ssCtrl].

Once you create a shortcut, you can assign it to the *ShortCut* property of a menu item.

The *TextToShortCut* function can also create a shortcut. This function converts a string to a shortcut; therefore, it's useful when you want to let the user specify the shortcut. *TextToShortCut* executes much more slowly, however, so you should use the *ShortCut* function whenever possible.

Example

This code creates a shortcut, *Ctrl+O*, at run time and assigns it to the Open command on a File menu.

```
begin
  OpenCommand.ShortCut := ShortCut(Word('O'), [ssCtrl]);
end;
```

See also

ShortCut property, *ShortCutToKey* procedure, *ShortCutToText* function, *TextToShortCut* function

ShortCut property

Applies to

TMenuItem component

Declaration

```
property ShortCut: TShortCut;
```

The *ShortCut* property determines the key strokes users can use to access a menu item. The key combination the user can use appears to the right of the menu item in the menu. To see an example of menu shortcuts, pull down the Delphi Edit menu and note the menu shortcuts on the right side of some of the editing commands.

Usually you set menu shortcuts for menu items in the Object Inspector, which gives you a long list to choose from. If you create menu items at run time, however, you can create shortcuts for them too. Choose from these functions and procedures for more information about working with shortcuts at run time:

Routine	Purpose
ShortCut function	Creates a shortcut for a menu item programmatically.
ShortCutToKey procedure	Obtains the virtual key code and shift state of an existing shortcut.
ShortCutToText function	Returns the text string of an existing shortcut. Use this function to display a shortcut you created at run time on a menu item.
TextToShortCut function	Converts a text string to a shortcut. Use this function to allow users to specify the shortcut characters.

Example

This code creates a shortcut, *Ctrl+C*, at run time and assigns it to the Close command on a File menu.

```
begin
  CloseCommand.ShortCut := ShortCut(Word('C'), [ssCtrl]);
end;
```

ShortCutToKey procedure **Menus**

Declaration

```
procedure ShortCutToKey(ShortCut: TShortCut; var Key: Word; var Shift: TShiftState);
```

The *ShortCutToKey* procedure breaks a menu shortcut apart into its virtual key code and shift state parts.

Example

The following code redefines the *ShortCut* of CloseCommand if the original short cut used the [ssCtrl] shift state.

```
var
  TheKey: Word;
  TheShiftState: TShiftState;
begin
  ShortCutToKey(CloseCommand.ShortCut, TheKey, TheShiftState);
  if TheShiftState = [ssCtrl] then
    CloseCommand.ShortCut := ShortCut(Word('C'), [ssShift]);
end;
```

See also

ShortCut function, *ShortCut* property, *ShortCutToText* function, *TextToShortCut* function

ShortCutToText function **Menus**

Declaration

```
function ShortCutToText(ShortCut: TShortCut): string;
```

The *ShortCutToText* function converts a shortcut into a string. Your application can use this function any time it needs to display a menu shortcut as a string.

Note The *ShortCut* property of a menu item is of type *TShortCut*, so you can assign a shortcut you create at run time using either the *ShortCut function* or the *TextToShortCut function* directly to the menu item's *ShortCut* property. You don't need to use *ShortCutToText* to convert the shortcut to a text string first.

Example

This code converts the menu shortcut assigned to the OpenCommand *ShortCut* property to a string and displays it in an edit box:

```
Edit1.Text := ShortCutToText(OpenCommand.ShortCut);
```

See also
ShortCut function, *ShortCut* property, *ShortCutToKey* procedure, *TextToShortCut* function

Show method

Applies to
All controls; *TForm* component

Declaration
```
procedure Show;
```

The *Show* method makes a form or control visible by setting its *Visible* property to *True*. If the *Show* method of a form is called and the form is somehow obscured, *Show* tries to make the form visible by bringing it to the front with the *BringToFront* method.

Example
This code puts away the current form and displays another:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Hide;
  Unit2.Form2.Show;
end;
```

See also
Hide method, *ShowModal* method, *Visible* property

ShowAccelChar property

Applies to
TLabel component

Declaration
```
property ShowAccelChar: Boolean;
```

The *ShowAccelChar* property determines how an ampersand in the caption of a label appears. If *ShowAccelChar* is *True*, an ampersand appears as an underline under the character to its right in the caption indicating the underlined character is an accelerator character. If *ShowAccelChar* is *False*, the ampersand character appears as an ampersand.

Example
This example uses two labels on a form. The first label has a caption with an accelerator character in it. The second label also includes an ampersand, but it does not appear as an accelerator character.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Label1.ShowAccelChar := True;
  Label1.Caption := 'An &Underlined character appears here';
  Label2.ShowAccelChar := False;
  Label2.Caption := 'An ampersand (&) appears here';
end;
```

See also

Caption property, *FocusControl* property

ShowException method

Applies to

TApplication component

Declaration

```
procedure ShowException(E: Exception);
```

The *ShowException* method displays an exception that occurred in your application in a message box.

ShowException procedure SysUtils

Declaration

```
procedure ShowException(ExceptObject: TObject; ExceptAddr: Pointer);
```

This procedure displays the message associated with an exception, together with the exception's physical address. The exception's address is converted to a logical segment address that can be used with Search|Find Error menu command to find the statement that raised the exception.

ShowGlyphs property

Applies to

TFileListBox component

Declaration

```
property ShowGlyphs: Boolean;
```

The value of the *ShowGlyphs* property determines whether glyphs (bitmaps) appear next to the file names listed in the file list box. If *ShowGlyphs* is *True*, the glyphs appear; if *ShowGlyphs* is *False*, the glyphs don't appear. The default value is *False*.

Example

If the files in the list box don’t have the glyphs next to them, this line of code redisplays the files with the glyphs included:

```
FileListBox1.ShowGlyphs := True;
```

ShowHint property

Applies to

All controls, *TApplication* component

The *ShowHint* property is used at both the control and the application level.

For all controls

Declaration

```
property ShowHint: Boolean;
```

Description

The *ShowHint* property determines if the control should display a Help Hint when the user’s mouse pointer rests momentarily on the control. The Help Hint is the value of the Hint property, which is displayed in a box just beneath the control. If *ShowHint* property is *True*, the Help Hint will appear.

If *ShowHint* is *False*, the Help Hint may or may not appear. If *ParentShowHint* is *False* also, the Help Hint won’t appear. If, however, *ParentShowHint* is *True*, whether or not the Help Hint appears depends on the setting of the *ShowHint* property of the control’s parent. For example, imagine a check box within a group box. If the *ShowHint* property of the group box is *True* and the *ParentShowHint* property of the check box is *True*, but the *ShowHint* property of the check box is *False*, the check box will still display its Help Hint.

The default value is *False*.

Changing the *ShowHint* value to *True* automatically sets the *ParentShowHint* property to *False*.

Example

This example uses an edit box on a form. When the application runs and the user places the mouse pointer over the edit box, a Help Hint in an aqua box appears:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.ShowHint := True;
  Application.HintColor := clAqua;
  Application.HintPause := 1000;
  Edit1.ShowHint := True;
```

```
    Edit1.Hint := 'Enter your name';
  end;
```

See also

HintColor property, *HintPause* property, *ParentShowHint* property, *ShowHint* for the application

For applications

Applies to

TApplication component

Declaration

```
property ShowHint: Boolean;
```

Run-time only. The *ShowHint* property determines whether Help Hints are enabled or disabled for the entire application. If *ShowHint* is *True*, Help Hints are enabled; if *ShowHint* is *False*, Help Hints are disabled. The default value is *True*.

Even if *ShowHint* is *True*, a Help Hint won't appear for a particular control unless its own *ShowHint* property is *True*, or its *ParentShowHint* property is *True* and its parent's *ShowHint* property is *True*.

Setting *ShowHint* for the application to *False* disables all Help Hints, regardless of the value of the *ShowHint* properties for individual controls.

Example

This example includes an control that has a *Hint* property value and has its ShowHint property value set to *True*. When the application runs and the user places the mouse cursor over the control, a Help Hint appears for the control in a red hint box after a delay of 1000 milliseconds:

```
  procedure TForm1.FormCreate(Sender: TObject);
  begin
    Application.ShowHint := True;
    Application.HintColor := clAqua;
    Application.HintPause := 1000;
  end;
```

See also

Hint property, *HintColor* property, *HintPause* property

Showing property

Applies to

All controls

Declaration

```
property Showing: Boolean;
```

Run-time and read only. The *Showing* property specifies whether a component is currently showing on the screen. If the *Visible* properties of a component and all the parents in its parent hierarchy are *True*, *Showing* is *True*. If one of the ancestors of the component has a *Visible* property value of *False*, *Showing* is *False*.

Example

The following code adds the name of all controls in a form for which *Showing* is *False* to a list box. When *Button2* is clicked, *ListBox1* is filled with the names of all windowed controls that aren't showing in *Form1*.

```
procedure TForm1.Button2Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ComponentCount -1 do
    if Components[I] is TWinControl then
     if not TWinControl(Components[I]).Showing then
         ListBox1.Items.Add(Components[I].Name);
end;
```

See also

Hide method, *Show* method

ShowMessage procedure

Dialogs

Declaration

```
procedure ShowMessage(const Msg: string);
```

The *ShowMessage* procedure displays a message box with an OK button. The *Msg* parameter is the message string that appears within the message box. The name of your application's executable file appears as the caption of the message box.

Example

This example uses a button on a form. When the user clicks the button, a message dialog box appears with instructions to push the OK button.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Push this button');
end;
```

See also

MessageBox method, *MessageDlg* function, *MessageDlgPos* function, *ShowMessagePos* procedure

ShowMessagePos procedure

Dialogs

Declaration

```
procedure ShowMessagePos(const Msg: string; X, Y: Integer);
```

The *ShowMessagePos* procedure displays a message box with an OK button at a specified screen location. The *Msg* parameter is the message string that appears within the message box. The *X* and *Y* parameters are the screen coordinates for the upper left corner of the message box. The name of your application's executable file appears as the caption of the message box.

Example

This example uses a button on a form. When the user clicks the button, a message dialog box appears with a comment about Delphi programmers. The upper left corner of the message box appears at screen location 100, 100.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessagePos('Delphi programmers are more productive', 100, 100);
end;
```

See also

MessageBox method, *MessageDlg* function, *MessageDlgPos* function, *ShowMessage* procedure

ShowModal method

Applies to

TForm component

Declaration

```
function ShowModal: Integer;
```

The *ShowModal* method makes a form the active form, just like *Show*, but also makes the form modal; therefore the user must put the form away before the application can continue to run.

When the user chooses to close the form in some manner, the value of the form's *ModalResult* property changes to a nonzero value. When *ModalResult* has a nonzero value, the form terminates, and the *ModalResult* value is passed as the result of the *ShowModal* method.

Example

The following code displays a message box after *BtnBottomDlg* is shown modally and is closed by an OK button.

```
BtnBottomDlg.ShowModal;
if BtnBottomDlg.ModalResult=mrOK then
```

```
    MessageDlg('OK!', mtInformation, [mbOK], 0);
```

This code uses two forms and a button on the first form. The user must close *Form2* before the focus returns to *Form1*.

```
  uses Unit2;

  procedure TForm1.Button1Click(Sender: TObject);
  begin
    Form2.ShowModal;
  end;
```

See also

ModalResult property, *Show* method

Sin function — System

Declaration

```
function Sin(X: Real): Real;
```

The *Sin* function returns the sine of the argument.

X is a real-type expression. *Sin* returns the sine of the angle *X* in radians.

Example

```
  var
    R: Real;
    S: string;
  begin
    R := Sin(Pi);
    Str(R:5:3, S);
    Canvas.TextOut(10, 10, 'The Sin of Pi is ' + S);
  end;
```

See also

ArcTan function, *Cos* function, *TypeOf* function

S

Size property

Applies to

TFieldDef, *TFont* object; *TBCDField*, *TBlobField*, *TBytesField*, *TGraphicField*, *TIntegerField*, *TMemoField*, *TStringField*, *TTimeField*, *TVarBytesField* components

For fonts

Declaration

```
property Size: Integer;
```

The *Size* property value is the size of the font, which is the height of the font minus the internal leading that appears at the top of the font. Whenever you specify a font size in points, use the *Size* property. If you are concerned with the height of the font on the screen—the number of pixels the font needs—use the *Height* property instead of *Size*. Users usually specify font sizes in points within an application, while programmers are usually concerned with the actual height of the font—which includes the internal leading—when displaying a font on the screen.

Delphi determines the value of the *Size* property using this formula:

```
Font.Size = -Font.Height * 72 / Font.PixelsPerInch
```

Therefore, whenever you enter a positive value for the *Size* property, the font's *Height* property value changes to a negative number. Conversely, if you enter a positive value for the *Height* property, the font's *Size* property changes to a negative number.

Example

This examples uses a button on a form. When the user clicks the button, the size of the font used by the button changes to 24 points.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Font.Size := 24;
end;
```

See also

Font property, *Height* property, *PixelsPerInch* property

For TFieldDef objects

Declaration

```
property Size: Integer;
```

Run-time and read only. Reports the size of the *TFieldDef* object. *Size* is meaningful only for a *TFieldDef* object with one of the following *TFieldType* values: *ftString*, *ftBCD*, *ftBytes*, *ftVarBytes*, *ftBlob*, *ftMemo* or *ftGraphic*. For string and byte fields, *Size* is the number of bytes reserved in the table for the field. For a BCD field, it is the number of digits following the decimal point. For a BLOB, memo, or graphic field it is the number of bytes in the field.

Example

```
{ Allocate enough memory to make a copy of the BLOB }
GetMem(PBlob, BlobFieldDef.Size);
```

See also
TField component

For field components

Declaration

```
property Size: Integer;
```

For a *TStringField*, *Size* is the number of bytes reserved for the field in the dataset. For a *TBCDField*, it is the number of digits following the decimal point. For a *TBlobField*, *TBytesField*, *TVarBytesField*, *TMemoField,* or *TGraphicField,* it is the size of the field as stored in the table.

SizeOf function System

Declaration

```
function SizeOf(X): Word;
```

The *SizeOf* function returns the number of bytes occupied by *X*.

X is either a variable reference or a type identifier.

Always use *SizeOf* when passing values to *FillChar*, *Move,* and *GetMem*.

When applied to an instance of an object type that has a virtual method table (VMT), *SizeOf* returns the size stored in the VMT.

Example

```
type
  CustRec = record
    Name: string[30];
    Phone: string[14];
  end;
 var
  P: ^CustRec;
begin
  GetMem(P, SizeOf(CustRec));
  Canvas.TextOut(10, 10, 'The size of the record is ' + IntToStr(SizeOf(CustRec)));
  FreeMem (P, SizeOf(CustRec));
  Readln;
end;
```

See also
FillChar procedure, *GetMem* procedure, *Move* procedure

SmallChange property

Applies to
TScrollBar component

Declaration

```
property SmallChange: TScrollBarInc;
```

The *SmallChange* property determines how far the thumb tab moves when the user clicks the arrows at the end of the scroll bar to scroll or uses the arrow keys on the keyboard. The default value is 1.

For example, if *SmallChange* is 1000, each time the user clicks an arrow on the scroll bar, the thumb tab moves 1000 positions. The number of positions is determined by the difference between the *Max* property value and the *Min* property value. If the *Max* property is 30000 and the *Min* property is 0, the user would need to click an arrow on the scroll bar 30 times to move the thumb tab from one end of the scroll bar to the other.

Example
This code determines that when the user clicks an arrow on the scroll bar, the thumb tab moves 10 positions on the scroll bar:

```
ScrollBar1.SmallChange := 10;
```

See also
LargeChange property, *Max* property, *Position* property

Sort method

Applies to
TStringList object

Declaration

```
procedure Sort;
```

The *Sort* method sorts the strings in a string list object in alphabetical order.

Example
The following code sorts *MyStringList*.

```
MyStringList.Sort;
```

See also
Sorted property

Sorted property

Applies to

TStringList object; *TComboBox*, *TDBComboBox*, *TDBListBox*, *TListBox* components

For combo and list boxes

Declaration

```
property Sorted: Boolean;
```

The *Sorted* property indicates whether the items in a list box or combo box are arranged alphabetically. To sort the items, set the *Sorted* value to *True*. If *Sorted* is *False*, the items are unsorted.

If you add or insert items when *Sorted* is *True*, Delphi automatically places them in alphabetical order.

Example

This example uses an edit box, a list box, and two buttons on a form. The buttons are named *Add* and *Sort*. When the user clicks the *Add* button, the text in the edit box is added to the list in the list box. When the user clicks the *Sort* button, the list in the list box is sorted and remains sorted, even if additional strings are added:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ListBox1.Items.Add('Not');
  Listbox1.Items.Add('In');
  ListBox1.Items.Add('Alphabetical');
  ListBox1.Items.Add('Order');
end;

procedure TForm1.AddClick(Sender: TObject);
begin
  ListBox1.Items.Add(Edit1.Text);
end;

procedure TForm1.SortClick(Sender: TObject);
begin
  ListBox1.Sorted := True;
end;
```

See also

Add method, *Insert* method, *Items* property

For string list objects

Declaration

```
property Sorted: Boolean;
```

The value of the *Sorted* property determines the order of the strings in the list of strings maintained by the string list. If *Sorted* is *True*, the strings are sorted in ascending order. If *Sorted* is *False*, the strings are unsorted.

Example

This example uses a list box on a form. When the application runs, a string list object is created and three strings are added to it. The strings are sorted and added to the list box, where they appear in their sorted order:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  MyList: TStringList;
begin
  MyList := TStringList.Create;
  MyList.Add('Plants');
  MyList.Add('Animals');
  MyList.Add('Minerals');
  MyList.Sorted := True;
  ListBox1.Items.AddStrings(MyList);
  MyList.Free;
end;
```

See also

Add method, *IndexOf* method, *Sort* method, *Strings* property

Source property

Applies to

TBatchMove component

Declaration

```
property Source: TDataSet;
```

Source specifies a dataset (a *TQuery* or *TTable* component) corresponding to an existing source table.

Example

```
BatchMove1.Source := Table1;
```

See also
Destination property

Spacing property

Applies to
TBitBtn, *TSpeedButton* components

Declaration

```
property Spacing: Integer;
```

The *Spacing* property determines where the image and text appear on a bitmap or speed button. *Spacing* determines the number of pixels between the image (specified in the *Glyph* property) and the text (specified in the *Caption* property). The default value is 4.

If *Spacing* is a positive number, its value is the number of pixels between the image and text. If *Spacing* is 0, no pixels will be between the image and text. If *Spacing* is –1, the text appears centered between the image and the button edge. The number of pixels between the image and text is equal to the number of pixels between the text and the button edge opposite the glyph.

Example
This example loads a bitmap from a file when the form is created and places the bitmap 20 pixels from the right side of the button text:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  BitBtn1.Glyph.LoadFromFile('c:\delphi\bin\mybitmap.bmp');
  BitBtn1.Layout := blGlyphLeft;
  BitBtn1.Spacing := 20;
end;
```

See also
Caption property, *Layout* property, *Margin* property

SPtr function System

Declaration

```
function SPtr: Word;
```

The *SPtr* function returns the offset of the stack pointer within the stack segment.

Example

```
function MakeHexWord(w: Word): string;
  const
    hexChars: array [0..$F] of Char =
      '0123456789ABCDEF';
var
    HexStr : string;
  begin
    HexStr := '';
    HexStr := HexStr + hexChars[Hi(w) shr 4];
    HexStr := HexStr + hexChars[Hi(w) and $F];
    HexStr := HexStr + hexChars[Lo(w) shr 4];
    HexStr := HexStr + hexChars[Lo(w) and $F];
    MakeHexWord := HexStr;
  end;

var
  i: Integer;
  Y: Integer;
  S: string;
begin
  Y := 10;
  S := 'The current code segment is $' + MakeHexWord(CSeg);
  Canvas.TextOut(5, Y, S);

Y := Y + Canvas.TextHeight(S) + 5;
  S := 'The global data segment is $' + MakeHexWord(DSeg);
  Canvas.TextOut(5, Y, S);

Y := Y + Canvas.TextHeight(S) + 5;
  S := 'The stack segment is $' + MakeHexWord(SSeg);
  Canvas.TextOut(5, Y, S);

Y := Y + Canvas.TextHeight(S) + 5;
  S := 'The stack pointer is at $' + MakeHexWord(SPtr);
  Canvas.TextOut(5, Y, S);

Y := Y + Canvas.TextHeight(S) + 5;
  S := 'i is at offset $' + MakeHexWord(Ofs(i));
  Canvas.TextOut(5, Y, S);

Y := Y + Canvas.TextHeight(S) + 5;
  S := 'in segment $' + MakeHexWord(Seg(i));
  Canvas.TextOut(5, Y, S);
end;
```

See also
SSeg function

SQL property

Applies to
TQuery component

Declaration

```
property SQL: TStrings;
```

The *SQL* property holds the text of the SQL statement that will be executed when *Open* or *ExecSQL* is called. Once a query has been executed by *Open*, you must call the *Close* method before you can change the *SQL* text.

You can create the text for the *SQL* property:

Delphi also supports *heterogeneous queries* against more than one server or table type (for example, data from an Oracle table and a Paradox table).

Note The *SQL* property may contain only one complete SQL statement at a time. In general, multiple statements are not allowed. Some servers support multiple statement "batch" syntax; if the server supports this, then such statements are allowed.

See also

Text property

Sqr function — System

Declaration

```
function Sqr(X: Real): (Real);
```

The *Sqr* function returns the square of the argument.

X is a real-type expression. The result, of the same type as *X*, is the square of *X*, or *X***X*.

Example

```
var
  S, Temp: string;
begin
   Str(Sqr(5.0):3:1, Temp);
   S := '5 squared is ' + Temp + #13#10;
   Str(Sqrt(2.0):5:4, Temp);
   S := S + 'The square root of 2 is ' + Temp;
   MessageDlg(S, mtInformation, [mbOk], 0);
end;
```

See also

Sqrt function

Sqrt function — System

Declaration

```
function Sqrt(X: Real): Real;
```

The *Sqrt* function returns the square root of the argument.

X is a real-type expression. The result is the square root of *X*.

Example

```
var
  S, Temp: string;
begin
   Str(Sqr(5.0):3:1, Temp);
   S := '5 squared is ' + Temp + #13#10;
   Str(Sqrt(2.0):5:4, Temp);
   S := S + 'The square root of 2 is ' + Temp;
   MessageDlg(S, mtInformation, [mbOk], 0);
end;
```

See also

Sqr function

SSeg function **System**

Declaration

```
function SSeg: Word;
```

The *SSeg* function returns the current value of the SS register.

The result, of type *Word,* is the segment address of the stack segment.

Example

```
function MakeHexWord(w: Word): string;
  const
    hexChars: array [0..$F] of Char =
      '0123456789ABCDEF';
  var
    HexStr : string;
  begin
    HexStr := '';
    HexStr := HexStr + hexChars[Hi(w) shr 4];
    HexStr := HexStr + hexChars[Hi(w) and $F];
    HexStr := HexStr + hexChars[Lo(w) shr 4];
    HexStr := HexStr + hexChars[Lo(w) and $F];
    MakeHexWord := HexStr;
  end;

var
  i: Integer;
  Y: Integer;
  S: string;
begin
  Y := 10;
  S := 'The current code segment is $' + MakeHexWord(CSeg);
  Canvas.TextOut(5, Y, S);

Y := Y + Canvas.TextHeight(S) + 5;
```

```
    S := 'The global data segment is $' + MakeHexWord(DSeg);
    Canvas.TextOut(5, Y, S);

  Y := Y + Canvas.TextHeight(S) + 5;
    S := 'The stack segment is $' + MakeHexWord(SSeg);
    Canvas.TextOut(5, Y, S);

  Y := Y + Canvas.TextHeight(S) + 5;
    S := 'The stack pointer is at $' + MakeHexWord(SPtr);
    Canvas.TextOut(5, Y, S);

  Y := Y + Canvas.TextHeight(S) + 5;
    S := 'i is at offset $' + MakeHexWord(Ofs(i));
    Canvas.TextOut(5, Y, S);

  Y := Y + Canvas.TextHeight(S) + 5;
    S := 'in segment $' + MakeHexWord(Seg(i));
    Canvas.TextOut(5, Y, S);
  end;
```

See also
CSeg function, *DSeg* function, *SPtr* function

Start property

Applies to
TMediaPlayer component

Declaration

```
property Start: Longint;
```

The *Start* property specifies the starting position within the currently loaded medium. *Start* is the beginning of the medium for devices that don't use tracks, or the beginning of the first track for devices that use tracks. *Start* is defined when a multimedia device is opened with the *Open* method. *Start* is specified according to the current time format, which is stored in the *TimeFormat* property. *Start* is read only at run time and is unavailable at design time.

S

Example
The following code displays the start position of the Microsoft Video for Windows file in an edit box named *Edit1*. Attach the code to the *OnClick* event handler of a bitmap button named *BitBtn1*. The code assumes Video for Windows has been installed and a video file named NOTES.AVI is present.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  with MediaPlayer1 do
  begin
    try
      FileName := 'NOTES.AVI';
      Open;
```

```
        Edit1.Text := IntToStr(MediaPlayer1.Start);
      except
        MessageDlg(MediaPlayer1.ErrorMessage, mtError, [mbOk], 0);
      end;
    end;
  end;
```

See also

Length property

StartMargin property

Applies to

TTabSet component

Declaration

```
property StartMargin: Integer;
```

The *StartMargin* property determines how far in pixels the first tab appears from the left edge of the tab set control. The default value is 5. Together with the*EndMargin* property, *StartMargin* can play a role in determining how many tabs can fit within the tab set control.

Example

This example displays the tab set control so that the tabs are no closer than 20 pixels from the edge of the tab control on the left and from the scroll buttons on the right:

```
  procedure TForm1.FormCreate(Sender: TObject);
  begin
    with TabSet1 do
    begin
      AutoScroll := True;
      StartMargin := 20;
      EndMargin := 20;
    end;
  end;
```

See also

AutoScroll property, *EndMargin* property

StartPage property

Applies to

TReport component

Declaration

```
property StartPage: Word;
```

The value of the *StartPage* property determines which page you want the report to start printing from. The default value is 1, indicating the first page. You can change that value to begin printing the report on some other page.

Example

The following code determines the page on which to start printing the report from an edit box.

```
Report1.StartPage := StrToInt(Edit1.Text);
```

See also

EndPage property, *PrintCopies* property, *Print* method

StartPos property

Applies to

TMediaPlayer component

Declaration

```
property StartPos: Longint;
```

Run-time only. The *StartPos* property specifies the position within the currently loaded medium from which to begin playing or recording. *StartPos* is specified using the current time format, which is specified in the *TimeFormat* property.

The *StartPos* property affects only the next *Play* or *StartRecording* method called after setting *StartPos*. You must reset *StartPos* to affect any subsequent calls to *Play* or *StartRecording*.

StartPos does not affect the current position of the medium (specified in the *Position* property) until the next *Play* or *StartRecording* method is called.

Example

The following procedure begins playing the .WAV audio file from the middle of the file.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with MediaPlayer1 do begin
    FileName := 'd:\winapps\sounds\cartoon.wav';
    Open;
    StartPos := TrackLength[1] div 2;
    Play;
  end;
end;
```

See also
EndPos property

StartRecording method

Applies to
TMediaPlayer component

Declaration

```
procedure StartRecording;
```

The *StartRecording* method begins recording from the current *Position* or from the position specified in *StartPos*. *StartRecording* is called when the Record button on the media player control is clicked at run time.

Upon completion, *StartRecording* stores a numerical error code in the *Error* property and the corresponding error message in the *ErrorMessage* property.

The *Wait* property determines whether control is returned to the application before the *StartRecording* method has completed. The *Notify* property determines whether *StartRecording* generates an *OnNotify* event.

By default, the *Notify* property becomes *True,* and the *Wait* property becomes *False* upon completion of the *StartRecording* method. However, if you've set these properties to specific values prior to calling *StartRecording*, they remain unchanged.

Example
The following code tells *MediaPlayer1* to start recording.

```
  MediaPlayer1.StartRecording;
```

See also
Capabilities property, *Pause* method, *PauseOnly* method, *Play* method, *Stop* method

StartTransaction method

Applies to
TDataBase component

Declaration

```
procedure StartTransaction;
```

The *StartTransaction* method begins a transaction at the isolation level specified by the *TransIsolation* property. If a transaction is currently active, Delphi will raise an exception.

Modifications made to the database will be held by the server until the *Commit* method is called to commit the changes or the *Rollback* method is called to cancel the changes.

Use this method only when connected to a server database.

Example

```
with Database1 do
  begin
  StartTransaction;
{ Update one or more records in tables linked to Database1 }
...
  Commit;
  end;
```

State property

Applies to

TCheckBox, *TDBCheckBox*, *TDataSource*, *TTable*, *TQuery*, *TStoredProc* components

For check boxes

Declaration

```
property State: TCheckBoxState;
```

The *State* property determines the various states a check box control can have. These are the possible values:

Value	Meaning
cbUnchecked	The check box has no check mark indicating the user hasn't selected the option.
cbChecked	The check box has a check mark in it indicating the user has selected the option.
cbGrayed	The check box is gray indicating a third state that is neither checked nor unchecked. Your application determines the meaning of a grayed check box.

State is a run-time only property of a database check box component.

Example

This code examples uses three check boxes on a form. When the form is created, the code sets the initial state of each of the check boxes: the first check box is checked, the second check box is dimmed (or grayed), and the third check box is unchecked:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  CheckBox1.State := cbChecked;
  CheckBox2.State := cbGrayed;
  CheckBox3.State := cbUnChecked;
end;
```

See also

AllowGrayed property, *Checked* property

For data source components

Declaration

```
property State: TDataSetState;
```

Description

State reads the current status of the dataset component. Possible values are those of the *TDataSetState* type: *dsInactive*, *dsBrowse*, *dsEdit*, *dsInsert*, *dsSetKey*, or *dsCalcFields*. The value of *State* is the same as that of the *State* property of *DataSet*, except that when *Enabled* is *False* or *DataSet* has not been assigned a value, *State* will be *dsInactive*.

Example

```
if DataSource1.Dataset <> nil then
  PostButton.Enabled := DataSource1.State in [dsEdit, dsInsert];
```

For tables, queries, and stored procedures

Declaration

```
property State: TDataSetState;
```

Run-time and read only. The *State* property specifies the current state of the *dataset*. The possible values are those of the *TDataSetState* type:

- *dsInactive* when the dataset is closed
- *dsBrowse* when the dataset is in Browse state
- *dsEdit* when the dataset is in Edit state
- *dsInsert* when the dataset is in Insert state
- *dsSetKey* when the dataset is in SetKey state
- *dsCalcFields* when the *OnCalcFields* event is called.

Example

```
{ Open the dataset if it is not already }
if Table1.State = dsInactive then Table1.Active := True;
```

Step method

Applies to
TMediaPlayer component

Declaration

```
procedure Step;
```

The *Step* method steps forward a number of frames (determined by the *Frames* property) in the currently loaded medium. *Step* is called when the Step button on the media player control is clicked at run time.

Upon completion, *Step* stores a numerical error code in the *Error* property and the corresponding error message in the *ErrorMessage* property.

The *Wait* property determines whether control is returned to the application before the *Step* method has completed. The *Notify* property determines whether *Step* generates an *OnNotify* event.

Example

The following example lets the user pick an .AVI video file using *OpenDialog1* and opens that file in *MediaPlayer1*. The 'Step' button can then be used to step forward through the .AVI clip. This could be used to hide *MediaPlayer1* if you wanted to design your own user interface for the media player.

```
procedure TForm1.FormClick(Sender: TObject);
begin
  OpenDialog1.Filename := '*.*';
  if OpenDialog1.Execute then begin
    MediaPlayer1.Filename := OpenDialog1.Filename;
    MediaPlayer1.Open;
  end;
end;

procedure TForm1.BackClick(Sender: TObject);
begin
  MediaPlayer1.Step;
end;
```

See also

Back method, *Capabilities* property

StmtHandle property

Applies to

TQuery, *TStoredProc* component

Declaration

```
property StmtHandle: HDBIStmt;
```

Run-time and read only. The *StmtHandle* property enables an application to make direct calls to the Borland Database Engine (BDE) API using the result of the last query. Under most circumstances you should not need to use this property, unless your application requires some functionality not encapsulated in the VCL.

Stop method

Applies to
TMediaPlayer component

Declaration

```
procedure Stop;
```

The *Stop* method stops playing or recording. *Stop* is called when the Stop button on the media player control is clicked at run time.

Upon completion, *Stop* stores a numerical error code in the *Error* property, and the corresponding error message in the *ErrorMessage* property.

The *Wait* property determines whether control is returned to the application before the *Stop* method has completed. The *Notify* property determines whether *Stop* generates an *OnNotify* event.

Example
The following procedure stops the currently playing multimedia device when *Button2* is clicked.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  MediaPlayer1.Stop;
end;
```

See also
Pause method, *PauseOnly* method, *Play* method, *StartRecording* method

Storage property

Applies to
TOLEContainer component

Declaration

```
property Storage: IStorage;
```

Read-only. The *Storage* property allows access to the OLE IStorage interface of an OLE container component.

Note The concept of IStorage is described in detail in OLE 2.0 documentation such as the Microsoft OLE 2.0 SDK.

See also
LoadFromFile method, *SaveToFile* method

StoredProcName property

Applies to

TStoredProc component

Declaration

```
property StoredProcName: string;
```

StoredProcName is the name of the stored procedure on the server.

Oracle servers allow more than one stored procedure with the same name. Set the *Overload* property to specify the procedure to execute on an Oracle server.

Example

```
StoredProc1.StoredProcName := 'FOO';
```

Str procedure **System**

Declaration

```
procedure Str(X [: Width [: Decimals ]]; var S);
```

The *Str* procedure converts *X* to a string representation according to the *Width* and *Decimals* formatting parameters. The effect is like a call to *Write* except the resulting string is stored in *S* instead of being written to a text file.

X is an integer-type or real-type expression. *Width* and *Decimals* are integer-type expressions. *S* is a string-type variable or a zero-based character array variable if extended syntax is enabled.

Example

```
function MakeItAString(I: Longint): string;
{ Convert any integer type to a string }
 var
  S: string[11];
begin
  Str(I, S);
  IntToStr := S;
end;

begin
  Canvas.TextOut(10, 10, MakeItAString(-5322));
end;
```

See also

Val procedure, *Write* procedure

S

StrAlloc function **SysUtils**

Declaration

```
function StrAlloc(Size: Word): PChar;
```

This function allocates a buffer for a null-terminated string with a maximum length of *Size* - 1 (1 byte must be reserved for the termination character). The maximum value of *Size* is 65,526. The result points to the location where the first character of the string is to be stored. A 16-bit number giving the total amount of memory allocated is stored in the two bytes preceding the first character; it is equal to *Size* + 2. If space for a string is allocated with *StrAlloc*, it should be deallocated via *StrDispose*.

StrAlloc is used by *NewStr* and is a general purpose routine.

StrBufSize function **SysUtils**

Declaration

```
function StrBufSize(Str: PChar): Word;
```

This function returns the maximum number of characters that can be stored in a string buffer allocated by *StrAlloc*. This number includes the termination character. If *Str* does not point to a string buffer allocated by *StrAlloc*, no error message is returned, and the result is unpredictable.

StrCat function **SysUtils**

Declaration

```
function StrCat(Dest, Source: PChar): PChar;
```

The *StrCat* function appends a copy of *Source* to the end of *Dest* and returns the concatenated string.

StrCat does not perform any length checking. The destination buffer must have room for at least *StrLen*(*Dest*)+*StrLen*(*Source*)+*1* characters.

If you want length checking, use the *StrLCat* function.

Example

```
uses SysUtils;

const
  Obj: PChar = 'Object';
  Pascal: PChar = 'Pascal';
 var
  S: array[0..15] of Char;
begin
  StrCopy(S, Obj);
```

```
    StrCat(S, ' ');
    StrCat(S, Pascal);
    Canvas.TextOut(10, 10, StrPas(S));
  end;
```

See also
StrLCat function

StrComp function
SysUtils

Declaration

```
function StrComp(Str1, Str2 : PChar): Integer;
```

The *StrComp* function compares *Str1* to *Str2*.

Return value	Condition
<0	if *Str1*< *Str2*
=0	if *Str1*= *Str2*
>0	if *Str1* > *Str2*

Example

```
  uses SysUtils;

  const
    S1: PChar = 'Wacky';
    S2: PChar = 'Code';

  var
    C: Integer;
    Result: string;
  begin
    C := StrComp(S1, S2);
    if C < 0 then Result := ' is less than ' else
      if C > 0 then Result := ' is greater than ' else
        Result := ' is equal to ';
    Canvas.TextOut(10, 10, StrPas(S1) + Result + StrPas(S2));
  end;
```

See also
StrIComp function, *StrLComp* function, *StrLIComp* function

StrCopy function
SysUtils

Declaration

```
function StrCopy(Dest, Source: PChar): PChar;
```

The *StrCopy* function copies *Source* to *Dest* and returns *Dest*.

StrCopy does not perform any length checking. The destination buffer must have room for at least *StrLen*(*Source*)+*1* characters.

If you want to use length checking, use the *StrLCopy* function.

Example

```
uses SysUtils;

var
  S: array[0..12] of Char;
begin
  StrCopy(S, 'ObjectPascal');
  Canvas.TextOut(10, 10, StrPas(S));
end;
```

See also

StrECopy function, *StrLCopy* function

StrDispose function SysUtils

Declaration

```
function StrDispose(Str: PChar);
```

The *StrDispose* function disposes of a string on a heap that was previously allocated with *StrNew*.

If *Str* is **nil**, *StrDispose* does nothing.

See also

StrNew function

StrECopy function SysUtils

Declaration

```
function StrECopy(Dest, Source: PChar): PChar;
```

The *StrECopy* function copies *Source* to *Dest* and returns *StrEnd*(*Dest*).

StrECopy does not perform any length checking. The destination buffer must have room for at least *StrLen*(*Source*)+*1* characters.

Nested calls to *StrECopy* to concatenate a sequence of strings will run more efficiently than multiple calls to *StrCat*.

Example

```
uses SysUtils;

const
  Turbo: PChar = 'Object';
  Pascal: PChar = 'Pascal';
 var
  S: array[0..15] of Char;
begin
  StrECopy(StrECopy(StrECopy(S, Turbo), ' '), Pascal);
  Canvas.TextOut(10, 10, StrPas(S));
end;
```

See also

StrCat function, *StrCopy* function, *StrEnd* function

StrEnd function

SysUtils

Declaration

```
function StrEnd(Str: PChar): PChar;
```

The *StrEnd* function returns a pointer to the null character at the end of *Str*.

Example

```
uses SysUtils;

const
  S: PChar = 'Yankee Doodle';
begin
  Canvas.TextOut(5, 10, 'The string length of "' + StrPas(S) + '" is ' +
    IntToStr(StrEnd(S) - S));
end;
```

See also

StrLen function

Stretch property

Applies to

TImage, *TDBImage* components

Declaration

```
property Stretch: Boolean;
```

S

Setting the *Stretch* property to *True* permits bitmaps and metafiles to assume the size and shape of the image control. When the image control is resized, the image resizes also. The *Stretch* property has no affect on icons.

If you prefer to have the image control resize to fit the native size of the image, set the *AutoSize* property to *True*.

Example

This example uses an image component on a form. When the form is created, the specified image is loaded and stretched to fit the boundaries of the image component.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Image1.Stretch := True;
  Image1.Picture.LoadFromFile('C:\DELPHI\DEMOS\GRAPHEX\PASTE.BMP');
end;
```

See also

AutoSize property, *LoadFromFile* method, *Picture* property

StretchDraw method

Applies to

TCanvas object

Declaration

```
procedure StretchDraw(const Rect: TRect; Graphic: TGraphic);
```

This method draws the graphic specified by the *Graphic* parameter in the rectangle specified by the *Rect* parameter. Use this method to stretch or resize a graphic to the size of the rectangle.

Example

The following code stretches the bitmap to fill the client area of *Form1*.

```
Form1.Canvas.StretchDraw(Form1.ClientRect, TheGraphic);
```

StrFmt function **SysUtils**

Declaration

```
function StrFmt(Buffer, Format: PChar; const Args: array of const): PChar;
```

This function formats the series of arguments in the open array *Args*. Formatting is controlled by the null-terminated format string *Format*; the results are returned in *Buffer*. The function result contains a pointer to the destination buffer.

For information on the format strings, see Format Strings.

Strings property

Applies to

TStringList, *TStrings* objects

Declaration

```
property Strings[Index: Integer]: string;
```

Run-time only. With the *Strings* property, you can access a specific string of a string or string list object. Specify the position of the string in the string list as the value of the *Index* parameter. The index of the *Strings* property is zero-based, so the first string has an *Index* value of 0, the second has an *Index* value of 1, and so on. To find out what the index of a particular string is, call the *IndexOf* method.

Strings is the default property of string objects. Therefore, you can safely omit the reference to the *Strings* identifier and just the treat the string object itself as an indexed array of strings. In the following example, *Lines* is a string object property of a memo component (*TMemo*). These two lines of code are both acceptable and do the same thing:

```
Memo1.Lines.Strings[0] := ‘This is the first line’;
Memo1.Lines[0] := ‘This is the first line’;
```

Example

This example uses a list box and a button on a form. When the form is created, three string are added to the list box. When the user clicks the button, each of the strings of the *Items* property, a *TStrings* object, changes:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ListBox1.Items.Add('One');
  ListBox1.Items.Add('Two');
  ListBox1.Items.Add('Three');
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Strings[0] := 'First';
  ListBox1.Items.Strings[1] := 'Second';
  ListBox1.Items.Strings[2] := 'Third';
end;
```

Because *Strings* is the default property of a string object, you can omit the reference to *Strings* in the preceding code. For example, you can write the code like this:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ListBox1.Items.Add('One');
  ListBox1.Items.Add('Two');
  ListBox1.Items.Add('Three');
end;

procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  ListBox1.Items[0] := 'First';
  ListBox1.Items[1] := 'Second';
  ListBox1.Items[2] := 'Third';
end;
```

See also

Add method, *AddObject* method, *AddStrings* method, *Assign* method, *Clear* method, *Count* property, *Delete* method, *Exchange* method, *IndexOf* method, *LoadFromFile* method, *Objects* property, *SaveToFile* method

StrLCat function SysUtils

Declaration

```
function StrLCat(Dest, Source: PChar; MaxLen: Word): PChar;
```

The *StrLCat* function appends at most *MaxLen* – *StrLen*(*Dest*) characters from *Source* to the end of *Dest* and returns *Dest*. The *SizeOf* standard function can be used to determine the *MaxLen* parameter.

Example

```
uses SysUtils;

var
  S: array[0..13] of Char;
begin
  StrLCopy(S, 'Object', SizeOf(S) - 1);
  StrLCat(S, ' ', SizeOf(S) - 1);
  StrLCat(S, 'Pascal', SizeOf(S) - 1);
  Canvas.TextOut(10, 10, StrPas(S));
end;
```

See also

SizeOf function, *StrCat* function

StrIComp function SysUtils

Declaration

```
function StrIComp(Str1, Str2:PChar): Integer;
```

The *StrIComp* function compares *Str1* to *Str2* without case sensitivity. The return value is the same as *StrComp*.

Example

```
uses SysUtils;
```

```
const
  S1: PChar = 'Wacky';
  S2: PChar = 'Code';

var
  C: Integer;
  Result: string;
begin
  C := StrIComp(S1, S2);
  if C < 0 then Result := ' is less than ' else
    if C > 0 then Result := ' is greater than ' else
      Result := ' is equal to ';
  Canvas.TextOut(10, 10, StrPas(S1) + Result + StrPas(S2));
end;
```

See also

StrComp function, *StrLComp* function, *StrLIComp* function

StrLComp function — SysUtils

Declaration

```
function StrLComp(Str1, Str2: PChar; MaxLen: Word): Integer;
```

The *StrLComp* function compares *Str1* to *Str2*, up to a maximum length of *MaxLen* characters. The return value is the same as *StrComp*.

Example

```
uses SysUtils;

const
  S1: PChar = 'Enterprise'
  S2: PChar = 'Enter'

var
  Result: string;
begin
  if StrLComp(S1, S2, 5) = 0 then
    Result := 'equal'
  else
    Result := 'different';
  Canvas.TextOut(10, 10, 'The first five characters are ' + Result);
end;
```

See also

StrComp function, *StrIComp* function, *StrLIComp* function

StrLCopy function SysUtils

Declaration

```
function StrLCopy(Dest, Source: PChar; MaxLen: Cardinal): PChar;
```

The *StrLCopy* function copies at most *MaxLen* characters from *Source* to *Dest* and returns *Dest*. The *SizeOf* standard function can be used to determine the *MaxLen* parameter.

Example

```
uses SysUtils;

var
  S: array[0..11] of Char;
begin
  StrLCopy(S, 'ObjectPascal', SizeOf(S) - 1);
  Canvas.TextOut(10, 10, StrPas(S));
end;
```

See also

SizeOf function, *StrCopy* function

StrLen function SysUtils

Declaration

```
function StrLen(Str: PChar): Cardinal;
```

The *StrLen* function returns the number of characters in *Str*, not counting the null terminator.

Example

```
uses SysUtils;

const
  S: PChar = 'E Pluribus Unum';
begin
  Canvas.TextOut(5, 10, 'The string length of “' + StrPas(S) + '” is ' +
    IntToStr(StrLen(S)));
end;
```

See also

StrEnd function

StrLFmt function **SysUtils**

Declaration

```
function StrLFmt(Buffer: PChar; MaxLen: Word; Format: PChar; const Args: array of const):
PChar;
```

This function formats the series of arguments in the open array *Args*. Formatting is controlled by the null-terminated format string *Format*; the results are returned in *Buffer*, whose maximum length is given by *MaxLen*. The function result contains a pointer to the destination buffer.

For information on the format strings, see Format Strings.

See also

FormatBuf function, *StrFmt* function, *FmtStr* procedure

StrLIComp function **SysUtils**

Declaration

```
function StrLIComp(Str1, Str2: PChar; MaxLen: Word): Integer;
```

StrLIComp compares *Str1* to *Str2* , up to a maximum length of *MaxLen* characters, without case sensitivity.

The return value is the same as *StrComp*.

Example

```
uses SysUtils;

const
  S1: PChar = 'Enterprise'
  S2: PChar = 'Enter'

var
  Result: string;
begin
  if StrLIComp(S1, S2, 5) = 0 then
    Result := 'equal'
  else
    Result := 'different';
  Canvas.TextOut(10, 10, 'The first five characters are ' + Result);
end;
```

See also

StrComp function, *StrIComp* function, *StrLComp* function

S

StrLower function — SysUtils

Declaration

```
function StrLower(Str: PChar): PChar;
```

The *StrLower* function converts *Str* to lowercase and returns *Str*.

Example

```
uses SysUtils;

const
  S: PChar = 'A fUnNy StRiNg'
begin
  Canvas.TextOut(5, 10, StrPas(StrLower(S)) + ' ' + StrPas(StrUpper(S)));
end;
```

See also

StrUpper function

StrMove function — SysUtils

Declaration

```
function StrMove(Dest, Source: PChar; Count: Cardinal): PChar;
```

The *StrMove* function copies exactly *Count* characters from *Source* to *Dest* and returns *Dest*. *Source* and *Dest* can overlap.

Example

```
uses SysUtils;

function AHeapaString(S: PChar): PChar;
{ Allocate string on heap }
 var
  L: Word;
  P: PChar;
begin
 StrNew := nil;
 if (S <> nil) and (S[0] <> #0) then
  begin
    L := StrLen(S) + 1;
    GetMem(P, L);
    StrNew := StrMove(P, S, L);
  end;
end;

procedure DisposeDaString(S: PChar);
{ Dispose string on heap }
begin
```

```
    if S <> nil then FreeMem(S, StrLen(S) + 1);
  end;

  var
    S: PChar;
  begin
    AHeapaString(S);
    DisposeDaString(S);
  end;
```

See also

Move procedure

StrNew function **SysUtils**

Declaration

```
function StrNew(Str: PChar): PChar;
```

The *StrNew* function allocates a copy of *Str* on the heap.

If *Str* is **nil** or points to an empty string, *StrNew* returns **nil** and does not allocate any heap space.

Otherwise, *StrNew* makes a duplicate of *Str*, obtaining space with a call to the *StrAlloc* procedure, and returns a pointer to the duplicated string.

The allocated space is *StrLen*(*Str*) + *3* bytes long.

Example

```
  uses SysUtils;

  const
    S: PChar = 'Nevermore';
   var
    P: PChar;
  begin
    P := StrNew(S);
    Canvas.TextOut(10, 10, StrPas(P));
    StrDispose(P);
  end;
```

See also

GetMem procedure, *StrDispose* function

StrPas function **SysUtils**

Declaration

```
function StrPas(Str: PChar): string;
```

The *StrPas* function converts the null-terminated *Str* to a Pascal-style string.

Example

```
uses SysUtils;

const
  A: PChar = 'I love the smell of Object Pascal in the morning.';
 var
  S: string[79];
begin
  S := StrPas(A);
  Canvas.TextOut(10, 10, S);
end;
```

See also
StrPCopy function

StrPCopy function — SysUtils

Declaration

```
function StrPCopy(Dest: PChar; Source: string): PChar;
```

The *StrPCopy* function copies a Pascal-style string *Source* into a null-terminated string *Dest*.

StrPCopy does not perform any length checking.

The destination buffer must have room for at least *Length*(*Source*)+*1* characters.

Example

```
uses SysUtils;

var
  A: array[0..79] of Char;
begin
  S := 'Honk if you know Blaise.';
  StrPCopy(A, S);
  Canvas.TextOut(10, 10, StrPas(A));
end;
```

See also
StrCopy procedure

StrPLCopy function SysUtils

Declaration

```
function StrPLCopy(Dest: PChar; const Source: string; MaxLen: Word): PChar;
```

StrPLCopy copies a maximum of *MaxLen* characters from the Pascal-style string *Source* into the null-terminated string *Dest*. *Dest* is also returned as the function result.

StrPos function SysUtils

Declaration

```
function StrPos(Str1, Str2: PChar): PChar;
```

The *StrPos* function returns a pointer to the first occurrence of *Str2* in *Str1*.

If *Str2* does not occur in *Str1, StrPos* returns **nil**.

Example

```
uses SysUtils;

const
  S: PChar = 'Ready, Set, Go! ';
  SubStr: PChar = 'Set';

var
  P: PChar;
begin
  P := StrPos(S, SubStr);
  if P = nil then
    Canvas.TextOut(10, 10, 'Substring not found')
  else
    Canvas.TextOut(10, 10, 'Substring found at index ' + IntToStr(P - S));
end;
```

S

StrRScan function SysUtils

Declaration

```
function StrRScan(Str: PChar; Chr: Char): PChar;
```

The *StrRScan* function returns a pointer to the last occurrence of *Chr* in *Str*.

If *Chr* does not occur in *Str, StrRScan* returns **nil**. The null terminator is considered to be part of the string.

Example

```
{ Return pointer to name part of a full path name }
```

```
uses SysUtils;

function NamePart(FileName: PChar): PChar;
 var
  P: PChar;
begin
  P := StrRScan(FileName, '\');
  if P = nil then
  begin
    P := StrRScan(FileName, ':');
    if P = nil then P := FileName;
  end;
  NamePart := P;
end;

var
  S : string;
begin
  S := StrPas(NamePart('C:\Test.fil'));
  Canvas.TextOut(10, 10, S);
end;
```

See also

StrScan function

StrScan function — SysUtils

Declaration

```
function StrScan(Str: PChar; Chr: Char): PChar;
```

The *StrScan* function returns a pointer to the first occurrence of *Chr* in *Str*.

If *Chr* does not occur in *Str, StrScan* returns **nil**. The null terminator is considered to be part of the string.

Example

```
uses SysUtils;

function HasWildcards(FileName: PChar): Boolean;
{ Return true if file name has wildcards in it }
begin
  HasWildcards := (StrScan(FileName, '*') <> nil) or
    (StrScan(FileName, '?') <> nil);
end;

const
  P: PChar = 'C:\Test.* ';
begin
  if HasWildcards(P) then
    Canvas.TextOut(20, 20, 'The string has wildcards')
  else
```

```
    Canvas.TextOut(20, 20, 'The string doesn't have wildcards')
end;
```

See also

StrRScan function

StrToDate function SysUtils

Declaration

```
function StrToDate(const S: string): TDateTime;
```

The *StrToDate* function converts a string to date format. The date in the string must be a valid date.

The string must consist of two or three numbers, separated by the character defined by the *DateSeparator* global variable. The order for month, day, and year is determined by the *ShortDateFormat* global variable--possible combinations are m/d/y, d/m/y, and y/m/d.

If the string contains only two numbers, it is interpreted as a date (m/d or d/m) in the current year. Year values between 0 and 99 are assumed to mean 1900 to 1999.

If the given string does not contain a valid date, an *EConvertError* exception is raised.

Note The correct format of the date string varies if you change the value of some of the date and time typed constants.

Example

This example uses an edit box, a label, and a button on a form. When the user enters a date in the edit box in the MM/DD/YY format, the string entered is converted to a *TDateTime* value. This value is then converted back to a string value so it can appear as the caption of the label:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ADate: TDateTime;
begin
  ADate := StrToDate(Edit1.Text);
  Label1.Caption := DateToStr(ADate);
end;
```

See also

DateToStr function, *StrToDateTime* function, *StrToTime* function

StrToDateTime function SysUtils

Declaration

```
function StrToDateTime(const S: string): TDateTime;
```

The *StrToDateTime* function converts a string into a date and time format. The string specified as the *S* parameter must be in the MM/DD/YY HH:MM:SS format unless the value of the value of the date and time typed constants has changed. Specifying AM or PM as part of the time is optional, as are the seconds. You should use 24-hour time (7:45 PM is entered as 19:45, for example) if you don't specify AM or PM.

Note You must use another format to specify a date and time string if you change the value of the some of the date and time typed constants.

Example

This example uses an edit box, a label, and a button on the form. When the user enters a date and time in the edit box in the MM/DD/YY HH:MM:SS format, the string entered is converted to a *TDateTime* value. This value is then converted back to a string value so it can appear as the caption of the label:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ADateAndTime: TDateTime;
begin
  ADateAndTime := StrToDateTime(Edit1.Text);
  Label1.Caption := DateTimeToStr(ADateAndTime);
end;
```

See also

DateTimeToStr function, *StrToDate* function, *StrToTime* function

StrToFloat function SysUtils

Declaration

```
function StrToFloat(const S: string): Extended;
```

StrToFloat converts the given string to a floating-point value. The string must consist of an optional sign (+ or –), a string of digits with an optional decimal point, and an optional 'E' or 'e' followed by a signed integer. Leading and trailing blanks in the string are ignored.

The *DecimalSeparator* global variable defines the character that must be used as a decimal point. Thousand separators and currency symbols are not allowed in the string. If the string doesn't contain a valid value, an *EConvertError* exception is raised.

StrToInt function SysUtils

Declaration

```
function StrToInt(const S: string): Longint;
```

The *StrToInt* function converts a string representing an integer-type number in either decimal or hexadecimal notation into a number. If the string does not represent a valid number, *StrToInt* raises an *EConvertError* exception.

Example

This example uses an edit box and a button on a form. When the user clicks the button, the code converts the string '22467' into an integer, increments that value, then reconverts that new value back to an a string so that it can display in the edit box:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  S: string;
  I: Integer;
begin
  S := '22467';
  I := StrToInt(S);
  Inc(I);
  Edit1.Text := IntToStr(I);
end;
```

See also

IntToHex function, *IntToStr* function, *StrToIntDef* function

StrToIntDef function — Sysutils

Declaration

```
function StrToIntDef(const S: string; Default: Longint): Longint;
```

The *StrToIntDef* function converts the string passed in *S* into a number. If *S* does not represent a valid number, *StrToIntDef* returns the number passed in *Default*.

Example

This example uses two edit boxes and a button on a form. The user enters a number in the first edit box and clicks the button. If the number entered was a valid integer, the same value appears in the second edit box. If the number was not a valid integer, the default value of 1000 appears in the second edit box:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NumberString: string;
  Number: Integer;
begin
  NumberString := Edit1.Text;
  Number := StrToIntDef(NumberString, 1000);
  Edit2.Text := IntToStr(Number);
end;
```

See also

IntToStr function, *StrToInt* function

StrToTime function SysUtils

Declaration

```
function StrToTime(const S: string): TDateTime;
```

The *StrToTime* function converts a string to a *TDateTime*. Specify the time string in the HH:MM:SS format. Specifying AM or PM is optional, as are the seconds. You should use 24 hour time (7:45 PM is entered as 19:45, for example) if you don't specify AM or PM. You can use another format if you change the value of the some of the date and time typed constants. If the given string does not contain a valid time an *EConvertError* exception is raised.

Example

This example uses an edit box, a label, and a button on a form. When the user enters a time in the edit box in the HH:MM:SS format, the string entered is converted to a *TDateTime* value. This value is then converted back to a string value so it can appear as the caption of the label:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ATime: TDateTime;
begin
  ATime := StrToTime(Edit1.Text);
  Label1.Caption := TimeToStr(ATime);
end;
```

See also

StrToDate function, *StrToDateTime* function, *TimeToStr* function

StrUpper function SysUtils

Declaration

```
function StrUpper(Str: PChar): PChar;
```

The *StrUpper* function converts *Str* to uppercase and returns *Str*.

Example

```
uses SysUtils;

const
  S: PChar = 'A fUnNy StRiNg'
begin
```

```
  Canvas.TextOut(5, 10, StrPas(StrLower(S)) + ' ' + StrPas(StrUpper(S)));
end;
```

See also
StrLower function

Style property

Applies to
TFont, *TPen* objects; *TBevel*, *TBitBtn*, *TComboBox*, *TDBComboBox*, *TDBListBox*, *TDBLookupCombo*, *TListBox*, *TOutline*, *TTabSet* components

For pen objects

Declaration

```
property Style: TPenStyle;
```

The *Style* property determines the style in which the pen draws lines. The following table shows the different style values and what they produce:.

Style	Meaning
psSolid	The pen draws a solid line.
psDash	The pen draws a line made up of a series of dashes.
psDot	The pen draws a line made up of a series of dots.
psDashDot	The pen draws a line made up of alternating dashes and dots.
psDashDotDot	The pen draws a line made up of a series of dash-dot-dot combinations.
psClear	The pen draws lines made up no visible marks.
psInsideFrame	The pen draws lines within the frame of closed shapes that specify a bounding rectangle.

Example
This example uses two radio buttons on a form. When the user drags the mouse pointer across the form, lines are drawn. The user can use the two radio buttons to choose between two pen styles. Selecting the first radio button draws a dotted line. Selecting the second radio button draws a solid line.

```
var
  Drawing: Boolean;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
  Canvas.MoveTo(X, Y);
end;

procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
```

```
begin
  if Drawing then
    Canvas.LineTo(X, Y);
end;

procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);
  Drawing := False;
end;

procedure TForm1.RadioButton1Click(Sender: TObject);
begin
  Canvas.Pen.Style := psDot;
end;

procedure TForm1.RadioButton2Click(Sender: TObject);
begin
  Canvas.Pen.Style := psSolid;
end;
```

For brushes

Declaration

```
property Style: TBrushStyle;
```

The *Style* property of a brush determines the brush's pattern for painting backgrounds of windows or graphic shapes. The following table shows the different values for *Style* and the resulting patterns:

Hatch	Pattern	Hatch	Pattern
bsSolid	■	*bsCross*	▦
bsClear		*bsDiagCross*	▩
bsBDiagonal	▨	*bsHorizontal*	▤
bsFDiagonal	▧	*bsVertical*	▥

Example

This example displays a rectangle filled with red horizontal stripes whenever a form *OnPaint* event occurs.

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

```
    with Canvas do
    begin
      Brush.Style := bsHorizontal;
      Brush.Color := clRed;
      Rectangle(12, 50, 100, 200);
    end;
  end;
```

For fonts

Declaration

```
property Style: TFontStyles;
```

The *Style* property determines whether the font is normal, italic, underlined, bold, and so on. These are the possible values:

Value	Meaning
fsBold	The font is boldfaced.
fsItalic	The font is italicized.
fsUnderline	The font is underlined.
fsStrikeout	The font is displayed with a horizontal line through it.

The *Style* property is a set, so it can contain multiple values. For example, a font could be both boldfaced and italicized.

Example

The following code boldfaces the font used in the memo..

```
  Memo1.Font.Style := [fsBold];
```

For combo boxes

Declaration

```
property Style: TComboBoxStyle;
```

The *Style* property determines how a combo box displays its items. By default, *Style* is *csDropDown*, meaning that the combo box displays each item as a string in a drop-down list. By changing the value of *Style*, you can create owner-draw combo boxes, meaning that items can be graphical of either fixed or varying height. You can set *Style* to any of the following values:

Value	Meaning
csDropDown	Creates a drop-down list with an edit box in which the user can enter text. All items are strings, with each item having the same height. For database combo boxes, the combo box displays the contents of the field of the current records. The user can choose another item from the drop-down list and change the value of the field or type a new value in the edit box.
csSimple	Creates an edit box with no list. For database combo boxes, the current contents of the linked field displays in the combo box. The user can change the contents of the field by typing in a new value.
csDropDownList	Creates a drop-down list with no attached edit box, so the user can't edit an item or type in a new item. All items are strings, with each item having the same height. For database combo boxes, the edit box is blank unless the current contents of the field matches one of the specified *Items* in the drop-down list. The user can change the contents of the field only by selecting one of the strings from the drop-down list.
csOwnerDrawFixed	Each item in the combo box is the height specified by the *ItemHeight* property. For database combo boxes, the combo box is blank unless the current contents of the field matches one of the specified *Items* in the drop-down list. The user can change the contents of the field only by selecting one of the strings from the drop-down list.
csOwnerDrawVariable	Items in the combo box can be of varying heights. For database combo boxes, the combo box is blank unless the current contents of the field matches one of the specified *Items* in the drop-down list. The user can change the contents of the field only by selecting one of the strings from the drop-down list.

Owner-draw combo boxes can display items other than strings. For example, a combo box could display graphical images along with or instead of its strings. Owner-draw combo boxes require more programming, however, as the application needs information on how to render the image for each item in the list.

Each time an item is displayed in an *csOwnerDrawFixed* combo box, the *OnDrawItem event* occurs. The event handler for *OnDrawItem* draws the specified item. The *ItemHeight property* determines the height of all the items.

Each time an item is displayed in an *csOwnerDrawVariable* combo box, two events occur. The first is the *OnMeasureItem event*. The event handler for *OnMeasureItem* can set the height of each item. Then the *OnDrawItem* event occurs. The *OnDrawItem* handler draws each item in the list box using the size specified by the *OnMeasureItem* handler.

Example

This example uses a combo box and a check box on a form. If the user checks the check box, the combo box becomes a drop-down list. When the user unchecks the check box, the combo box becomes a simple combo box:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if CheckBox1.Checked then
```

```
      ComboBox1.Style := csDropDownList
    else
      ComboBox1.Style := csSimple;
  end;
```

See also

ReadOnly property, *Creating an owner-draw control*

For list boxes

Declaration

```
property Style: TListBoxStyle
```

The *Style* property determines how a list box displays its items. By default, *Style* is *lbStandard*, meaning that the list box displays each item as a string. By changing the value of *Style*, you can create owner-draw list boxes, meaning that items can be graphical and of either fixed or varying height. These are the possible values for *Style*:

Value	Meaning
lbStandard	All items are strings, with each item the same height.
lbOwnerDrawFixed	Each item in the list box is the height specified by the *ItemHeight* property.
lbOwnerDrawVariable	Items in the list box can be of varying heights.

Owner-draw list boxes can display items other than strings. For example, a list box could display graphical images along with or instead of its strings. Owner-draw list boxes require more programming, however, because the application needs information on how to render the image for each item in the list.

Each time an item is displayed in an *lbOwnerDrawFixed* list box, the *OnDrawItem event* occurs. The event handler for *OnDrawItem* draws the specified item. The *ItemHeight' property* determines the height of all the items.

Each time an item is displayed in an *lbOwnerDrawVariable* list box, two events occur. The first is the *OnMeasureItem event*. The code you write for the *OnMeasureItem* handler can set the height of each item. Then the *OnDrawItem* event occurs. The code you write for the *OnDrawItem* handler draws each item in the list box using the size specified by the *OnMeasureItem* handler.

Example

This example uses a list box and a check box. When the user checks the check box, the list box becomes an fixed owner-draw list box. When the user unchecks the check box, the list box becomes a standard list box:

```
  procedure TForm1.CheckBox1Click(Sender: TObject);
  begin
    if CheckBox1.Checked then
      ListBox1.Style := lbOwnerDrawFixed
    else
    ListBox1.Style := lbStandard;
end;
```

For bitmap buttons

Declaration

```
property Style: TButtonStyle;
```

The *Style* property of a bitmap button determines the appearance of a bitmap button. These are the possible values:

Value	Meaning
bsAutoDetect	When you are using Windows 3.x, the bitmap button uses the standard Windows 3.x look. When you are using a later version of Windows, the bitmap button uses a newer look.
bsWin31	Uses the standard Windows 3.1 look, regardless of which version of Windows you are running.
bsNew	Uses a new bitmap button look, regardless of which version of Windows you are running.

Example

This example uses a bitmap button and a check box on a form. When the user checks the check box, the bitmap button assumes the new bitmap style. When the user unchecks the check box, the bitmap button takes on the Windows 3.1 look:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if CheckBox1.Checked then
    BitBtn1.Style := bsNew
  else
    BitBtn1.Style := bsWin31;
end;
```

See also

Kind property

For tab set controls

Declaration

```
property Style: TTabStyle;
```

The *Style* property of a tab set control (*TTabSet* component) determines how a tab appears. These are the possible values:

Value	Meaning
tsStandard	Each tab has the standard size and look.
tsOwnerDraw	Each tab has the height specified with the *TabHeight* property and width needed to hold the text or glyph.

Owner-draw tabs can display objects other than strings, such as graphical images. Owner-draw tabs require more programming, however, as the application needs information on how to render the image for each tab in the tab set control.

Each time an item is displayed in an *tsOwnerDraw* tab, two events occur. The first is the *OnMeasureTab* event. In the *OnMeasureTab* event handler, you write the code that calculates the width of the tab needed to hold the text or graphical image. After the OnMeasureTab event, the *OnDrawTab* event occurs. The code you write for the *OnDrawTab* event draws the tab and its contents using the width found with the *OnMeasureTab* event and the height specified as the value of the *TabHeight* property.

Example

When this example runs, the tab set on the form becomes an owner-draw tab set:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  TabSet1.Style := tsOwnerDraw;
end;
```

See also

TTabStyle type

For outlines

Declaration

```
property Style: TOutlineType;
```

The *Style* property determines how a outline displays its items. By default, *Style* is *osStandard*, meaning that the outline displays items in the style determined by the *OutlineStyle* property. By changing the value of *Style* to *otOwnerDraw*, you can create owner-draw outlines, meaning that items are drawn on the *Canvas* of the *TOutline* component by code that you write. These are the possible values for *Style*:

Value	Meaning
otStandard	Items are drawn according to the setting of *OutlineStyle.*
otOwnerDraw	Items are drawn on the *Canvas* by your code.

Owner-draw outlines can display items other than the *Text* of an item and the standard bitmaps specified in the *PictureClosed*, *PictureOpen*, *PictureMinus*, *PicturePlus*, and *PictureLeaf* properties. Owner-draw outlines require more programming, however, as the application needs information on how to render the image for each item in the list.

Each time an item is displayed in an *otOwnerDraw* outline, the *OnDrawItem event* occurs. The event handler for *OnDrawItem* draws the specified item. The *ItemHeight property* determines the height of all the items.

Example

The following code sets the style of *Outline1* to owner-draw.

```
Outline1.Style := otOwnerDraw;
```

For bevels

Declaration

property Style: TBevelStyle;

The value of the *Style* property determines if the bevel is raised or lowered. These are the possible values:

Value	Meaning
bsLowered	The bevel is lowered.
bsRaised	The bevel is raised.

Example

This example uses a bevel and a check box on a form. When the check box is checked, the bevel is raised. When the check box is unchecked, the bevel is lowered:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if CheckBox1.Checked then
    Bevel1.Style := bsRaised
  else
    Bevel1.Style := bsLowered;
end;
```

For database lookup combo boxes

Declaration

property Style: TDBLookupComboStyle;

The *Style* property determines how a database lookup combo box displays its items. These are the possible values:

Value	Meaning
csDropDown	Creates a drop-down list with an edit box in which the user can enter text.
csDropDownList	Creates a drop-down list with no attached edit box, so the user can't edit an item or type in a new item.

The default value is *csDropDown*.

Note If the value of the *LookupDisplay* property differs from the value of the *LookupField* property, the database lookup combo box will function as if its *Style* is *csDropDownList*, regardless of the value of the *Style* property.

\Example

The following code sets the style of *DBLookupCombo1* to have a drop-down list with no edit box.

```
DBLookupCombo1.Style := csDropDownList;
```

Succ function System

Declaration

```
function Succ(X);
```

The *Succ* function returns the successor of the argument.

X is an ordinal-type expression. The result, of the same type as *X*, is the successor of *X*.

Example

```
uses Dialogs;

type
  Colors = (RED,BLUE,GREEN);
 var
   S: string;
 begin
   S := 'The predecessor of 5 is ' + IntToStr(Pred(5)) + #13#10;
   S := S + 'The successor of 10 is ' + IntToStr(Succ(10)) + #13#10;
   if Succ(RED) = BLUE then
     S := S + 'In the type Colors, RED is the predecessor of BLUE.';
   MessageDlg(S, mtInformation, [mbOk], 0);
 end;
```

See also

Dec procedure, *Inc* procedure, *Pred* function

Swap function System

Declaration

```
function Swap(X);
```

The *Swap* function exchanges the high-order bytes with the low-order bytes of the argument.

X is an expression of type *Integer* or *Word*.

Example

```
var
  X: Word;
begin
```

```
    X := Swap($1234);   { $3412 }
  end;
```

See also
Hi function, *Lo* function

TableName property

Applies to
TTable component

Declaration

```
property TableName: TFileName;
```

The *TableName* property is the name of the database table to which the *TTable* is linked.

Note The *TTable* must be closed to change this property.

TableType property

Applies to
TTable component

Declaration

```
property TableType: TTableType
```

The *TableType* property specifies the type of the underlying database table. This property is not used for SQL tables.

If *TableType* is set to *Default*, the table's file-name extension determines the table type:

- Extension of .DB or no file-name extension: Paradox table
- Extension of .DBF : dBASE table
- Extension of .TXT : ASCII table

If the value of *TableType* is not *Default*, then the table will always be of the specified *TableType*, regardless of file-name extension:

- *ttASCII*: Text file
- *ttDBase*: dBASE table
- *ttParadox*: Paradox table

Note The *TTable* must be closed to change this property.

See also
CreateTable method, *TableName* property

Tag property

Applies to
All components

Declaration

```
property Tag: Longint;
```

The *Tag* property is available to store an integer value as part of a component. While the *Tag* property has no meaning to Delphi, your application can use the property to store a value for its special needs.

Example
The following code assumes that the *OnClick* event handlers of more than one button point to the *TForm1.Button1Click* method. When a button is clicked, the procedure checks to see if the value of the *Tag* of the clicked button is 42. If so, the caption of that button is changed.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if (Sender as TButton).Tag = 42 then
    (Sender as TButton).Caption := 'A-Ha!';
end;
```

TAlign type — Controls

Declaration

```
TAlign = (alNone, alTop, alBottom, alLeft, alRight, alClient);
```

TAlign defines the possible values of the *Align* property.

TAlignment type — Classes

Declaration

```
TAlignment = (taLeftJustify, taRightJustify, taCenter);
```

TAlignment is the type of the *Alignment* property.

TApplication component — Forms

Each Delphi application automatically uses a *TApplication* component, which encapsulates your application. Delphi declares an *Application* variable of type *TApplication* that is an instance of your application.

When you execute your application, the application's *Run* method is called. The *Terminate* method ends application execution. The name of your application's executable file is the value of the *ExeName* property.

The main form of your application is the form specified as the value of the *MainForm* property. By default, Delphi uses *Form1* as the main form. If you want to make another form the main form, the form that is created first, use the Forms page of the Options | Project Options dialog box to do so.

When the user minimizes the application, the *Minimize* method is called. When the user restores the application to normal size, the *Restore* method is called. Your application can call these methods programmatically as well.

A minimized application appears as an icon on the Windows desktop. You can assign the icon of your choice to represent your application using the *Icon* property. The text that appears below the icon is the value of the *Title* property. If your application has a help file, specify its name as the value of the *HelpFile* property, and the Windows Help system can display help for your application. You can specify an icon, a help file, and the title of the application in the Options | Project dialog box on the Application page.

To display the help file for your application, call the *HelpContext* method.

You can specify how exceptions are handled for your application using the *HandleException* method, the *OnException* event, and the *ShowException* method.

To display a message to the user, use the *MessageBox* method.

TApplication has several events that let you specify how your application processes the occurrence of special events. The code you write in the *OnActivate* and *OnDeactivate* event handlers specifies what happens when your application becomes active and inactive. You specify how help hints appear in the *OnHint* and *OnShowHint* event handlers. The *OnIdle* event handler is used to determine what happens as your application becomes idle, and the *OnMessage* event handler is used to process Windows messages your application receives. Search help for "Handling Application Events" for more information about creating event handlers for application events.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all components.

Properties

▷ 🔑 Active	▷ 🔑 HelpFile	▷ Name
▷ ComponentCount	▷ 🔑 Hint	▷ Owner
▷ ComponentIndex	▷ 🔑 HintColor	▷ 🔑 ShowHint
▷ Components	▷ 🔑 HintPause	▷ Tag
▷ 🔑 ExeName	▷ 🔑 Icon	▷ 🔑 Terminated
▷ Handle	▷ 🔑 MainForm	▷ 🔑 Title

Methods

Create	🔑 HelpContext	RemoveComponent
CreateForm	🔑 HelpJump	Restore

Destroy	InsertComponent	🔑 RestoreTopMosts
FindComponent	🔑 MessageBox	Run
Free	🔑 Minimize	ShowException
HandleException	🔑 NormalizeTopMosts	🔑 Terminate
🔑 HelpCommand	🔑 ProcessMessages	

Events

▷ 🔑 OnActivate	▷ 🔑 OnHelp	▷ 🔑 OnMessage
▷ 🔑 OnDeactivate	▷ 🔑 OnHint	
▷ 🔑 OnException	▷ 🔑 OnIdle	

TAttachMode type — Outline

Declaration

```
TAttachMode = (oaAdd, oaAddChild, oaInsert);
```

Description

TAttachMode is the type of the *AttachMode* parameter of the *MoveTo* method. *TAttachMode* specifies the different ways which an outline item can attach to the new position when moved in an outline.

TAutoActivate type — ToCtrl

Declaration

```
TAutoActivate = (aaManual, aaGetFocus, aaDoubleClick);
```

TAutoActivate is the type of the *AutoActivate* property of the *TOLEContainer* component.

TBatchMode type — DBTables

Declaration

```
TBatchMode = (batAppend, batUpdate, batAppendUpdate, batDelete, batCopy);
```

The *TBatchMode* type is the set of values which are passed to the *BatchMove* method of a *TTable* or the *Mode* property of a *TBatchMove* component. *batAppend* appends all records. (The destination must not have any records with the key of the any of the records in the source.) *batUpdate* replaces all existing records with the new versions. (Each record in the source must have a record in the destination with the same key.) *batAppendUpdate* appends any records which do not already exist and replaces those which do. *batDelete* deletes the records in the source from the destination. (Each source record must have a key which is also found in the destination.) *batCopy* makes an exact duplicate of the source table.

TBatchMove component

TBatchMove enables you to perform operations on groups of records or entire tables. Set the *Source* property to specify a dataset (a *TQuery* or *TTable* component) corresponding to an existing source table. Set the *Destination* property to specify a *TTable* component corresponding to a database table. The destination table may or may not already exist.

Use the *Mode* property to specify the operations to perform. Set the *Mappings* property if the *Source* and *Destination* have different column names and you want to control how those fields are transferred.

Set *AbortOnProblem* to specifiy whether to abort the operation when a data type conversion error occurs. Set the *AbortOnKeyViol* property to specify whether to abort the operation when an integrity (key) violation occurs. Set *KeyViolTableName* and *ProblemTableName* to create Paradox tables to hold records that caused errors. Set the *ChangedTableName* property to save the replaced or deleted records from *Destination*.

Set the *Transliterate* property to specify whether to transliterate character data to the preferred character set for the destination table.

In addition to these properties and methods, this component also has the properties and methods that apply to all components.

Properties

🔑 AbortOnKeyViol	▷ 🔑 KeyViolTableName	▷ 🔑 ProblemCount
🔑 AbortOnProblem	🔑 Mappings	🔑 ProblemTableName
▷ 🔑 ChangedCount	🔑 Mode	🔑 RecordCount
🔑 ChangedTableName	🔑 MovedCount	🔑 Source
🔑 Destination	Name	🔑 Transliterate
▷ 🔑 KeyViolCount	▷ Owner	Tag

Methods

🔑 Execute

TBCDField component

A *TBCDField* represents a field of a record in a dataset. It is represented as a BCD value. Use *TBCDField* for a floating-point number with a fixed number of digits following the decimal point. The range depends on the number of digits after the decimal point, since the accuracy is 18 digits.

Set the *DisplayFormat* property to control the formatting of the field for display purposes, and the *EditFormat* property for editing purposes. Set the *Size* property to define the number of BCD digits following the decimal point. Use the *Value* property to access or change the current field value.

The *TBCDField* component has the properties, methods, and events of the *TField* component.

Properties

🔑 Alignment	🔑 DisplayLabel	🔑 MinValue
▷ 🔑 AsBoolean	▷ 🔑 DisplayName	Name
▷ 🔑 AsDateTime	▷ 🔑 DisplayText	▷ Owner
▷ 🔑 AsFloat	🔑 DisplayWidth	🔑 Precision
▷ 🔑 AsInteger	🔑 EditFormat	🔑 ReadOnly
▷ 🔑 AsString	🔑 EditMask	🔑 Required
🔑 Calculated	▷ 🔑 EditMaskPtr	🔑 Size
▷ 🔑 CanModify	🔑 FieldName	Tag
🔑 Currency	▷ 🔑 FieldNo	▷ 🔑 Text
▷ 🔑 DataSet	🔑 Index	▷ 🔑 Value
▷ 🔑 DataSize	▷ 🔑 IsIndexField	🔑 Visible
▷ 🔑 DataType	▷ 🔑 IsNull	
🔑 DisplayFormat	🔑 MaxValue	

Methods

🔑 Assign	🔑 FocusControl	🔑 SetData
🔑 AssignValue	🔑 GetData	
🔑 Clear	🔑 IsValidChar	

Events

🔑 OnChange	🔑 OnSetText	🔑 OnValidate
🔑 OnGetText		

TBevel component

ExtCtrls

The *TBevel* component lets you put beveled lines, boxes, or frames on the forms in your application.

You determine if the bevel appears as a box, frame, or line using the *Shape* property. The bevel can appear raised or lowered, depending on the value selected for the *Style* property.

To keep the bevel aligned within the form so that even if the user resizes the form, the bevel remains in the same relative position, set the *Align* property.

In addition to these properties and methods, this component also has the properties and methods that apply to all controls.

For more information, search for Bevel component in the online Help, and choose the topic Using the Bevel Component.

Properties

	Align		Height	🔑	Shape
▷	BoundsRect		Hint		ShowHint
▷	ComponentIndex		Left	🔑	Style
▷	Components		Name		Tag
▷	ControlCount	▷	Owner		Top
▷	Controls	▷	Parent		Visible
▷	Handle		ParentShowHint		Width

Methods

BeginDrag	Hide	SetBounds
BringToFront	Refresh	Show
ClientToScreen	Repaint	Update
Dragging	ScreenToClient	
EndDrag	SendToBack	

TBevelShape type StdCtrls

Declaration

```
TBevelShape = (bsBox, bsFrame, bsTopLine, bsBottomLine, bsLeftLine, bsRightLine);
```

The *TBevelShape* type defines the possible values of the *Shape* property of the *TBevel* component.

TBevelStyle type Card

Declaration

```
TBevelStyle = (bsLowered, bsRaised);
```

The *TBevelStyle* type defines the possible values of the *Style* property of the *TBevel* component.

TBevelWidth type

ExtCtrls

Declaration

```
TBevelWidth: 1..MaxInt;
```

Description

The *TBevelWidth* type defines the possible values of the *BevelWidth* property for a panel component (*TPanel*).

TBitBtn component

Buttons

A *TBitBtn* component is a push button control that can include a bitmap on its button face. You can choose from predefined bitmap buttons styles or specify your own bitmap for the button. Users use bitmap buttons as they would use a *TButton* component—to initiate actions.

Like buttons, bitmap buttons are frequently used within dialog boxes. A default bitmap button is the button whose *OnClick* event handler runs whenever the user presses the Enter key while using the dialog box. To make a bitmap button a default button, set the button's *Default* property to *True*.

A Cancel bitmap button is the button whose *OnClick* event handler runs whenever the user presses the Esc key while using the dialog box. To make a bitmap button a Cancel button, set the button's *Cancel* property to *True*.

You can have a bitmap button close a modal form without writing an event handler that includes code specifically to close the form. Set the button's *ModalResult* property to a value other than 0.

You can select from several predefined bitmap buttons. Specify the kind of bitmap button you want with the *Kind* property.

To create a customized bitmap button, use the *Glyph* property to specify the bitmap you want to appear on the button, and use the *Layout*, *Margin*, and *Spacing* properties to specify how to arrange the caption and bitmap on the button. You can use different images to represent the different states of the bitmap button. For example, you can use one image when the button is unselected, another when it is selected, and another when it is disabled. Use the *NumGlyphs* property to specify multiple images.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for the BitmapButton component in the online Help, and choose the topic Using the Bitmap Button Component.

Properties

▷	Align		HelpContext		PopupMenu
▷	BoundsRect		Hint		ShowHint
	Cancel	🔑	Kind	▷	Showing
	Caption	🔑	Layout	▷ 🔑	Spacing
▷	ComponentIndex		Left	▷ 🔑	Style
	Cursor	🔑	Margin		TabOrder
	Default	🔑	ModalResult		TabStop
	DragCursor		Name		Tag
	DragMode	🔑	NumGlyphs		Top
	Enabled	▷	Owner		Visible
	Font	▷	Parent		Width
🔑	Glyph		ParentFont		
	Height		ParentShowHint		

Methods

BeginDrag	GetTextBuf	SendToBack
BringToFront	GetTextLen	SetBounds
CanFocus	Hide	SetFocus
Click	Refresh	SetTextBuf
ClientToScreen	Repaint	Show
Dragging	ScaleBy	Update
EndDrag	ScreenToClient	
Focused	ScrollBy	

Events

OnClick	OnEnter	OnKeyUp
OnDragDrop	OnExit	OnMouseDown
OnDragOver	OnKeyDown	OnMouseMove
OnEndDrag	OnKeyPress	OnMouseUp

TBitBtnKind type — Buttons

Declaration

```
TBitBtnKind = (bkCustom, bkOK, bkCancel, bkHelp, bkYes, bkNo, bkClose, bkAbort, bkRetry,
bkIgnore, bkAll);
```

The *TBitBtnKind* type contains the values the *Kind* property of a *TBitBtn* bitmap button can assume.

TBitmap object

Graphics

A *TBitmap* object contains a bitmap graphic (.BMP file format). A *TBitmap* encapsulates a Windows HBITMAP and an HPALETTE and manages the realizing of the palette automatically.

The canvas of the *TBitmap* is a *TCanvas* object specified by the *Canvas* property. The palette of the *TBitmap* is specified by the *Palette* property.

The height and width in pixels of the bitmap are specified by the *Height* and *Width* properties, respectively.

If the *Monochrome* property is set to *False*, the bitmap is displayed in color. If *Monochrome* is set to *True*, the bitmap is displayed in monochrome.

To load a bitmap from a file, call the *LoadFromFile* method. To save a bitmap to a file, call *SaveToFile*.

To draw a bitmap on a canvas, call the *Draw* or *StretchDraw* methods of a *TCanvas* object, passing a *TBitmap* as a parameter.

When the bitmap is modified, an *OnChange* event occurs.

In addition to these properties, methods, and events, this object also has the methods that apply to all objects.

Properties

▷ 🔑 Canvas	▷ 🔑 Height	▷ 🔑 Width
▷ 🔑 Empty	▷ 🔑 Monochrome	
▷ 🔑 Handle	▷ 🔑 Palette	

Methods

Assign	Create	🔑 ReleaseHandle
ClassName	Destroy	🔑 ReleasePalette
ClassParent	Free	🔑 SaveToFile
ClassType	🔑 LoadFromFile	

Events

🔑 OnChange

TBlobField component

A *TBlobField* component represents a field of a record in a dataset. It is represented by a value consisting of an arbitrary set of bytes of indefinite size.

Use the *Assign* method to copy values from another field to a *TBlobField*. Use the *LoadFromFile* method to load a field's contents from a file. Use *LoadFromStream* method to load a field from a *Stream*. Use *SaveToFile* method to write a field's contents to a file. Use *SaveToStream* method to write a field's contents to a *Stream*.

The *TBlobField* component has the properties, methods, and events of the *TField* component.

Properties

☞ Alignment	▷ ☞ DataType	▷ ☞ IsIndexField
▷ ☞ AsBoolean	☞ DisplayLabel	▷ ☞ IsNull
▷ ☞ AsDateTime	▷ ☞ DisplayName	Name
▷ ☞ AsFloat	▷ ☞ DisplayText	▷ Owner
▷ ☞ AsInteger	☞ DisplayWidth	☞ ReadOnly
▷ ☞ AsString	☞ EditMask	☞ Required
☞ Calculated	▷ ☞ EditMaskPtr	☞ Size
▷ ☞ CanModify	☞ FieldName	Tag
▷ ☞ DataSet	▷ ☞ FieldNo	▷ ☞ Text
▷ ☞ DataSize	☞ Index	☞ Visible

Methods

☞ Assign	☞ GetData	☞ SaveToFile
☞ AssignValue	☞ IsValidChar	☞ SaveToStream
☞ Clear	☞ LoadFromFile	☞ SetData
☞ FocusControl	☞ LoadFromStream	

Events

☞ OnChange	☞ OnSetText	☞ OnValidate
☞ OnGetText		

TBlobStream object

The *TBlobStream* object provides a simple technique to access or modify a *TBlobField*, *TBytesField* or *TVarBytesField* by allowing you to "read" from or "write" to the field as if it were a file or stream.

Use the *Create* constructor to link the field to the BLOB stream. Call the *Read* or *Write* methods to access or change the contents of the field. Use *Seek* to position within the field. Call the *Truncate* method to delete all information in the field from the current position on.

In addition to these methods, this object also has the methods that apply to all objects.

Methods

ClassName	Destroy	Truncate
ClassParent	Free	Write
ClassType	Read	
Create	Seek	

TBlobStreamMode type DBTables

Declaration

```
TBlobStreamMode = (bmRead, bmWrite, bmReadWrite);
```

The *TBlobStreamMode* type is the set of values which are passed to the *Create* method of a *TBlobStream* object. Use *bmRead* to access an existing *TBlobField, TBytesField or TVarBytesField. Use bmWrite to clear the contents of the field and assign a new value. Use bmReadWrite to modify an existing value.*

TBookmark type DB

Declaration

```
TBookmark = Pointer;
```

The *TBookmark* type is the type of the *Bookmark* parameter you use to call the *GetBookmark*, *GotoBookmark*, and *FreeBookmark* methods of a dataset component.

TBooleanField component

A *TBooleanField* represents a field of a record in a dataset. A Boolean field is either *True* or *False*, but the display string in a data-aware control can be varied.

Set the *DisplayValues* property to control the formatting of the field for display purposes or input recognition. Use the *Value* property to access or change the current field value.

The *TBooleanField* component has the properties, methods, and events of the *TField* component.

Properties

🔑 Alignment	🔑 DisplayLabel	▷ 🔑 IsNull
▷ 🔑 AsBoolean	▷ 🔑 DisplayName	Name
▷ 🔑 AsDateTime	▷ 🔑 DisplayText	▷ Owner
▷ 🔑 AsFloat	🔑 DisplayValues	🔑 ReadOnly
▷ 🔑 AsInteger	🔑 DisplayWidth	🔑 Required
▷ 🔑 AsString	🔑 EditMask	▷ 🔑 Size

🔑 Calculated	▷ 🔑 EditMaskPtr	Tag
▷ 🔑 CanModify	🔑 FieldName	▷ 🔑 Text
▷ 🔑 DataSet	▷ 🔑 FieldNo	▷ 🔑 Value
▷ 🔑 DataSize	🔑 Index	🔑 Visible
▷ 🔑 DataType	▷ 🔑 IsIndexField	

Methods

🔑 Assign	🔑 FocusControl	🔑 SetData
🔑 AssignValue	🔑 GetData	
🔑 Clear	🔑 IsValidChar	

Events

🔑 OnChange	🔑 OnSetText	🔑 OnValidate
🔑 OnGetText		

TBorderIcons type **Forms**

Declaration

```
TBorderIcon = (biSystemMenu, biMinimize, biMaximize);
```

```
TBorderIcons = set of TBorderIcon;
```

The *TBorderIcons* type defines which icons appear in a form's title bar. *TBorderIcons* is the type of the *BorderIcons* property.

TBorderStyle type **Forms**

Declaration

```
TBorderStyle = bsNone..bsSingle;
```

TBorderStyle is the type of the *BorderStyle* property for controls.

The *BorderStyle* property for forms and windows uses the type *TFormBorderStyle*.

TBorderWidth type **ExtCtrls**

Declaration

```
TBorderWidth: 0..MaxInt;
```

Description

The *TBorderWidth* type defines the possible values for the *BorderWidth* property of a panel component (*TPanel*).

TBrush object **Graphics**

A *TBrush* object is used when filling solid shapes, such as rectangles and ellipses. The interior of the shape is filled with a color or pattern. *TBrush* encapsulates a Windows HBRUSH.

The color of the brush is specified by the *Color* property. The pattern is specified by the *Style* property. If a bitmap is specified by the *Bitmap* property, the pattern of the brush is defined by the bitmap rather than the *Style* property.

If the brush is modified, an *OnChange* event occurs.

In addition to these properties, methods, and events, this object also has the methods that apply to all objects.

Properties

▷ 🔑 Bitmap	▷ 🔑 Handle	▷ 🔑 Style
▷ 🔑 Color		

Methods

Assign	ClassType	Free
ClassName	Create	
ClassParent	Destroy	

Events

OnChange

TBrushStyle type **Graphics**

T

Declaration

```
TBrushStyle = (bsSolid, bsClear, bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross,
bsDiagCross);
```

The *TBrushStyle* type is used by the *Style* property to determine the pattern of a *TBrush* object.

TButton component

StdCtrls

A *TButton* component is a push button control. Users choose button controls to initiate actions. Buttons are most commonly used in dialog boxes.

A default button is the button whose *OnClick* event handler runs whenever the user presses the *Enter* key while using the dialog box. To make a button a default button, set the button's *Default* property to *True*.

A Cancel button is the button whose *OnClick* event handler runs whenever the user presses the *Esc* key while using the dialog box. To make a button a Cancel button, set the button's *Cancel* property to *True*.

You can have a button close a modal form without writing an event handler that includes code to specifically close the form. Set the button's *ModalResult* property to one of the values other than 0.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for Button component in the online Help, and choose the topic Using the Button Component.

Properties

▷	Align		Font		ParentShowHint
▷	BoundsRect		Height		PopupMenu
🔑	Cancel		HelpContext		ShowHint
	Caption		Hint	▷	Showing
▷	ComponentIndex		Left		TabOrder
	Cursor	🔑	ModalResult		TabStop
🔑	Default		Name		Tag
	DragCursor	▷	Owner		Top
	DragMode	▷	Parent		Visible
	Enabled		ParentFont		Width

Methods

BeginDrag	GetTextBuf	ScrollBy
BringToFront	GetTextLen	SendToBack
CanFocus	Hide	SetBounds
ClientToScreen	Refresh	SetFocus
Dragging	Repaint	SetTextBuf
EndDrag	ScaleBy	Show
Focused	ScreenToClient	Update

Events

OnClick	OnEnter	OnKeyUp
OnDragDrop	OnExit	OnMouseDown
OnDragOver	OnKeyDown	OnMouseMove
OnEndDrag	OnKeyPress	OnMouseUp

TButtonLayout type **Buttons**

Declaration

```
TButtonLayout = (blGlyphLeft, blGlyphRight, blGlyphTop, blGlyphBottom);
```

The *TButtonLayout* type defines the values the *Layout* property of a bitmap button (*TBitBtn*) or speed button (*TSpeedButton*) can assume.

TButtonSet type **MPlayer and DBCtrls**

For media players

Declaration

```
TButtonSet = set of TMPBtnType;
```

The *TButtonSet* type is a set of the buttons of the media player component. This set is used with the *ColoredButtons*, *EnabledButtons,* and *VisibleButtons* properties to determine how the buttons are displayed.

For database navigators

Declaration

```
TNavigateBtn = (nbFirst, nbPrior, nbNext, nbLast, nbInsert, nbDelete, nbEdit, nbPost,
nbCancel, nbRefresh);
```

```
TButtonSet = set of TNavigateBtn;
```

The *TButtonSet* type defines the possible values of the *VisibleButtons* property for the database navigator control.

TButtonStyle type **Buttons**

Declaration

```
TButtonStyle = (bbStandard, bbWin31, bbNew);
```

The *TButtonStyle* type contains the values the *Style* property of bitmap buttons (*TBitBtn*) and speed buttons (*TSpeedButton*) can assume.

TByteArray

SysUtils

Declaration

```
PByteArray = ^TByteArray;

TByteArray = array[0..32767] of Byte;
```

TByteArray declares a general array of type *Byte* that can be used in typecasting.

TBytesField component

A *TBytesField* represents a field of a record in a dataset. It is represented by a value consisting of an arbitrary set of bytes with indefinite size.

Use the *Assign* method to copy values from another field to a *TBytesField*.

The *TBytesField* component has the properties, methods, and events of the *TField* component.

Properties

🔑 Alignment	▷ 🔑 DataType	▷ 🔑 IsIndexField
▷ 🔑 AsBoolean	🔑 DisplayLabel	▷ 🔑 IsNull
▷ 🔑 AsDateTime	▷ 🔑 DisplayName	Name
▷ 🔑 AsFloat	▷ 🔑 DisplayText	▷ Owner
▷ 🔑 AsInteger	🔑 DisplayWidth	🔑 ReadOnly
▷ 🔑 AsString	🔑 EditMask	🔑 Required
🔑 Calculated	▷ 🔑 EditMaskPtr	🔑 Size
▷ 🔑 CanModify	🔑 FieldName	Tag
▷ 🔑 DataSet	▷ 🔑 FieldNo	▷ 🔑 Text
▷ 🔑 DataSize	🔑 Index	🔑 Visible

Methods

🔑 Assign	🔑 FocusControl	🔑 SetData
🔑 AssignValue	🔑 GetData	
🔑 Clear	🔑 IsValidChar	

Events

🔑 OnChange	🔑 OnSetText	🔑 OnValidate
🔑 OnGetText		

TCanvas object

Graphics

The *TCanvas* object is a drawing surface. It represents an area in which your application can draw. A *TCanvas* object encapsulates a Windows HDC display context.

The brush, pen, and font used to draw on the canvas are specified by the *Brush*, *Pen*, and *Font* properties.

The current position of the pen is specified by the *PenPos* property. To move the pen, call the *MoveTo* method.

To output text, call the *TextOut* method. To determine if text fits in a particular area, use *TextHeight* and *TextWidth*.

To draw a straight line, call *LineTo*. To draw a series of straight lines, call *PolyLine*. To draw curved lines, use the *Arc* or *Chord* methods.

You can draw a variety of shapes on a canvas. To draw a rectangle, call *Rectangle*. To draw a rectangle with rounded corners, call *RoundRect*. To draw an ellipse, call *Ellipse*. To draw a pie slice, call *Pie*. To draw a polygon defined by a series of points, call *Polygon*.

To fill a rectangular area with the pattern defined by *Brush*, call *FillRect*. To fill an entire area until boundaries are encountered, call *FloodFill*.

To output a graphic on a canvas, such as a bitmap or metafile, call *Draw*. To resize the graphic to a particular shape when drawn, call *StretchDraw*. To make a copy of t a rectangular area of the canvas, use *CopyRect*.

When a canvas is modified, an *OnChange* event occurs. Immediately prior to the modification of the canvas, an *OnChanging* event occurs.

In addition to these properties, methods, and events, this object also has the methods that apply to all objects.

Properties

▷ 🔑 Brush	▷ 🔑 Font	▷ 🔑 PenPos
▷ 🔑 ClipRect	▷ 🔑 Handle	▷ 🔑 Pixels
▷ 🔑 CopyMode	▷ 🔑 Pen	

Methods

🔑 Arc	▷ DrawFocusRect	🔑 PolyLine
🔑 BrushCopy	▷ Ellipse	🔑 Rectangle
🔑 Chord	▷ FillRect	🔑 RoundRect
ClassName	▷ FloodFill	🔑 StretchDraw
ClassParent	▷ FrameRect	🔑 TextHeight
ClassType	Free	🔑 TextOut
🔑 CopyRect	🔑 LineTo	🔑 TextRect
Create	🔑 MoveTo	🔑 TextWidth

Destroy	☞ Pie
☞ Draw	☞ Polygon

Events

☞ OnChange	☞ OnChanging

TCaption type — Controls

Declaration

```
TCaption = string[255];
```

The *TCaption* type defines the string type used for control captions. *TCaption* is the type of the *Caption* property and the *Text* property.

TChangeRange type — Outline

Declaration

```
TChangeRange = -1..1;
```

Description

TChangeRange specifies the valid values that can be passed to the *ChangeLevelBy* method. -1 moves an outline item up one level, and 1 moves an outline item down one level. 0 has no effect.

TCheckBox component — StdCtrls

A check box presents an option for the user; the user can check it to select the option, or uncheck it to deselect the option.

When the user checks or unchecks a check box, the value of the *Checked* property changes. The *OnClick* event also occurs. The text associated with the check box that identifies its purpose is the value of the *Caption* property.

If you want the user to be able to dim or gray the check box, set the *AllowGrayed* property to *True*. Whether the check box is checked, unchecked, or grayed is determined by the value of the *State* property. You can change value of *State* to change the check box's appearance.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for Check Box component in the online Help, and choose the topic Using the Check Box Component.

Properties

▷	Align		Font		PopupMenu
🔑	Alignment		Height		ShowHint
🔑	AllowGrayed		HelpContext	▷	Showing
	Caption		Hint	🔑	State
🔑	Checked		Left		TabOrder
	Color		Name		TabStop
	ComponentIndex	▷	Owner		Tag
	Ctl3D	▷	Parent		Top
	Cursor		ParentColor		Visible
	DragCursor		ParentCtl3D		Width
	DragMode		ParentFont		
	Enabled		ParentShowHint		

Methods

BeginDrag	Focused	ScrollBy
BringToFront	GetTextBuf	ScreenToClient
CanFocus	GetTextLen	SendToBack
ClientToScreen	Hide	SetBounds
Dragging	Refresh	SetTextBuf
EndDrag	Repaint	Show
FindComponent	ScaleBy	Update

Events

OnClick	OnEnter	OnKeyUp
OnDragDrop	OnExit	OnMouseDown
OnDragOver	OnKeyDown	OnMouseMove
OnEndDrag	OnKeyPress	OnMouseUp

See also

TDBCheckBox component, *TRadioButton* component

TCheckBoxState type — StdCtrls

Declaration

```
TCheckBoxState = (cbUnchecked, cbChecked, cbGrayed);
```

The *TCheckBoxState* type defines the different types of states the check box can assume. *TCheckBoxState* is the type of the *State* property of a *TCheckBox* check box control.

TClipboard object **Clipbrd**

The *TClipboard* object encapsulates the Windows Clipboard. Whenever you cut, copy, or paste text or graphics objects within a Delphi application or between a Delphi application and another Windows application, you are using the *TClipboard* object.

The *Clipbrd* unit declares the variable *Clipboard* as an instance of *TClipboard*. Use the *Clipboard* variable instead of creating your own instance of *TClipboard*.

You can place text in and retrieve text from the Clipboard using the *AsText* property. If you want to place pictures in and retrieve pictures from the Clipboard, use the *Assign* property. To add or retrieve a component object to the Clipboard, call the *GetComponent* and *SetComponent* methods.

The list of all the current formats on the Clipboard is found in the *Formats* property. The number of formats is the value of the *FormatCount* property. To find out if a specific format is on the Clipboard, call the *HasFormat* method.

Calling the *Clear* method clears the contents of the Clipboard.

Each time you add an item to the Clipboard, the previous contents are cleared automatically. To add multiple items, you should use the *Open* method to prevent the contents from being overwritten or being changed by another application. Call *Close* when you are finished adding items to the Clipboard.

You can add and other formats to the Clipboard using Windows handles with the *GetAsHandle* and *SetAsHandle* methods.

In addition to these properties and methods, this object also has the methods that apply to all objects.

Properties

▷ ☞ AsText	▷ ☞ FormatCount	▷ ☞ Formats

Methods

☞ Assign	Destroy	☞ Open
☞ Clear	GetAsHandle	SetAsHandle
☞ Close	☞ GetComponent	SetComponent
Create	☞ HasFormat	SetTextBuf

TCloseEvent type **Forms**

Declaration

```
TCloseAction = (caNone, caHide, caFree, caMinimize);

TCloseEvent = procedure(Sender: TObject; var Action: TCloseAction) of object;
```

The *TCloseEvent* type points to a method that handles the closing of a form. The value of the *Action* parameter determines if the form actually closes. The possible values of *Action* are defined by the *TCloseAction* type.

TCloseEvent is the type of the *OnClose* event.

TCloseQueryEvent type — Forms

Declaration

```
TCloseQueryEvent = procedure(Sender: TObject; var CanClose: Boolean) of object;
```

The *TCloseQueryEvent* type points to the method that determines whether a form can be closed. The value of the *CanClose* parameter determines if the form can close or not.

TCloseQueryEvent is the type of the *OnCloseQuery* event.

TColor type — Graphics

Declaration

```
TColor = -(COLOR_ENDCOLORS + 1)..$02FFFFFF;
```

The *TColor* type is used to specify the color of an object. It is used by the *Color* property of many components and the *BackgroundColor* of a tab set (*TTabSet*).

The *Graphics* unit contains definitions of useful constants for *TColor*. These constants map either directly to the closest matching color in the system palette (for example, *clBlue* maps to blue) or to the corresponding system screen element color defined in the Color section of the Windows Control panel (for example, *clBtnFace* maps to the system color for button faces).

The constants that map to the closest matching system colors are *clAqua, clBlack, clBlue, clDkGray, clFuchsia, clGray, clGreen, clLime, clLtGray, clMaroon, clNavy, clOlive, clPurple, clRed, clSilver, clTeal, clWhite,* and *clYellow.*

The constants that map to the system screen element colors are *clActiveBorder, clActiveCaption, clAppWorkSpace, clBackground, clBtnFace, clBtnHighlight, clBtnShadow, clBtnText, clCaptionText, clGrayText, clHighlight, clHighlightText, clInactiveBorder, clInactiveCaption, clInactiveCaptionText, clMenu, clMenuText, clScrollBar, clWindow, clWindowFrame,* and *clWindowText.*

If you specify *TColor* as a specific 4-byte hexadecimal number instead of using the constants defined in the *Graphics* unit, the low three bytes represent RGB color intensities for blue, green, and red, respectively. The value $00FF0000 represents full-intensity, pure blue, $0000FF00 is pure green, and $000000FF is pure red. $00000000 is black and $00FFFFFF is white.

If the highest-order byte is zero ($00), the color obtained is the closest matching color in the system palette. If the highest-order byte is one ($01), the color obtained is the closest matching color in the currently realized palette. If the highest-order byte is two ($02), the value is matched with the nearest color in the logical palette of the current device context.

To work with logical palettes, you must select the palette with the Windows API function *SelectPalette*. To realize a palette, you must use the Windows API function *RealizePalette*.

TColorDialog component — Dialogs

The *TColorDialog* component makes a Color dialog box available to your application. The purpose of the dialog box is to allow a user to select a color. When the user selects a color and chooses OK in the dialog box, the user's color selection is stored in the dialog box's *Color* property, which you can then use to process as you want.

Display the Color dialog box by calling the *Execute* method.

You can use the *Options* property to customize how the Color dialog box appears. For example, you can specify that a Help button be included in the dialog box.

In addition to these properties and methods, this component also has the properties and methods that apply to all components.

For more information, search for ColorDialog component in the online Help, and choose the topic Using the Color Dialog component.

Properties

🔑	Color		HelpContext	▷	Owner
▷	ComponentIndex	▷	Left		Tag
	Ctl3D		Name		Top
🔑	CustomColors		Options		

Methods

🔑	Execute

TColorDialogOptions type — Dialogs

Declaration

```
TColorDialogOption = (cdFullOpen, cdPreventFullOpen, cdShowHelp);
```

```
TColorDialogOptions = set of TColorDialogOption;
```

The *TColorDialogOptions* type declares the three options enumerated in the *TColorDialogOption* type as members of a set used by the *Options* property of the *TColorDialog* component.

TComboBox component

StdCtrls

A *TComboBox* component is a control that combines an edit box with a list, much like that of a list box. Users can either type text in the edit box or select an item from the list.

When users enter data into the combo box, either by typing text or selecting an item from the list, the value of the *Text* property changes. Your application can also change the *Text* property by displaying text for the user in the edit box of the combo box.

The list of items in the list is the value of the *Items* property. The *ItemIndex* property indicates which item in the list is selected.

You can add, delete, insert, and move items in the list using the *Add*, *Delete*, and *Insert* methods of the *Items* object, which is of type *TStrings*. For example, to add a string to the list, you could write this line of code:

```
ComboBox1.Items.Add('New item');
```

Sort the items in the list with the *Sorted* property.

At run time, you can select all the text in the edit box with the *SelectAll* method. To find out which text the user selected, or to replace selected text, use the *SelText* property. To select only part of the text or to find out what part of the text is selected, use the *SelStart* and *SelLength* properties.

You can change the style of the combo box or make it an owner-draw control by changing the value of the *Style* property.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for Combo Box component in the online Help, and choose the topic Using the Combo Box component.

Properties

▷	Align		Hint	▷ 🔑	SelLength
▷	BoundsRect	🔑	ItemHeight	▷ 🔑	SelStart
▷	Canvas	🔑	ItemIndex	▷ 🔑	SelTSelStartext
	Color	🔑	Items		ShowHint
▷	ComponentIndex		Left	▷	Showing
	Ctl3D	🔑	MaxLength	🔑	Sorted
	Cursor		Name	🔑	Style
	DragCursor	▷	Owner		TabOrder
	DragMode	▷	Parent		TabStop
🔑	DropDownCount		ParentColor		Tag
	Enabled		ParentCtl3D	🔑	Text
	Font		ParentFont		Top
	Height		ParentShowHint		Visible
	HelpContext		PopupMenu		Width

Methods

BeginDrag	GetTextBuf	ScrollBy
BringToFront	GetTextLen	☞ SelectAll
CanFocus	Hide	SendToBack
Clear	Invalidate	SetBounds
ClientToScreen	Refresh	SetTextBuf
Dragging	Repaint	Show
EndDrag	ScaleBy	Update
Focused	ScreenToClient	

Events

OnChange	☞ OnDrawItem	OnKeyDown
OnClick	OnDropDown	OnKeyPress
OnDblClick	OnEndDrag	OnKeyUp
OnDragDrop	OnEnter	☞ OnMeasureItem
OnDragOver	OnExit	

See also

Creating an owner-draw control

TComboBoxStyle type — StdCtrls

Declaration

```
TComboBoxStyle = (csDropDown, csSimple, csDropDownList, csOwnerDrawFixed,
csOwnerDrawVariable);
```

The *TComboBoxStyle* type defines the styles of combo boxes. *TComboBoxStyle* is the type of the combo box control's *Style* property.

TComponentName type — Classes

Declaration

```
TComponentName: string[63];
```

The *TComponentName* type is the type of the *Name* property for all components.

TControlScrollBar object — Forms

The *TControlScrollBar* object is used by the *HorzScrollBar* and *VertScrollBar* properties of a form or scroll box to display horizontal and vertical scroll bars users can use to scroll the form or scroll box.

The *HorzScrollBar* and *VertScrollBar* objects have nested properties that determine how these scroll bars behave. The *Range* property determines how far a user can scroll a form or scroll box. *Increment* determines how many positions the thumb tab on a scroll bar moves when the user clicks on the scroll bar arrows. You can set the position of the thumb tab with the *Position* property. If you don't want a scroll bar to appear, set the *Visible* property to *False*.

If you want to prevent controls from scrolling partially off screen so the user can't scroll them back into view, use the *Margin* property.

In addition to these properties, this object also has the methods that apply to all objects.

Properties

▷ Align	▷ Margin	▷ ☞ Range
▷ ComponentIndex	▷ Name	▷ ☞ ScrollPos
▷ ☞ Increment	▷ Owner	▷ Tag
▷ ☞ Kind	▷ ☞ Position	▷ ☞ Visible

TCopyMode type **Graphics**

Declaration

```
TCopyMode = Longint;
```

TCopyMode is the type of the *CopyMode* property of a *TCanvas* object.

TCurrencyField component

A *TCurrencyField* represents a field of a record in a dataset. It is represented as a binary value with a range from (positive or negative) 5.0 * 10^{-324} to 1.7 * 10^{308}. It has an accuracy of 15 to 16 digits. Use *TCurrencyField* for fields that hold currency values.

Set the *DisplayFormat* property to control the formatting of the field for display purposes, and the *EditFormat* property for editing purposes. Use the *Value* property to access or change the current field value.

The *TCurrencyField* component has the properties, methods, and events of the *TField* component.

Properties

☞ Alignment	☞ DisplayLabel	☞ MinValue
▷ ☞ AsBoolean	▷ ☞ DisplayName	Name
▷ ☞ AsDateTime	▷ ☞ DisplayText	▷ Owner
▷ ☞ AsFloat	☞ DisplayWidth	☞ Precision
▷ ☞ AsInteger	☞ EditFormat	☞ ReadOnly
▷ ☞ AsString	☞ EditMask	☞ Required

🔑 Calculated	▷ 🔑 EditMaskPtr	▷ 🔑 Size
▷ 🔑 CanModify	🔑 FieldName	Tag
🔑 Currency	▷ 🔑 FieldNo	▷ 🔑 Text
▷ 🔑 DataSet	🔑 Index	▷ 🔑 Value
▷ 🔑 DataSize	▷ 🔑 IsIndexField	🔑 Visible
▷ 🔑 DataType	▷ 🔑 IsNull	
🔑 DisplayFormat	🔑 MaxValue	

Methods

🔑 Assign	🔑 FocusControl	🔑 SetData
🔑 AssignValue	🔑 GetData	
🔑 Clear	🔑 IsValidChar	

Events

🔑 OnChange	🔑 OnSetText	🔑 OnValidate
🔑 OnGetText		

TCursor type — Controls

Declaration

```
TCursor = -32768..32767;
```

The *TCursor* type defines the different kinds of standard cursors a component can have. *TCursor* is the type of the *Cursor* property and the *DragCursor* property.

TCustomColors type — Dialogs

Declaration

```
TCustomColors = array[0..15] of Longint;
```

The *TCustomColors* type is an array that holds the color values for the custom colors the user can create using the Color dialog box (*TColorDialog* component).

TDatabase component — DB

The *TDatabase* component is not required for database access, but it provides additional control over factors that are important for client/server applications. If you do not create an explicit *TDatabase* component for a database, and an application opens a table in the database, then Delphi will create a temporary (virtual) *TDatabase* component.

DatabaseName is the name of the database connection that can be used by dataset components. In other words, this is the name of the local alias defined by the component that will show up in the *DatabaseName* drop-down list of dataset components.

AliasName is the name of an existing BDE alias defined with the BDE Configuration Utility. This is where the *TDatabase* component gets its default parameter settings. This property will be cleared if *DriverName* is set. The *Params* property holds the connection parameters for the alias.

DriverName is the name of a BDE driver, such as STANDARD (for dBASE and Paradox), ORACLE, SYBASE, INFORMIX or INTERBASE. This property will be cleared if *AliasName* is set, because an *AliasName* specifies a driver type.

The *DataSets* property of *TDatabase* is an array of references to the active datasets in the *TDatabase*. The *DatasetCount* property is an integer that specifies the number of active datasets.

Set the *Connected* property to open or close the database. Set *KeepConnection* to *True* to avoid having to log in to the server each time the database is opened.

Set *LoginPrompt* to *True* to always prompt for user name and password when logging in to the database server.

The *TDatabase* component controls server transactions. Call *StartTransaction* to begin a transaction, *RollBack* to cancel it, or *Commit* to commit the changes. The *TransIsolation* property specifies the transaction isolation level to request on the server.

In addition to these properties and methods, this component also has the properties and methods that apply to all components.

Properties

🔑 AliasName	▷ 🔑 Handle	▷ Owner
🔑 Connected	▷ 🔑 IsSQLBased	🔑 Params
🔑 DatabaseName	🔑 KeepConnection	Tag
▷ 🔑 DatasetCount	▷ 🔑 Locale	▷ 🔑 Temporary
▷ 🔑 Datasets	🔑 LoginPrompt	TransIsolation
🔑 DriverName	Name	

Methods

🔑 Close	🔑 Commit	🔑 Rollback
🔑 CloseDatasets	🔑 Open	🔑 StartTransaction

Events

🔑 OnLogin

TDataChangeEvent type DB

Declaration

```
TDataChangeEvent = procedure(Sender: TObject; Field: TField) of object;
```

The *TDataChangeEvent* points to a method that handles the changing of data in a data source component (*TDataSource*). The *Field* parameter is the field in which the data is changing. It is used by the *OnDataChange* event of the data source.

TDataMode type DDEMan

Declaration

```
TDataMode = (ddeAutomatic, ddeManual);
```

The *TDataMode* type contains the types of connect modes used when initiating a DDE conversation. Specify the connect mode in the *ConnectMode* property.

TDataSetNotifyEvent type DB

Declaration

```
TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;
```

The *TDataSetNotifyEvent* type points to a method that notifies a dataset component that an event has occurred. It is used by all the events of the tables, queries, and stored procedures (*TTable*, *TQuery*, and *TStoredProc* components).

TDataSetState type DB

Declaration

```
TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert, dsSetKey, dsCalcFields);
```

The *TDataSetState* type is the set of values of the *State* property of a dataset component.

TDataSource component DB

TDataSource is the interface between a dataset component and data-aware controls on forms. *TDataSource* attaches to a dataset through the *Dataset* property. Data-aware controls, such as database edit boxes and data grids, attach to a *TDataSource* through their *DataSource* properties. Usually there is only one data source for each dataset component, but there can be as many data source components connected to a dataset as you need.

The *Dataset* property identifies the dataset from which the data is obtained. Set the *AutoEdit* property to *False* to prevent the dataset from going into edit mode automatically when the value of an attached data-aware control is modified (you can still call the *Edit* method to permit modifications). Set the *Enabled* property to *False* to clear and disable the data-aware controls. Check the current status of the dataset with the *State* property. To monitor changes to both the dataset and attached data-aware controls, assign a method to the *OnDataChange* event. To monitor changes in the dataset's state, assign a method to the *OnStateChange* event. To update the dataset prior to a post, assign a method to the *OnUpdateData* event.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all components.

Properties

🔑 AutoEdit	Name	Tag
🔑 Dataset	▷ Owner	
🔑 Enabled	▷ 🔑 State	

Methods

🔑 *Edit*

Events

🔑 OnDataChange	🔑 OnStateChange	🔑 OnUpdateData

TDateField component

A *TDateField* represents a field of a record in dataset. It represents a value consisting of a date.

Set the *DisplayFormat* property to control the formatting of the field for display purposes, and the *EditFormat* property for editing purposes. Use the *Value* property to access or change the current field value.

The *TDateField* component has the properties, methods, and events of the *TField* component.

Properties

☞ Alignment	▷ ☞ DataType	▷ ☞ IsIndexField
▷ ☞ AsBoolean	☞ DisplayLabel	▷ ☞ IsNull
▷ ☞ AsDateTime	▷ ☞ DisplayName	Name
▷ ☞ AsFloat	▷ ☞ DisplayText	▷ Owner
▷ ☞ AsInteger	☞ DisplayWidth	☞ ReadOnly
▷ ☞ AsString	☞ EditMask	☞ Required
☞ Calculated	▷ ☞ EditMaskPtr	▷ ☞ Size
▷ ☞ CanModify	☞ FieldName	Tag
▷ ☞ DataSet	▷ ☞ FieldNo	▷ ☞ Text
▷ ☞ DataSize	☞ Index	☞ Visible

Methods

☞ Assign	☞ FocusControl	☞ SetData
☞ AssignValue	☞ GetData	
☞ Clear	☞ IsValidChar	

Events

☞ OnChange	☞ OnSetText	☞ OnValidate
☞ OnGetText		

TDateTime type **System**

Declaration

```
TDateTime: Float;
```

TDateTime is the type used by date and time routines to hold date and time values.

Delphi stores dates in the *TDateTime* type as the number of days that have passed since 1/1/0001. The resulting value is an integer. Time is stored as the floating-point part of the *TDateTime*. The floating-point part represents the fractional part of the day.

TDateTimeField component

A *TDateTimeField* component represents a field of a record in a dataset. It represents a value consisting of a date and time.

Set the *DisplayFormat* property to control the formatting of the field for display purposes, and the *EditFormat* property for editing purposes. Use the *Value* property to access or change the current field value.

The *TDateTimeField* component has the properties, methods, and events of the *TField* component.

Properties

🔑 Alignment	🔑 DisplayFormat	▷ 🔑 IsNull
▷ 🔑 AsBoolean	🔑 DisplayLabel	Name
▷ 🔑 AsDateTime	▷ 🔑 DisplayName	▷ Owner
▷ 🔑 AsFloat	▷ 🔑 DisplayText	🔑 ReadOnly
▷ 🔑 AsInteger	🔑 DisplayWidth	🔑 Required
▷ 🔑 AsString	🔑 EditMask	▷ 🔑 Size
🔑 Calculated	▷ 🔑 EditMaskPtr	Tag
▷ 🔑 CanModify	🔑 FieldName	▷ 🔑 Text
▷ 🔑 DataSet	▷ 🔑 FieldNo	▷ 🔑 Value
▷ 🔑 DataSize	🔑 Index	🔑 Visible
▷ 🔑 DataType	▷ 🔑 IsIndexField	

Methods

🔑 Assign	🔑 FocusControl	🔑 SetData
🔑 AssignValue	🔑 GetData	
🔑 Clear	🔑 IsValidChar	

Events

🔑 OnChange	🔑 OnSetText	🔑 OnValidate
🔑 OnGetText		

TDBCheckBox component **DBCtrls**

A check box presents an option to the user; the user can check it to select the option, or uncheck it to deselect the option. A database check box (*TDBCheckBox*) is much like an ordinary check box (*TCheckBox*), except that it is aware of the data in a particular field of a dataset.

You can link a database check box with a dataset by specifying the data source component (*TDataSource*) that identifies the dataset as the value of the check box's *DataSource* property. Specify the field in the dataset you want to access as the value of the check box's *DataField* property.

If the contents of a field in the current record of the dataset equals the string of the *ValueChecked* property, the database check box is checked. If the contents matches the string specified as the value of the *ValueUnchecked* property, the check box is unchecked.

When the user checks or unchecks a database check box, the string specified as the value of the *ValueChecked* or *ValueUnchecked* property becomes the value of the field in the dataset, as long as the value of the *ReadOnly* property is *False* and the dataset is in edit mode. If you want the user to be able to view the data in the field but not modify it, set *ReadOnly* to *True*.

If your application doesn't require the data-aware capabilities of *TDBCheckBox*, use the check box (*TCheckBox*) component instead to conserve system resources.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for DBCheckBox component in the online Help, and choose the Using the DBCheck Box Component.

Properties

▷	Align		Enabled		PopupMenu
🔑	Alignment		Font		ReadOnly
🔑	AllowGrayed		Height		ShowHint
	Caption		HelpContext	▷	Showing
🔑	Checked		Hint	🔑	State
	Color		Left		TabOrder
▷	ComponentIndex		Name		TabStop
	Ctl3D	▷	Owner		Tag
	Cursor	▷	Parent		Top
🔑	DataField		ParentColor	🔑	ValueChecked
🔑	DataSource		ParentCtl3D	🔑	ValueUnchecked
	DragCursor		ParentFont		Visible
	DragMode		ParentShowHint		Width

Methods

BeginDrag	GetTextBuf	ScreenToClient
BringToFront	GetTextLen	ScrollBy
CanFocus	Hide	SendToBack
ClientToScreen	Invalidate	SetBounds
Dragging	Refresh	SetTextBuf
EndDrag	Repaint	Show
Focused	ScaleBy	Update

Events

OnClick	OnEnter	OnKeyUp
OnDragDrop	OnExit	OnMouseDown
OnDragOver	OnKeyDown	OnMouseMove
OnEndDrag	OnKeyPress	OnMouseUp

TDBComboBox component DBCtrls

A *TDBComboBox* component is a data-aware combo box control. It allows the user to change the value of the field of the current record in a dataset either by selecting an item from a list or by typing in the edit box part of the control. The selected item or entered text becomes the new value of the field if the database combo box's *ReadOnly* property is *False*.

How a database combo box appears and behaves depends on the value of its *Style* property.

You can link the database combo box with a dataset by specifying the data source component (*TDataSource*) that identifies the dataset as the value of the memo's *DataSource* property. Specify the field in the dataset you want to access as the value of the *DataField* property.

You specify the values the user can choose from in the combo box with the *Items* property. For example, if you want the user to choose from five different values in the combo box list, specify five strings as the value of *Items*. Just as with an ordinary combo box, you can add, delete, and insert items to it using the *Add*, *Delete*, and *Insert* methods of the *Items* object, which is of type *TStrings*. For example, to add a string to a database combo box, you could write this line of code:

```
DBListBox1.Items.Add('New item');
```

The *ItemIndex* property indicates which item in the database combo box is selected.

Sort the items in the list with the *Sorted* property.

At run time, you can select all the text in the edit box of the database combo box with the *SelectAll* method. To find out which text the user selected, or to replace selected text, use the *SelText* property. To select only part of the text or to find out what part of the text is selected, use the *SelStart* and *SelLength* properties.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for DBComboBox component in the online Help, and choose the topic Using the DBCombo Box Component.

Properties

▷	Align		Height	🔑	ReadOnly
▷	BoundsRect		HelpContext	▷ 🔑	SelLength
	Color		Hint	▷ 🔑	SelStart
▷	ComponentIndex	🔑	ItemHeight	▷ 🔑	SelText
	Ctl3D	▷ 🔑	ItemIndex		ShowHint
	Cursor	🔑	Items	▷	Showing
🔑	DataField		Left	🔑	Sorted
🔑	DataSource		Name	🔑	Style
	DragCursor	▷	Owner		TabOrder
	DragMode	▷	Parent		TabStop
🔑	DropDownCount		ParentColor		Tag
	Enabled		ParentCtl3D		Text
▷ 🔑	Fields		ParentFont		Top

T

	Font		ParentShowHint		Visible
▷	Handle		PopupMenu		Width

Methods

	BeginDrag		Focused		ScreenToClient
	BringToFront		GetTextBuf		ScrollBy
	CanFocus		GetTextLen	🔑	SelectAll
🔑	Clear		Hide		SendToBack
	ClientToScreen		Invalidate		SetBounds
🔑	CopyToClipboard	🔑	PasteFromClipboard		SetFocus
🔑	CutToClipboard		Refresh		SetTextBuf
	Dragging		Repaint		Show
	EndDrag		ScaleBy		Update

Events

	OnChange	🔑	OnDrawItem		OnKeyDown
	OnClick		OnDropDown		OnKeyPress
	OnDblClick		OnEndDrag		OnKeyUp
	OnDragDrop		OnEnter		OnMeasureItem
	OnDragOver		OnExit		

TDBEdit component

DBEdit

A *TDBEdit* component is a data-aware edit box with all the capabilities of an ordinary edit box (a *TEdit* component).

Unlike an ordinary edit box, you can use the database edit box to enter data into a field, or to simply display data from a field in a dataset. Link the database edit box with a dataset by specifying the data source component (*TDataSource*) that identifies the dataset as the value of the edit box's *DataSource* property. Specify the field in the dataset you want to access as the value of the *DataField* property.

Your application can tell if the text displayed in the edit box changed by checking the value of the *Modified* property. To limit the number of characters users can enter into the edit box, use the *MaxLength* property.

If you want to prevent the user from modifying the contents of the field linked to the edit box, set the *ReadOnly* property to *True*. .

You can choose to have the text in an edit box automatically selected whenever it becomes the active control with the *AutoSelect* property. At run time, you can select all the text in the edit box with the *SelectAll* method. To find out which text in the edit box the user has selected or to replace selected text, use the *SelText* property. To clear selected text, call the *ClearSelection* method. To select only part of the text or to find out what part of the text is selected, use the *SelStart* and *SelLength* properties.

You can cut, copy, and paste text in an edit box using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods.

Your application can use an edit box that displays a specified character rather than the actual character typed into it. If the edit box is used to enter a password, onlookers won't be able to read the typed text. Specify the special character with the *PasswordChar* property.

If you want the edit box to automatically resize to accommodate a change in font size, use the *AutoSize* property.

If your application doesn't require the data-aware capabilities of *TDBEdit*, use the *TEdit* component instead to conserve system resources.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for DBEdit component in the online Help, and choose the topic Using the DBEdit Component.

Properties

▷	Align	▷ 🔑	Fields	🔑	PasswordChar
▷ 🔑	AutoSelect		Font		PopupMenu
🔑	AutoSize		Height	🔑	ReadOnly
🔑	BorderStyle		HelpContext	▷ 🔑	SelLength
▷	BoundsRect		Hint	▷ 🔑	SelStart
🔑	CharCase	🔑	IsMasked	▷ 🔑	SelText
	Color		Left		ShowHint
▷	ComponentIndex	🔑	MaxLength	▷	Showing
	Ctl3D	▷ 🔑	Modified		TabOrder
	Cursor		Name		TabStop
🔑	DataField	▷	Owner		Tag
🔑	DataSource	▷	Parent	▷ 🔑	Text
	DragCursor		ParentColor		Top
	DragMode		ParentCtl3D		Visible
▷ 🔑	EditText		ParentFont		Width
	Enabled		ParentShowHint		

Methods

	BeginDrag	🔑	GetSelTextBuf	🔑	SelectAll
	BringToFront		GetTextBuf		SendToBack
	CanFocus		GetTextLen		SetBounds
🔑	Clear		Hide		SetFocus
🔑	ClearSelection		Invalidate	🔑	SetSelTextBuf
	ClientToScreen	🔑	PasteFromClipboard		SetTextBuf
🔑	CopyToClipboard		Refresh		Show
🔑	CutToClipboard		Repaint		Update

Dragging	ScaleBy	🔑 ValidateEdit
EndDrag	ScreenToClient	
Focused	ScrollBy	

Events

OnChange	OnEndDrag	OnKeyUp
OnClick	OnEnter	OnMouseDown
OnDblClick	OnExit	OnMouseMove
OnDragDrop	OnKeyDown	OnMouseUp
OnDragOver	OnKeyPress	

TDBGrid component **DBGrids**

The *TDBGrid* component can access the data in a database table or query and display it in a grid. Your application can use the data grid to insert, delete, or edit data in the database, or simply to display it.

The most convenient way to move through data in a data grid and to insert, delete, and edit data is to use the database navigator (*TDBNavigator*) with the data grid.

The *Fields* property is an array of all the fields in the dataset displayed in the data grid. To determine which field is the currently selected field, use the *SelectedField* property. Use the *FieldCount* property to find out how many fields are in the dataset displayed in the data grid.

You can change the appearance and behavior of a data grid by changing the value of the *Options* property. For example, you can choose to allow the user to use the **Tab** key to move to a new column, or you can decide to display grid lines between columns, but not between rows.

If you want the user to be able only to view the data and not to edit it, set the *ReadOnly* property to *True*. If you want the user to be able to edit the data, set *ReadOnly* to *False*. Also, the dataset must be in Edit state, and the *ReadOnly* property of the data must be *False*. The user can cancel an edit by pressing **Esc**.

Users don't really insert or edit the data in a field using the data grid until they move to a different record or close the application. Your application can also post edits using code within event handlers such as *OnColExit* or *OnColEnter*.

The value of the *TitleFont* property determines which font is used to display the column headings.

To customize the order the fields appear in the grid, use the Fields editor. You can find information about it in the online Help; search for Fields Editor.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for DBGrid component in the online Help, and choose the topic Using the DBGrid Component.

Properties

Align	▷ 🔑 EditorMode	PopupMenu
🔑 BorderStyle	Enabled	🔑 ReadOnly
▷ BoundsRect	▷ 🔑 FieldCount	▷ 🔑 SelectedField
▷ Brush	▷ 🔑 Fields	▷ 🔑 SelectedIndex
▷ Canvas	🔑 FixedColor	▷ Showing
▷ ClientHeight	Font	TabOrder
▷ ClientOrigin	Height	TabStop
▷ ClientRect	HelpContext	Tag
▷ ClientWidth	Hint	🔑 TitleFont
Color	Left	Top
▷ ComponentIndex	Name	🔑 TopRow
Ctl3D	Options	Visible
Cursor	Owner	Width
🔑 DataSource	Parent	
🔑 DefaultDrawing	ParentColor	
DragCursor	ParentCtl3D	
DragMode	ParentFont	

Methods

BeginDrag	GetTextBuf	ScrollBy
BringToFront	GetTextLen	SendToBack
CanFocus	Hide	SetBounds
ClientToScreen	Invalidate	SetFocus
Dragging	Refresh	SetTextBuf
EndDrag	Repaint	Show
FindComponent	ScaleBy	Update
Focused	ScreenToClient	

Events

OnColEnter	OnDragOver	OnExit
OnColExit	🔑 OnDrawDataCell	OnKeyDown
OnDblClick	OnEndDrag	OnKeyPress
OnDragDrop	OnEnter	OnKeyUp

TDBGridOptions type — DBGrids

Declaration

```
TDBGridOption = (dgEditing, dgAlwaysShowEditor, dgTitles, dgIndicator,
dgColumnResize,dgColLines, dgRowLines, dgTabs, dgRowSelect, dgAlwaysShowSelection,
dgConfirmDelete, dgCancelOnExit);
```

```
TDBGridOptions = set of TDBGridOption;
```

The *TDBGridOptions* type is a set that defines the possible values of the *Options* property of the data grid (*TDBGrid*).

TDBImage component DBCtrls

The *TDBImage* component displays a graphic image from a BLOB (binary large object) stored in a field of the current record of a dataset. You can also modify the image if the *ReadOnly* property is set to *False*.

You can link the database image with a dataset by specifying the data source component (*TDataSource*) that identifies the dataset as the value of the image's *DataSource* property. Specify the field in the dataset you want to access as the value of the image's *DataField* property.

You can control when the image appears in the database control with the *AutoDisplay* property.

You can change the size at which the BLOB is displayed by using the *Stretch* property.

You can cut, copy, and paste images in the database image control. While your application is running and the database image control has the focus, use the Windows cut, copy, and paste keys (*Ctrl+X*, *Ctrl+C*, and *Ctrl+V*). If you change your mind, you can return to the original state of the database image control by pressing *Esc* before moving to another record.

If your application doesn't require the data-aware capabilities of *TDBImage*, use a database image control (*TImage*) instead to conserve system resources.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for DBImage Component in the online Help, and choose the topic Using the DBImage Component.

Properties

	Align	🔑	Fields		ParentShowHint
🔑	AutoDisplay		Font		PopupMenu
🔑	BorderStyle	▷	Handle	🔑	ReadOnly
🔑	Center		Height		ShowHint
	Color		HelpContext	🔑	Stretch
▷	ComponentIndex		Hint		TabOrder
	Ctl3D		Left		TabStop
	Cursor		Name		Tag
🔑	DataField		Owner		Top
🔑	DataSource	▷	Parent		Visible
	DragCursor	▷	ParentColor		Width

DragMode	ParentCtl3D
Enabled	ParentFont

Methods

BeginDrag	Focused	ScreenToClient
BringToFront	Hide	SendToBack
ClientToScreen	Invalidate	SetBounds
🔑 CopyToClipboard	🔑 LoadPicture	Show
🔑 CutToClipboard	PasteFromClipboard	Update
Dragging	Refresh	
EndDrag	Repaint	

Events

OnClick	OnEnter	OnMouseDown
OnDblClick	OnExit	OnMouseMove
OnDragDrop	OnKeyDown	OnMouseUp
OnDragOver	OnKeyPress	
OnEndDrag	OnKeyUp	

TDBListBox component **DBCtrls**

The *TDBListBox* component is a data-aware list box. It allows the user to change the value of the field of the current record in a dataset by selecting an item from a list. The selected item becomes the new value of the field.

Link the database list box with a dataset by specifying the data source component (*TDataSource*) that identifies the dataset as the value of the memo's *DataSource* property. Specify the field in the dataset you want to access as the value of the *DataField* property.

You specify the values the user can choose from in the list box with the *Items* property. For example, if you want the user to choose from five different values in the list box, specify five strings as the value of *Items*. Just as with an ordinary list box, you can add, delete, and insert items in the list box using the *Add*, *Delete*, and *Insert* methods of the *Items* object, which is of type *TStrings*. For example, to add a string to a database list box, you could write this line of code:

```
DBListBox1.Items.Add('New item');
```

The *ItemIndex* property indicates which item in the list box is selected. If you want to prevent the user from being able to select an item in the list box, set the *ReadOnly* property to *False*.

If your application doesn't require the data-aware capabilities of *TDBListBox*, use a list box (*TListBox*) instead to conserve system resources.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for DBListBox component in the online Help, and choose the topic Using the DBListBox Component.

Properties

	Align		Font		ParentFont
🔑	BorderStyle	▷	Handle		PopupMenu
▷	BoundsRect		Height	🔑	ReadOnly
▷	Brush		HelpContext	▷ 🔑	SelCount
▷ 🔑	Canvas		Hint	▷ 🔑	Selected
	Color	▷ 🔑	ItemIndex	▷	Showing
	ComponentIndex	🔑	IntegralHeight	🔑	Sorted
	Ctl3D	🔑	ItemHeight	🔑	Style
	Cursor	🔑	Items		TabOrder
🔑	DataField		Left		TabStop
🔑	DataSource		Name		Tag
	DragCursor	▷	Owner		Top
	DragMode	▷	Parent	▷ 🔑	TopIndex
	Enabled		ParentColor		Visible
▷ 🔑	Fields		ParentCtl3D		Width

Methods

	BeginDrag		GetTextLen		ScrollBy
	BringToFront		Hide		SendToBack
	CanFocus	🔑	ItemAtPos		SetBounds
	Clear	🔑	ItemRect		SetFocus
	ClientToScreen		Invalidate		SetTextBuf
	Dragging		Refresh		Show
	EndDrag		Repaint		Update
	Focused		ScaleBy		
	GetTextBuf		ScreenToClient		

Events

	OnClick		OnEndDrag		OnKeyUp
	OnDblClick		OnEnter	🔑	OnMeasureItem
	OnDragDrop		OnExit		OnMouseDown
	OnDragOver		OnKeyDown		OnMouseMove
🔑	OnDrawItem		OnKeyPress		OnMouseUp

TDBLookupCombo component **DBLookup**

A *TDBLookupCombo* component is a data-aware combo box that “looks up” a value in a lookup table.

For example, imagine that *DataSource1* identifies the table called *Customers*, and *DataSource2* identifies the table called *Orders*. The *Orders* table contains a *CustNo* field which has a number that identifies the customer who placed the order. When the user moves through the records of the *Orders* table, you want the database lookup combo box to display the name of the customer, and you want the drop-down list of the combo box to display all the customer names. You can do this, because *Customers* also contains a field that identifies the customer by number (*CustNo*), as well as the customer’s name.

To have the combo box look up the customer name, set the *DataSource* property value of the combo box to *DataSource2*, which refers to the *Orders* table. Set the *DataField* property value to *CustNo*. The *LookupSource* is the data source that refers to the table the combo box uses to look up the name of the customer—in this case, *DataSource1*—because the *Customers* table contains the name of the customer.

Set the *LookupField* property to *CustNo*. *LookupField* links the two tables on the value that identifies the customer by number. In this example, both the *DataField* value and the *LookupField* value have the same field name, but this isn’t required. If the *Active* property of both tables is *True*, the database combo box displays the value of the *CustNo* field. You want to see the customer’s name—not the customer number—so set the *LookupDisplay* property to *Name*, the field that contains the full name of the customer. Now as you move through the records in the *Orders* table, the name of the customer who placed the order appears in the database lookup combo box.

You can choose to display multiple fields in the drop-down list of the combo box by entering a list of fields to display as the value of the *LookupDisplay* property. To display the resulting columns the way you want, use the *Options* property.

If the *ReadOnly* property is *False*, the user can select a displayed value in the database lookup combo box and the corresponding value in current record of the primary dataset updates with a new value. Using the *Customers* and *Orders* example, when the user selects a customer name in the lookup table using the database lookup combo box, the value of the *CustNo* field in the primary dataset updates accordingly.

The *Style* property determines whether the user can edit a selected item in the combo box or enter a new value and therefore change the value in the lookup table, or simply select items without being able to edit them.

The *DropDownCount* and *DropDownWidth* properties determine how long and how wide the drop-down list of the combo box is.

The *Value* property is the string the combo box uses to identify which record in the lookup table to display; it is the contents of the *DataField* for the current record. The *DisplayValue* is the actual displayed string in the combo box.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

Properties

▷	Align		Height		PopupMenu
🔑	AutoSelect		HelpContext	🔑	ReadOnly
▷	BoundsRect		Hint	▷ 🔑	SelLength
	Color		Left	▷ 🔑	SelStart
▷	ComponentIndex	🔑	LookupDisplay	▷ 🔑	SelText
	Ctl3D	🔑	LookupField		ShowHint
	Cursor	🔑	LookupSource	▷	Showing
🔑	DataField	🔑	MaxLength	🔑	Style
🔑	DataSource		Name		TabOrder
▷ 🔑	DisplayValue		Options		TabStop
	DragCursor	▷	Owner		Tag
	DragMode	▷	Parent	🔑	Text
🔑	DropDownCount		ParentColor		Top
🔑	DropDownWidth		ParentCtl3D	▷ 🔑	Value
	Enabled		ParentFont		Visible
	Font		ParentShowHint		Width

Methods

	BeginDrag		Focused		ScrollBy
	BringToFront		GetTextBuf		SelectAll
	CanFocus		GetTextLen		SendToBack
	Clear		Hide		SetBounds
	ClientToScreen		Invalidate		SetTextBuf
	CloseUp		Refresh		Show
	Dragging		Repaint		Update
🔑	DropDown		ScaleBy		
	EndDrag		ScreenToClient		

Events

	OnChange	🔑	OnDropDown		OnKeyPress
	OnClick		OnEndDrag		OnKeyUp
	OnDblClick		OnEnter		OnMouseDown
	OnDragDrop		OnExit		OnMouseMove
	OnDragOver		OnKeyDown		OnMouseUp

TDBLookupComboStyle type

DBLookup

Declaration

```
TDBLookupComboStyle = (csDropDown, csDropDownList);
```

The *TDBLookupComboStyle* determines the kind of combo box. It is the type of the *Style* property for a database lookup combo box (*TDBLookupCombo*).

TDBLookupList component **DBLookup**

A *TDBLookupList* component is a data-aware list box that "looks up" a value in a lookup table.

For example, imagine that *DataSource1* identifies the table called *Books*, and *DataSource2* identifies the table called *BookOrders*. The *BookOrders* table contains a *Volume* field that use a number to identify the book the customer ordered. When the user moves through the records of the *BookOrders* table, you want the database lookup list box to display the titles of the books. You can do this, because *Books* also contains a field that identifies the book by number (*Volume*), as well as the title of the book.

For the database lookup list box to look up the title of the book, set the *DataSource* property value of the list box to *DataSource2*, which refers to the *BookOrders* table. Set the *DataField* property value to *Volume*. The *LookupSource* is the data source that refers to the table the combo box uses to look up the title of the book—in this case, *DataSource1*—because the *Books* table contains the book's title.

Set the *LookupField* property to *Volume*. *LookupField* links the two tables on the value that identifies the book by number. In this example, both the *DataField* value and the *LookupField* value have the same field name, but this isn't required. If the *Active* property of both tables is *True*, the database list box now displays the value of the *Volume* field. You want to see the title of the book—not the volume number—so set the *LookupDisplay* property to *Title*, the field that contains the title of the book. Now as you move through the records in the *BookOrders* table, the title of the ordered book appears in the database lookup list box.

You can choose to display multiple fields in the list box by entering the list of fields to display as the value of the *LookupDisplay* property, separating each field with a semicolon. To display the resulting columns the way you want, use the *Options* property.

If the *ReadOnly* property is *False*, the user can select a displayed value in the database lookup list box and the corresponding value in current record of the primary dataset updates with a new value. Using the *Books* and *BookOrders* example, when the user selects a title in the lookup table using the database lookup list box, the value of the *CustNo* field in the primary dataset updates accordingly.

The *Value* property is the string the combo box uses to identify which record in the lookup table to display. The *DisplayValue* is the actual displayed string in the list box.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

Properties

Align	Fields	ParentFont
🔑 BorderStyle	Font	ParentShowHint

▷ BoundsRect	Height	PopupMenu
Color	HelpContext	🔑 ReadOnly
▷ ComponentIndex	Hint	▷ 🔑 SelectedField
Ctl3D	Left	▷ 🔑 SelectedIndex
Cursor	🔑 LookupDisplay	ShowHint
🔑 DataField	🔑 LookupField	TabOrder
🔑 DataSource	🔑 LookupSource	TabStop
▷ 🔑 DisplayValue	Name	Tag
DragCursor	Options	▷ 🔑 Value
DragMode	▷ Owner	Visible
▷ 🔑 EditorMode	▷ Parent	Width
Enabled	ParentColor	
▷ 🔑 FieldCount	ParentCtl3D	

Methods

BeginDrag	GetTextBuf	ScrollBy
BringToFront	GetTextLen	SendToBack
CanFocus	Hide	SetBounds
ClientToScreen	Invalidate	SetFocus
Dragging	Refresh	SetTextBuf
EndDrag	Repaint	Show
FindComponent	ScaleBy	Update
Focused	ScreenToClient	

Events

OnClick	OnEndDrag	OnKeyPress
OnDblClick	OnEnter	OnKeyUp
OnDragDrop	OnExit	
OnDragOver	OnKeyDown	

TDBLookupListOptions type

DBLookup

Declaration

```
TDBLookupListOption = (loColLines, loRowLines, loTitles);
```

```
TDBLookupListOptions = set of TDBLookupListOption;
```

The *TDBLookupListOptions* type defines the possible values contained in the *Options* set of a database lookup combo box (*TDBLookupCombo*) or database lookup list box (*TDBLookupList*).

TDBMemo component

DBCtrls

A *TDBMemo* component displays text for the user and permits the user display and enter data into a field much like a *TDBEdit* component. The *TDBMemo* component permits multiple lines to be entered or displayed, including text BLOBs (binary large objects).

Unlike an ordinary memo control, you can use the database memo to enter data into a field, or to simply display data from a field of the current record in a dataset. Link the database memo with a dataset by specifying the data source component (*TDataSource*) that identifies the dataset as the value of the memo's *DataSource* property. Specify the field in the dataset you want to access as the value of the *DataField* property.

The text in the database memo is the value of the *Text* property. When the value of *Text* changes, the new *Text* value becomes the value of the field for the current record in the dataset. If you just want the user to be able only to view the data in the field and not to change it, set the *ReadOnly* property to *True*.

Your application can tell if the value of *Text* changes by checking the value of the *Modified* property. To limit the number of characters users can enter into the database memo, use the *MaxLength* property.

You can access the text by line using the *Lines* property. If you want to work with the text as one chunk, use the *Text* property. If you want to work with individual lines of text, the *Lines* property will better suit your needs. Also, the first line of text is the value of the *Text* property, which can be up to 255 characters.

You can add, delete, insert, and move lines in a database memo control using the *Add*, *Delete*, and *Insert* methods of the *Lines* object, which is of type *TStrings*. For example, to add a line to a memo, you could write this line of code:

```
Memo1.Items.Add('Another line is added');
```

You can cut, copy, and paste text to and from a database memo control using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods.

If the memo displays a BLOB field, you can control when the text appears in the memo with the *AutoDisplay* property. You can also load the text using the *LoadMemo* method.

Several properties affect how the database memo appears and how text is entered. You can choose to supply scroll bars in the memo with the *ScrollBars* property. If you want the text to break into lines, set *WordWrap* to *True*. If you want the user to be able to use tabs in the text, set *WantTabs* to *True*.

At run time, you can select all the text in the memo with the *SelectAll* method. To find out which text in the memo the user has selected, or to replace selected text, use the *SelText* property. To select only part of the text or to find out what part of the text is selected, use the *SelStart* and *SelLength* properties.

If your application doesn't require the data-aware capabilities of *TDBMemo*, use a memo control (*TMemo*) instead to conserve system resources.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for DBMemo Component in the online Help, and choose the topic Using the DBMemo Component.

Properties

	Align		HelpContext	🔑	SelLength
🔑	Alignment		Hint	🔑	SelStart
🔑	AutoDisplay		Left	🔑	SelText
🔑	BorderStyle	▷ 🔑	Lines		ShowHint
	Color	🔑	MaxLength		Showing
▷	ComponentIndex	▷ 🔑	Modified		TabOrder
	Ctl3D		Name		TabStop
	Cursor	▷	Owner		Tag
🔑	DataField	▷	Parent	🔑	Text
🔑	DataSource		ParentColor		Top
	DragCursor		ParentCtl3D		Visible
	DragMode		ParentFont	🔑	WantTabs
	Enabled		ParentShowHint		Width
▷ 🔑	Fields		PopupMenu	🔑	WordWrap
	Font	🔑	ReadOnly		
	Height	🔑	ScrollBars		

Methods

	BeginDrag		GetTextBuf		ScrollBy
	BringToFront		GetTextLen	🔑	SelectAll
	CanFocus		Hide		SendToBack
	ClientToScreen		Invalidate		SetBounds
🔑	Clear	🔑	LoadMemo		SetFocus
	Dragging		Refresh	🔑	SetSelTextBuf
	EndDrag		Repaint		SetTextBuf
	Focused		ScaleBy		Show
🔑	GetSelTextBuf		ScreenToClient		Update

Events

OnChange	OnEndDrag	OnKeyUp
OnClick	OnEnter	OnMouseDown
OnDblClick	OnExit	OnMouseMove
OnDragDrop	OnKeyDown	OnMouseUp
OnDragOver	OnKeyPress	

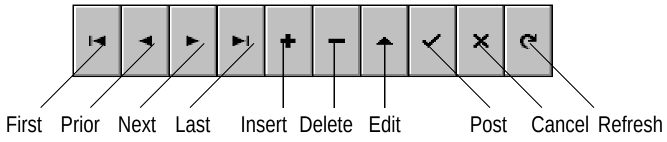
TDBNavigator component

DBCtrls

The *TDBNavigator* component (a database navigator) is used to move through the data in a database table or query, and perform operations on the data, such as inserting a blank record or posting a record. It is used in conjunction with the data-aware controls, such as the data grid, which give you access to the data, either for editing the data, or for simply displaying it.

You link the database navigator with a dataset when you specify a data source component that identifies the dataset as the value of navigator's *DataSource* property.

The database navigator consists of multiple buttons.



When the user chooses one of the navigator buttons, the appropriate action occurs on the dataset the navigator is linked to. For example, if the user clicks the Insert button, a blank record is inserted in the dataset.

This table describes the buttons on the navigator:

Button	Purpose
First	Sets the current record to the first record in the dataset, disables the First and Prior buttons, and enables the Next and last buttons if they are disabled
Prior	Sets the current record to the previous record and enables the Last and Next buttons if they are disabled
Next	Sets the current record to the next record and enables the First and Prior buttons if they are disabled
Last	Sets the current record to the last record in the dataset, disables the Last and Next buttons, and enables the First and Prior buttons if they are disabled
Insert	Inserts a new record before the current record, and sets the dataset into Insert and Edit states
Delete	Deletes the current record and makes the next record the current record
Edit	Puts the dataset into Edit state so that the current record can be modified
Post	Writes changes in the current record to the database
Cancel	Cancels edits to the current record, restores the record display to its condition prior to editing, and turns off Insert and Edit states if they are active
Refresh	Redisplays the current record from the dataset, thereby updating the display of the record on the form

Using the *VisibleButtons* property, you can decide which operations are allowed on the data and when.

You can customize the Help Hints available for the buttons on the database navigator by specifying values in the *Hints* property.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for DBNavigator in the online Help, and choose the topic Using the DBNavigator Component.

Properties

Align	Height	PopupMenu
▷ BoundsRect	HelpContext	▷ Showing
▷ ComponentIndex	🔑 Hint	ShowHint
ConfirmDelete	Hints	TabOrder
Ctl3D	Left	TabStop
Cursor	Name	Tag
DataSource	▷ Owner	Top
DragCursor	▷ Parent	Visible
DragMode	ParentCtl3D	🔑 VisibleButtons
Enabled	ParentShowHint	Width

Methods

BeginDrag	GetTextBuf	ScrollBy
BringToFront	GetTextLen	SendToBack
BtnClick	Hide	SetBounds
CanFocus	Invalidate	SetFocus
ClientToScreen	Refresh	SetTextBuf
Dragging	Repaint	Show
EndDrag	ScaleBy	Update
Focused	ScreenToClient	

Events

OnClick	OnEndDrag	OnMouseMove
OnDblClick	OnEnter	OnMouseUp
OnDragDrop	OnExit	🔑 OnResize
OnDragOver	OnMouseDown	

TDBRadioGroup component — DBCtrls

The *TDBRadioGroup* component displays a group of data-aware radio buttons. Only one of the radio buttons can be selected at a time, so the radio buttons present a set of mutually exclusive choices. Using a database radio button group box, you can ensure that the user must enter one of the presented options in a field, or the database radio group box can display the value of data in a field when the field is limited to a few possible values. For example, if only the values Red, Green, and Blue are valid in the field, the group box can have Red, Green, and Blue radio buttons.

Link the database radio group box with a dataset by specifying the data source component (*TDataSource*) that identifies the dataset as the value of the group box's *DataSource* property. Specify the field in the dataset you want to access as the value of the group box's *DataField* property.

The radio buttons are added to the group box when strings are entered as the value of the *Items* property. The strings entered in the *Items* property become the captions of the radio buttons if there are no strings in the *Values* property. If there are strings in the *Values* property, the first string is associated with the first radio button, the second with the second radio button, and so on. The *Values* string for a radio button is the value in the field of the current record that selects the radio button.

If the user selects a radio button and the *ReadOnly* property is *False*, the *Values* string for the radio button becomes the contents of the field for the current record in the dataset.

The *Value* property contains the contents of the field of the current record in the dataset.

You can display the radio buttons in a single column or in multiple columns by setting the value of the *Columns* property.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for DBRadioGroup in the online Help, and choose the topic Using the DBRadioGroup Component.

Properties

Align	Font	ParentShowHint
Caption	Height	PopupMenu
Color	HelpContext	🔑 ReadOnly
Columns	Hint	ShowHint
▷ ComponentIndex	▷ 🔑 ItemIndex	▷ Showing
Ctl3D	🔑 Items	TabOrder
Cursor	Left	TabStop
🔑 DataField	Name	Tag
🔑 DataSource	▷ Owner	Top
DragCursor	▷ Parent	▷ 🔑 Value
DragMode	ParentColor	🔑 Values
Enabled	ParentCtl3D	Visible
▷ 🔑 Fields	ParentFont	Width

Methods

BeginDrag	GetTextBuf	ScrollBy
BringToFront	GetTextLen	SendToBack
CanFocus	Hide	SetBounds
ClientToScreen	Invalidate	SetFocus
🔑 ContainsControl	Refresh	SetTextBuf
Dragging	Repaint	Show

EndDrag	ScaleBy	Update
Focused	ScreenToClient	

Events

OnChange	OnDragDrop	OnEnter
OnClick	OnDragOver	OnExit
OnDblClick	OnEndDrag	

TDBText component **DBCtrls**

The *TDBText* component is a data-aware control that displays text on a form. Your application can display the contents of a field in the current record of a dataset in a database text control, but the user won't be able to modify the field's contents.

Link the database text control with a dataset by specifying the data source component (*TDataSource*) that identifies the dataset as the value of the label's *DataSource* property. Specify the field in the dataset you want to access as the value of the label's *DataField* property.

The text of a database text control is the value of its *Caption* property. How the text of the caption aligns within the label is determined by the value of the *Alignment* property. You can have the text control resize automatically to fit a changing caption if you set the *AutoSize* property to *True*. If you prefer to have the text wrap, set *WordWrap* to *True*.

If you want a database text control to appear on top of a graphic, but you want to be able to see through the control so that part of the graphic isn't hidden, set the *Transparent* property to *True*.

If your application doesn't require the data-aware capabilities of *TDBText*, you should use the label component (*TLabel*) instead to conserve system resources.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all controls.

For more information, search for DBText component in the online Help, and choose the topic Using the DBText Component.

Properties

Align	Enabled	ParentShowHint
🔑 Alignment	▷ 🔑 Fields	PopupMenu
🔑 AutoSize	Font	ShowHint
▷ BoundsRect	Height	Tag
Color	Hint	Top
▷ ComponentIndex	Left	🔑 Transparent
Cursor	Name	Visible
🔑 DataField	▷ Owner	Width
🔑 DataSource	▷ Parent	🔑 WordWrap

DragCursor	ParentColor
DragMode	ParentFont

Methods

BeginDrag	GetTextBuf	ScreenToClient
BringToFront	GetTextLen	SendToBack
CanFocus	Hide	SetBounds
ClientToScreen	Invalidate	SetTextBuf
Dragging	Refresh	Show
EndDrag	Repaint	Update

Events

OnClick	OnDragOver	OnMouseMove
OnDblClick	OnEndDrag	OnMouseUp
OnDragDrop	OnMouseDown	

TDDEClientConv component **DDEMan**

A *TDDEClientConv* component establishes a Dynamic Data Exchange (DDE) conversation with a DDE server application. Use it in conjunction with a *TDDEClientItem* component to make your application a DDE client.

To link to a DDE server application, define the DDE conversation by specifying the server application name in the *DDEService* property and the topic of the DDE conversation in the *DDETopic* property. To establish a link at design time, click the ellipsis (...) button for *DDEService* or *DDETopic* in the Object Inspector and choose Paste Link in the DDE Info dialog box. To establish a link at run time, specify the service and topic with the *Setlink* method.

To send data to the DDE server, use the *PokeData* or *ExecuteMacro* methods. *PokeData* sends a text string to the linked item in the DDE server. *ExecuteMacro* sends a text string containing a macro command to be processed by the DDE server.

You can change the way a DDE conversation is established by specifying the *ConnectMode* property. If *ConnectMode* is *ddeAutomatic*, the client attempts to establish the conversation when the *TDDEClientConv* component is created at run time. If *ConnectMode* is *ddeManual*, you must write code that executes the *OpenLink* method to establish the DDE conversation.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all components.

For more information, search for TDDEClientConv component in the online Help, and choose the topic Using the DDE Client Conversation Component.

Properties

▷ ComponentIndex	🔑 DDETopic	▷ Owner
🔑 ConnectMode	🔑 FormatChars	🔑 ServiceApplication
🔑 DDEService	Name	Tag

Methods

🔑 CloseLink	🔑 OpenLink	🔑 RequestData
🔑 ExecuteMacro	🔑 PokeData	🔑 SetLink
🔑 ExecuteMacroLines	🔑 PokeDataLines	

Events

🔑 OnClose	🔑 OnOpen

TDDEClientItem component **DDEMan**

A *TDDEClientItem* component defines the item of a Dynamic Data Exchange (DDE) conversation. Use it in conjunction with a *TDDEClientConv* component to make your application a DDE client.

To link to a DDE conversation, specify the name of a *TDDEClientConv* component in the *DDEConv* property. The DDE item of the conversation should be specified in the *DDEItem* property. If the *TDDEClientConv* component has established a link with a DDE server, the server will automatically and continually update the client until the conversation is terminated.

The actual text data to exchange with the DDE server is specified in the *Text* property. When the server updates your DDE client, the new data will be automatically stored in the *Text* property. Whenever *Text* is updated, an *OnChange* event occurs. For text data longer than the 255 character limit of *Text*, use the *Lines* property to specify the text data to exchange.

In addition to these properties and events, this component also has the properties and methods that apply to all components.

For more information, search for TDDEClientItem component in the online Help, and choose the topic Using the DDE Client Item Component.

Properties

▷ ComponentIndex	🔑 Lines	Tag
🔑 DDEConv	Name	🔑 Text
🔑 DDEItem	▷ Owner	

Events

🔑 OnChange

TDDEServerConv component **DDEMan**

A *TDDEServerConv* component establishes a Dynamic Data Exchange (DDE) conversation with a DDE client application. Use it in conjunction with a *TDDEServerItem* component to make your application a DDE server.

If the DDE client sends a macro to your DDE server application, an *OnExecuteMacro* event occurs. You should write code in the *OnExecuteMacro* event handler to process this macro.

Using a *TDDEServerConv* component is optional. If you don't use a *TDDEServerConv* component, the client can still request an update directly from the *TDDEServerItem* component.

If you use a *TDDEServerConv* component, the DDE topic of the conversation is the value of the *Name* property of the *TDDEServerConv* component. If you don't use a *TDDEServerConv* component, the DDE topic of the conversation is the value of the *Caption* property of the form containing the *TDDEServerItem* component.

You should use a *TDDEServerConv* component with a *TDDEServerItem* component when the client might send a macro, or when the *Caption* of the form containing the *TDDEServerItem* might not be unique or constant at run time.

In addition to these properties and events, this component also has the properties and methods that apply to all components.

For more information, search for TDDEServerConv component in the online Help, and choose the topic Using the DDE Server Conversation Component.

Properties

▷ ComponentIndex	▷ Owner	Tag
Name		

Events

🔑 OnClose	🔑 OnExecuteMacro	🔑 OnOpen

TDDEServerItem component **DDEMan**

A *TDDEServerItem* component defines the item of a Dynamic Data Exchange (DDE) conversation. Use it by itself, or optionally, with a *TDDEServerConv* component to make your application a DDE server.

To use a *TDDEServerItem* component with a *TDDEServerConv* component, specify the name of a *TDDEServerConv* component in the *ServerConv* property.

The actual text data to exchange with the DDE client is specified in the *Text* property. When the client requests an update, your server sends the contents of the *Text* property to the client. When the *Text* property is modified, either by your own application or when the client pokes data, an *OnChange* event occurs. For text data longer than the 255-character limit of *Text*, use the *Lines* property to specify the data to exchange.

To test a link with a DDE client, use the *CopyToClipboard* method. This method will copy the contents of *Text* (or *Lines*), as well as DDE link information to the Clipboard. Then, if the DDE client can paste links, you can activate the client and paste the DDE data into the client application.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all components.

For more information, search for TDDEServerItem component in the online Help, and choose the topic Using the DDE Server Item Component.

Properties

▷ ComponentIndex	▷ Owner	🔑 Text
🔑 Lines	🔑 ServerConv	
Name	Tag	

Methods

🔑 CopyToClipboard

Events

🔑 OnChange	🔑 OnPokeData

TDirectoryListBox component **FileCtrl**

The *TDirectoryListBox* component is a specialized list box that is aware of the directory structure of the current drive. When the application runs, the user can use the directory list box to change directories, which changes the value of the *Directory* property.

The *Drive* property determines on which drive the list box displays the directory structure. When the value of *Drive* changes, the *Directory* value also changes to the current directory on the specified drive.

You can synchronize a directory list box with a file list box (*TFileListBox*), so that when the user uses a directory list box to change directories, the file list box displays the files in the new directory. This is the event handler for the *OnChange* event for the directory list box:

```
procedure TForm1.DirectoryListBox1Change(Sender: TObject);
```

```
begin
  FileListBox1.Directory := DirectoryListBox1.Directory;
end;
```

Note An *OnChange* event for a directory list box occurs when the user selects a new directory with the mouse or when the user move the selection bar and presses enter.

Another way to accomplish the same thing is to assign a file list box as the value of the *FileList* property. If you use the *DirLabel* property, you can have the caption of the label display the current directory.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for DirectoryListBox component in the online Help, and choose the topic Using the Directory List Box Component.

Properties

Align	Font	ParentShowHint
▷ BoundsRect	▷ Handle	PopupMenu
Color	Height	▷ 🔑 Selected
Columns	HelpContext	ShowHint
▷ ComponentIndex	Hint	▷ Showing
Ctl3D	🔑 IntegralHeight	Tag
Cursor	🔑 ItemHeight	Top
▷ 🔑 Directory	Left	TabOrder
🔑 DirLabel	Name	TabStop
DragCursor	▷ Owner	▷ 🔑 TopIndex
DragMode	▷ Parent	Visible
▷ 🔑 Drive	ParentColor	Width
Enabled	ParentCtl3D	
🔑 FileList	ParentFont	

Methods

BeginDrag	GetTextBuf	ScreenToClient
BringToFront	GetTextLen	ScrollBy
CanFocus	Hide	SendToBack
Clear	Invalidate	SetBounds
ClientToScreen	ItemAtPos	SetFocus
Dragging	ItemRect	SetTextBuf
EndDrag	Refresh	Show
Focused	Repaint	Update
🔑 GetItemPath	ScaleBy	

Events

OnClick	OnEndDrag	OnKeyUp
OnDblClick	OnEnter	OnMouseDown
OnDragDrop	OnExit	OnMouseMove
OnDragOver	OnKeyDown	OnMouseUp
OnDropDown	OnKeyPress	

TDragDropEvent type — Controls

Declaration

```
TDragDropEvent = procedure(Sender, Source: TObject; X, Y: Integer) of object;
```

The *TDragDropEvent* type points to a method that handles the dropping of a dragged object. The *Source* parameter is the object being dragged, *Sender* is the object the *Source* is being dropped on, and *X* and *Y* are screen coordinates in pixels.

TDragDropEvent is the type of the *OnDragDrop* event.

TDragMode type — Controls

Declaration

```
TDragMode = (dmManual, dmAutomatic);
```

The *TDragMode* type defines the values for the *DragMode* property of controls.

TDragOverEvent type — Controls

Declaration

```
TDragOverEvent = procedure(Sender, Source: TObject; X, Y: Integer; State: TDragState; var
Accept: Boolean) of object;
```

The *TDragOverEvent* type points to a method that handles the dragging of one object over another. The *Source* parameter is the object being dragged, *Sender* is the object the *Source* is being dragged over, *X* and *Y* are screen coordinates in pixels, *State* is the state of the drag object in relationship to the object being dragged over, and *Accept* determines whether the *Sender* recognizes the drag object. *Accept* does not default to *True* or *False*; you must assign the appropriate value to it.

TDragOverEvent is the type of the *OnDragOver* event.

See also

TDragState type

TDragState type

Controls

Declaration

```
TDragState = (dsDragEnter, dsDragLeave, dsDragMove);
```

The *TDragState* type specifies the drag state of a dragged control in relationship to another control. It is the type of the *State* parameter used in *OnDragOver* event handlers. These are the possible states:

Value	Meaning
dsDragEnter	The state a drag object is in when it enters a control that allows the drag object to be dropped. *dsDragEnter* is the default state.
dsDragMove	The state a drag object is in when it is moved within a control that allows the drag object to be dropped.
dsDragLeave	The state a drag object is in when it leaves a control that would allow the drag object to be dropped

Example

This code is a *OnDragOver* event handler that won't allow a label control to be dropped on a panel control and stops the dragging of the label as soon as the user drags the label onto the panel:

```
procedure TForm1.Panel1DragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  Accept := False;
  if (Source is TLabel) and (State = dsDragEnter) then
    (Source as TLabel).EndDrag(False);
end;
```

The *Source* parameter is the label being dragged, the *Sender* parameter is the panel control, and the *State* parameter is the drag state.

See also

TDragOverEvent type

TDrawCellEvent type

Grids

T

Declaration

```
TDrawCellEvent = procedure (Sender: TObject; ACol, ARow: Longint; ARect: TRect; AState:
TGridDrawState) of object;
```

The *TDrawCellEvent* type points to a method that handles the drawing of a cell in an owner-draw grid control. The *ACol* parameter is the column of the cell in the grid, and the *ARow* parameter is the row of the cell. *ARect* is the cell area where the drawing occurs, and *AState* is current state of the cell.

TDrawCellEvent is the type of the *OnDrawCell* event.

TDrawDataCellEvent type DBGrids

Declaration

```
TDrawDataCellEvent = procedure (Sender: TObject; const Rect: TRect; Field: TField; State:
TGridDrawState) of object;
```

The *TDrawDataCellEvent* type points to a method that handles the drawing of a cell in an owner-draw data grid control. The *Rect* parameter specifies the cell area where the drawing occurs. The *Field* parameter specifies which field the drawing takes place in, and the *State* parameter is the current state of the cell.

TDrawDataCellEvent is the type of the *OnDrawDataCell* event.

TDrawGrid component Grids

A *TDrawGrid* component is a grid control that permits the display of an existing data structure in column and row format.

The grid uses the *OnDrawCell* event to fill in the cells of the grid. If the *DefaultDrawing* property is *False*, the code you write in the *OnDrawCell* event handler draws in the cells. If *DefaultDrawing* is *True*, the contents of the cells are automatically drawn using some default values.

You can obtain the drawing area of a cell with the *CellRect* method. The *MouseToCell* method returns the column and row coordinates of the cell the mouse cursor is in.

You can determine which cell is selected in the grid by checking the value of the *Selection* property.

You can change the appearance and behavior of a data grid by changing the value of the *Options* property. For example, you can choose to allow the user to use the Tab key to move to a new column, you can decide to display grid lines between columns but not between rows, or let the user edit the data displayed in the grid.

Several properties affect the appearance of the grid. The *DefaultColWidth* and *DefaultRowHeight* properties determine the default widths and heights of the columns and rows. You can change the width or height of a specific column or row with the *ColWidths* and *RowHeights* properties. You can choose to have fixed or nonscrolling columns and rows with the *FixedCols* and *FixedRows* properties, and you can assign the color of the fixed columns and rows with the *FixedColor* property. Set the width of the grid lines with the *GridLineWidth* property. Add scroll bars to the grid with the *ScrollBars* property.

You can determine which row is currently the top row in the grid, or set a specified row to be the top row with the *TopRow* property. To determine which column is the first visible column in the grid, use the *LeftCol* property. The values of the *VisibleColCount* and *VisibleRowCount* properties are the number of columns and rows visible in the grid.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for DrawGrid component in the online Help, and choose the topic Using the Draw Grid Component.

Properties

Align	🔑 FixedColor	PopupMenu
🔑 BorderStyle	🔑 FixedCols	▷ 🔑 Row
▷ BoundsRect	🔑 FixedRows	🔑 RowCount
▷ Brush	Font	🔑 RowHeights
▷ Canvas	🔑 GridHeight	▷ 🔑 Scrollbars
▷ 🔑 Col	▷ 🔑 GridLineWidth	▷ 🔑 Selection
🔑 ColCount	🔑 GridWidth	▷ Showing
Color	Height	TabOrder
▷ 🔑 ColWidths	HelpContext	TabStop
▷ ComponentIndex	Hint	▷ TabStops
Ctl3D	Left	Tag
Cursor	▷ 🔑 LeftCol	Top
🔑 DefaultColWidth	Name	▷ 🔑 TopRow
🔑 DefaultDrawing	🔑 Options	Visible
🔑 DefaultRowHeight	▷ Owner	▷ 🔑 VisibleColCount
DragCursor	▷ Parent	▷ 🔑 VisibleRowCount
DragMode	ParentColor	Width
▷ 🔑 EditorMode	ParentCtl3D	
Enabled	ParentFont	

Methods

BeginDrag	EndDrag	ScaleBy
BringToFront	Focused	ScreenToClient
CanFocus	GetTextBuf	ScrollBy
🔑 CellRect	GetTextLen	SendToBack
ClassName	Hide	SetBounds
ClassParent	Invalidate	SetFocus
ClassType	🔑 MouseToCell	SetTextBuf
ClientToScreen	Refresh	Show
Dragging	Repaint	Update

Events

OnClick	OnEnter	OnMouseDown
🔑 OnColumnMoved	OnExit	OnMouseMove
OnDblClick	🔑 OnGetEditMask	OnMouseUp
OnDragDrop	🔑 OnGetEditText	🔑 OnRowMoved
OnDragOver	OnKeyDown	🔑 OnSelectCell

🔑 OnDrawCell	OnKeyPress	🔑 OnSetEditText
OnEndDrag	OnKeyUp	🔑 OnTopLeftChanged

TDrawItemEvent type **StdCtrls**

Declaration

```
TDrawItemEvent = procedure(ListBox: TListBox; Index: Integer; Rect: TRect; State:
TOwnerDrawState) of object;
```

The *TDrawItemEvent* type points to a method that handles the drawing of an item in an owner-draw list box. The *Index* parameter is the position of the item in the list box, *Rect* is the area in the list box where the item is to be drawn, and *State* is the current state of the item in the list box. These are the possible values of *State*:

Value	Meaning
odSelected	The item is selected
odDisabled	The entire list box is disabled
odFocused	The item currently has focus

TDrawItemEvent is the type of the *OnDrawItem* event.

TDrawTabEvent type **Tabs**

Declaration

```
TDrawTabEvent = procedure(Sender: TObject; TabCanvas: TCanvas; R: TRect; Index: Integer;
Selected: Boolean) of object;
```

The *TDrawTabEvent* type points to a method that handles the drawing of an item in an owner-draw tab. The *TabCanvas* parameter is the canvas on which the item is drawn, The *Index* parameter is the position of the tab in the tab set control, *R* is the area in the tab where the item is to be drawn, and *Selected* indicates whether the tab is currently selected or not.

TDrawTabEvent is the type of the *OnDrawTab* event.

TDriveComboBox component **FileCtrl**

The *TDriveComboBox* component is a specialized combo box that displays all the drives available when the application runs. When the user uses the combo box to select another drive, the value of the *Drive* property changes.

You can specify whether the text in the drive combo box is uppercase or lowercase with the *TextCase* property.

If your application uses the drive combo box with a directory list box and a file list box, you can synchronize them with this code written in the drive combo box *OnChange* event handler and in the directory list box *OnChange* event handler:

```
procedure TForm1.DriveComboBox1Change(Sender: TObject);
begin
  DirectoryListBox1.Drive := DriveComboBox1.Drive;
end;

procedure TForm1.DirectoryListBox1Change(Sender: TObject);
begin
  FileListBox1.Directory := DirectoryListBox1.Directory;
end;
```

Now when the user selects a new drive using the drive combo box, the directory list box and the file list box are updated also.

Another way to accomplish the same task is to set the *DirList* property of the drive combo box and the *FileList* property of the directory list box.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for DriveComboBox component in the online Help, and choose the topic Using the Drive Combo Box Component.

Properties

▷	Align		HelpContext	▷ 🔑	SelStart
▷	BoundsRect		Hint	▷ 🔑	SelText
	Color	▷ 🔑	ItemIndex	▷	Showing
▷	ComponentIndex	▷ 🔑	Items		TabOrder
	Ctl3D		Left		TabStop
	Cursor		Name		Tag
🔑	DirList	▷	Owner	▷ 🔑	Text
	DragCursor	▷	Parent	🔑	TextCase
	DragMode		ParentColor		Top
▷	Drive		ParentCtl3D		Visible
	Enabled		ParentFont	▷ 🔑	Width
	Font		PopupMenu		
	Height	▷ 🔑	SelLength		

Methods

	BeginDrag		GetTextBuf		ScrollBy
	BringToFront		GetTextLen	🔑	SelectAll
	CanFocus		Hide		SendToBack
🔑	Clear		Invalidate		SetBounds
	ClientToScreen		Refresh		SetFocus
	Dragging		Repaint		SetTextBuf

EndDrag	ScaleBy	Show
Focused	ScreenToClient	Update

Events

OnChange	OnDragOver	OnExit
OnClick	OnDropDown	OnKeyDown
OnDblClick	OnEndDrag	OnKeyPress
OnDragDrop	OnEnter	OnKeyUp

See also

TDirectoryListBox component, *TFileListBox* component

TDuplicates type — Classes

Declaration

```
TDuplicates = (dupIgnore, dupAccept, dupError);
```

The *TDuplicates* type defines the possible values of the *Duplicates* property of a string list object (*TStringList*).

TEdit component — StdCtrls

Use a *TEdit* component to put a standard Windows edit box control on your form. Edit boxes are used to retrieve information from the user, because the user can type data into an edit box. Edit boxes can also display information to the user.

When users enter data into an edit box or the application displays information to the user in the edit box, the value of the edit box's *Text* property changes. Your application can tell if the value of *Text* changes by checking the value of the *Modified* property. To limit the number of characters users can enter into the edit box, use the *MaxLength* property.

You can specify whether the text in the edit box is uppercase and lowercase with the *CharCase* property.

If you want to prevent the user from changing the value of the *Text* property, set the *ReadOnly* property to *True*.

You can choose to have the text in an edit box automatically selected whenever it becomes the active control with the *AutoSelect* property. At run time, you can select all the text in the edit box with the *SelectAll* method. To find out which text in the edit box the user has selected or to replace selected text, use the *SelText* property. To clear selected text, call the *ClearSelection* method. To select only part of the text or to find out what part of the text is selected, use the *SelStart* and *SelLength* properties.

You can cut, copy, and paste text to and from an edit box using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods.

Your application can replace the characters typed into the edit box with another specified character. If the edit box is used to enter a password, for example, onlookers won't be able to read the typed characters. Specify the replacement character with the *PasswordChar* property.

If you want the edit box to automatically resize to accommodate a change in font size, use the *AutoSize* property.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information about using edit boxes, search for Edit component in the online Help, and choose the topic Using the Edit Component.

Properties

▷	Align		HideSelection	🔑	ReadOnly
🔑	AutoSelect		Hint	▷ 🔑	SelLength
🔑	AutoSize		Left	▷ 🔑	SelStart
	BorderStyle	🔑	MaxLength	▷ 🔑	SelText
🔑	CharCase	🔑	Modified		ShowHint
	Color		Name	▷	Showing
▷	ComponentIndex		OEMConvert		TabOrder
	Ctl3D	▷	Owner		TabStop
	Cursor	▷	Parent		Tag
	DragCursor		ParentColor	🔑	Text
	DragMode		ParentCtl3D		Top
	Enabled		ParentFont		Visible
	Font		ParentShowHint		Width
	Height	🔑	PasswordChar		
	HelpContext		PopupMenu		

Methods

	BeginDrag		Free		ScreenToClient
	BringToFront	🔑	GetSelTextBuf		ScrollBy
	ClientToScreen		GetTextBuf	🔑	SelectAll
🔑	Clear		GetTextLen		SendToBack
🔑	ClearSelection		Hide		SetBounds
🔑	CopyToClipboard		Invalidate		SetFocus
🔑	CutToClipboard	🔑	PasteFromClipboard	🔑	SetSelTextBuf
	Dragging		Refresh		SetTextBuf
	EndDrag		Repaint		Show
	FindComponent		ScaleBy		Update

Events

OnChange	OnEnter	OnMouseDown
OnDblClick	OnExit	OnMouseMove
OnDragDrop	OnKeyDown	OnMouseUp
OnDragOver	OnKeyPress	
OnEndDrag	OnKeyUp	

See also

TDBEdit component, *TComboBox* component

TEditCharCase type **StdCtrls**

Declaration

```
TEditCharCase = (ecNormal, ecUpperCase, ecLowerCase);
```

The *TEditCharCase* type defines the possible values for the *CharCase* property of an edit box (*TEdit*).

Temporary property

Applies to

TDataBase component

Declaration

```
property Temporary: Boolean;
```

Run-time only. The *Temporary* property is *True* if the *TDatabase* component was created because none existed when a database table was opened. Such a database will automatically be destroyed when the table or query is closed. You can set *Temporary* to *False* so that it will be preserved until you explicitly free it with *Free*. If you explicitly created the *TDatabase* component, then *Temporary* will be *False*, but you can set it to be *True* and it will automatically be freed when the last dataset linked to it is closed.

Example

```
Table1.Database.Temporary := False;
```

See also

Database property

TEndDragEvent type

Controls

Declaration

```
TEndDragEvent = procedure(Sender, Target: TObject; X, Y: Integer) of object;
```

The *TEndDragEvent* type points to a method that handles the stopping of the dragging of an object. The *Sender* is the object being dragged, *Target* is the object *Sender* is dragged to, and *X* and *Y* are screen coordinates in pixels.

TEndDragEvent is the type of the *OnEndDrag* event.

Terminate method

Applies to
TApplication component

Declaration

```
procedure Terminate;
```

The *Terminate* method stops the execution of your application.

Example
This example uses a button on a form. When the user clicks the button, a message box appears, asking the user if the application should terminate. If the user chooses Yes, the application ends.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if MessageDlg('Ternminate the application?', mtConfirmation,
    [mbYes, mbNo], 0) = mrYes then
      Application.Terminate;
end;
```

See also
Run method, *Terminated* property

Terminated property

Applies to
TApplication component

Declaration

```
property Terminated: Boolean;
```

Read and run-time only. The *Terminated* property determines whether the application has received the Windows WM_QUIT message, which then terminates the application. Your Delphi application receives this message usually because the main window of the application has closed, or the *Terminate* method has been called, thereby requiring windows to quit the application.

The *Terminated* property is usually used when calling the *ProcessMessages* method so that your application doesn’t attempt to process Windows messages after the application has quit.

Example

The application calls the *ProcessMessages* method if the application has not received the message from Windows to quit executing:

```
if Application.Terminated = False then
  Application.ProcessMessages;
  ...
```

See also

ProcessMessages method, *Run* method, *Terminate* method

Test8086 variable — System

Declaration

```
var Test8086: Byte;
```

The *Test8086* variable identifies the type of 80x86 processor the system contains.

The run-time library's startup code contains detection logic that automatically determines what kind of 80x86 processor the system contains. The result of the CPU detection is stored in *Test8086* as one of the following values:

Value	Definition
0	Processor is an 8086.
1	Processor is an 80286.
2	Processor is an 80386 or later.

When the run-time library detects that the processor is an 80386 or later CPU, it will use 80386 instructions to speed up certain operations. In particular, *Longint* multiplication, division, and shifts are performed using 32-bit instructions when an 80386 is detected.

TExceptionEvent type — Forms

Declaration

```
TExceptionEvent = procedure (Sender: TObject; E: Exception) of object;
```

The *TExceptionEvent* type points to a method that handles exceptions in your application. The *Sender* parameter is the object that raised the exception and *E* is the exception message.

TExceptionEvent is the type of the *OnException* event.

Text property

Applies to

TOutlineNode, *TParam* objects; *TComboBox*, *TDBComboBox*, *TDBEdit*, *TDBMemo*, *TDDEClientItem*, *TDDEServerItem*, *TDriveComboBox*, *TEdit*, *TFilterComboBox*, *TMaskEdit*, *TMemo*, *TQuery*, *TBCDField*, *TBooleanField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TIntegerField*, *TSmallintField*, *TStringField*, *TTimeField*, *TWordField* components

The *Text* property specifies a text string to appear in a component or object.

For edit boxes and memo controls

Declaration

```
property Text: TCaption;
```

The *Text* property of a component determines the text that appears within an edit box or memo control. The default text is the name of the control. Your application can use the value of *Text* as input into the application, or to display data to the user. The maximum length of the string in the *Text* property is 255 characters.

The value of the *Text* property of a mask edit box (*TMaskEdit*) or database edit box (*TDBEdit*) or database memo (*TDBMemo*) includes the text and the literal mask characters specified with the *EditText* property if the user chooses to save the mask characters with the text. If the mask characters are not saved, the text does not include them.

The *Text* property of a database edit box or database memo is available at run time only. You should seldom assign a new value to the *Text* property of a database edit box or memo. If the dataset is read only when the new value is assigned to *Text*, the contents of the field won’t change. Instead, you should change the value of the underlying field by using the *Fields* property of the edit box. For example,

```
DBEdit1.Field.AsString := ‘New value’;
```

Example

This example uses an edit box, a list box, and a button named *Add* on a form. Each time the user clicks the *Add* button, the text in the edit box is added to the list in the list box:

```
procedure TForm1.AddClick(Sender: TObject);
begin
  ListBox1.Items.Add(Edit1.Text);
end;
```

See also

GetSelTextBuf method, *GetTextBuf* method, *SetTextBuf* method, *SetSelTextBuf* method

For combo boxes

Declaration

```
property Text: string;
```

The value of the *Text* property is the first item that appears in the combo box when the application runs. For simple combo boxes, the user can change the value of *Text* by entering a new value.

For other types of combo boxes, the value of *Text* is read only and accessible only at run time. The value of the *Drive* property determines the value of the *Text* property in a drive combo box. The value of the *Filter* property determines the value of the *Text* property in a filter combo box. If the *Filter* property specifies multiple filters, the first filter in the *Filter* string appears first in the filter combo box.

For the database lookup combo box, *Text* is the value of the field of the current record.

Example

The following code stores the value of the first item of a combo box in the *Text* property in the *OnCreate* event handler of the form containing the combo box. The first item will be displayed in the combo box at run time.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ComboBox1.Text := ComboBox1.Items[0];
end;
```

See also

Drive property, *Filter* property

For outline nodes

Declaration

```
property Text: string;
```

The *Text* property contains the string that identifies an outline item. This string is displayed in the outline. The value of *Text* can be assigned directly or can be set by the *Lines* property of the *TOutline* component. If set with the *Lines* property, each line, minus any leading spaces or tabs, is assigned to the *Text* property of an individual *TOutlineNode* object.

Example

The following code should be attached to the *OnClick* event handler of an *TOutline* component. When the value of the *Text* of the *SelectedItem* is ‘Aqua’, the *Color* of the outline becomes *clAqua*.

```
procedure TForm1.Outline1Click(Sender: TObject);
begin
  if Outline1.Items[Outline1.SelectedItem].Text = 'Aqua' then
    Outline1.Color := clAqua;
end;
```

See also
Data property

For DDE items

Declaration

```
property Text: string;
```

The *Text* property contains the text data to exchange in a DDE conversation. For *TDDEClientItem* components, *Text* specifies the text that is updated by the DDE server application. For *TDDEServerItem* components, *Text* specifies the text that is sent to any DDE clients when the value of *Text* changes or when a client requests an update. When *Text* is changed, an *OnChange* event occurs.

Text corresponds to the *Lines* property. Whenever the value of *Text* or *Lines* is changed, the other is updated so that the first line of *Lines* is always equal to *Text*. Use *Text* to contain text values up to 255 characters in length (which is the limit of the *Text* property). For longer strings, use the *Lines* property.

If the *Text* property is of a *TDDEClientItem* component, you can also send the text in *Text* directly to the DDE server by poking data with the *PokeData* method.

If the *Text* property is of a *TDDEServerItem* component, the DDE client can change *Text* by poking data. The poked data replaces the contents of *Text* and an *OnChange* event occurs.

Example
The following code assigns the value to the *Text* property of *DDEClientItem1* to the *Caption* of *Label1*. This code is executed in the *OnChange* event handler of *DDEClientItem1*, so whenever the client is updated, the new data from the server is displayed.

```
procedure TForm1.DdeClientItem1Change(Sender: TObject);
begin
  Label1.Caption := DDEClientItem1.Text
end;
```

For queries

Declaration

```
property Text: PChar;
```

Run-time and read only. The *Text* property holds the actual text of the SQL query sent to the Borland Database Engine. In general, you should not need to examine this property. However, if you encounter problems with an SQL statement, you may want to inspect the *Text* property to be sure that the statement is as expected.

Example

```
var
  ActualText: PChar;
  Buf: array [0..20] of Char;
...
  ActualText := Query1.Text;
  repeat
    StrLCopy(Buf, ActualText, SizeOf(Buf));
    WriteLn(Buf);
    if StrLen(ActualText) > 20 then Inc(ActualText, 20)
    else Break;
    until False;
```

See also

SQL property

For fields

Declaration

```
property Text: string;
```

Run-time only. *Text* contains the string value of the field a data-aware control uses for display when the control is in edit mode. Data-aware controls such as *TDBEdit* rely on *Text* to provide the editing format for each field.

You can control the strings returned by *Text* by assigning an *OnGetText* event handler, or you can accept Delphi defaults, which depend on the field’s data type.

For *TStringField*, the *AsString* property is returned.

For *TIntegerField*, *TSmallintField*, and *TWordField*, if *EditFormat* or *DisplayFormat* (in that order) is assigned a value, *FloatToTextFmt* is called. Otherwise, *Str* is called.

For *TBCDField* and *TFloatField*, *FloatToTextFmt* is called with value of *EditFormat* or *DisplayFormat* (in that order).

For a *TCurrencyField*, if *EditFormat* or *DisplayFormat* (in that order) is assigned a value, *FloatToTextFmt* is called. Otherwise, *FloatToTextFmt* is called with the *ffCurrency* flag and *CurrencyDecimals* variable.

For a *TDateTimeField*, *DateTimeToStr* is called with the value of *DisplayFormat*.

For a *TDateField*, *DateTimeToStr* is called with the *DisplayFormat* property, except that the *ShortDateFormat* variable is substituted if *DisplayFormat* is unassigned. For a *TTimeField*, *DateTimeToStr* is called with the *DisplayFormat* property, except that the *LongTimeFormat* variable is substituted if *DisplayFormat* is unassigned.

Example

```
Edit.Text := Field1.Text;
```

For TParam objects

Declaration

```
property Text: string;
```

The *Text* property is similar to the *AsString* property. Accessing the *Text* property attempts to convert the current data to a string value and returns that value. If the current data is NULL, the value is an empty string.

Example

```
{ Assign ‘1221’ to the CustNo parameter }
Parameters.ParamByName(‘CustNo’).Text := ‘1221’;
```

See also

TFieldType type, *AsString* property, *DateToStr* function, *TimeToStr* function, *DateTimeToStr* function, *IntToStr* function, *FloatToStr* function, *StrToInt* function, *StrToFloat* function, *StrToDate* function, *StrToTime* function, *StrToDateTime* function, *TDateTime* type

TextCase property

Applies to

TDriveComboBox component

Declaration

```
property TextCase: TTextCase;
```

The *TextCase* property determines if the volume name in the *Text* property appears in uppercase or lowercase. These are the possible values:

Value	Meaning
tcLowerCase	The volume name specified in the *Text* property is displayed in lowercase letters.
tcUpperCase	The volume name specified in the *Text* property is displayed in uppercase letters.

The default value is *tcLowerCase*. If you use the Object Inspector to change the value to *tcUpperCase*, you won’t see the results until your application runs.

Example

Assuming a drive combo box exists on the form, this code displays the volume name in the drive combo box in uppercase letters when the form appears:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DriveComboBox1.TextCase := tcUpperCase;
end;
```

TextFile type **System**

The type *TextFile* is used to declare text file variables. Borland Pascal uses the type *Text* but to avoid confusion with the *Text* property, Delphi uses *TextFile*. Note that the type *Text* is still supported by Delphi, and if you want to use it you should always qualify it.

Example

The following example declares the variable *F* as a text file.

```
var
  F: TextFile;
```

See also

AssignFile procedure, *CloseFile* procedure

TextHeight method

Applies to

TCanvas object

Declaration

```
function TextHeight(const Text: string): Integer;
```

TextHeight returns the height in pixels of the string passed in *Text* when rendered in the current font. You can use *TextHeight* to specify whether the entire string will appear in a given space.

Example

This example displays the height of a text string in the current font of the canvas in an edit box on the form:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  L: LongInt;
begin
  L := Canvas.TextHeight('Object Pascal is the best');
  Edit1.Text := IntToStr(L) + ' pixels in height';
end;
```

See also

Font property, *TextWidth* method

TextOut method

Applies to

TCanvas object

Declaration

```
procedure TextOut(X, Y: Integer; const Text: string);
```

TextOut draws the string contained in *Text* on the canvas using the current font, with the upper left corner of the text at the point (*X*, *Y*).

Example

This example displays a text string at a specified position on the form when the user clicks the button on the form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Canvas.TextOut(20, 20, 'Delphi makes Windows programming easy');
end;
```

See also

TextRect method

TextRect method

Applies to

TCanvas object

Declaration

```
procedure TextRect(Rect: TRect; X, Y: Integer; const Text: string);
```

The *TextRect* method displays text inside a clipping rectangle. Any portions of the text passed in the *Text* parameter that fall outside the rectangle passed in the *Rect* parameter are clipped and don't appear onscreen. The upper left corner of the text is placed at the point (*X*, *Y*).

Example

The following code outputs the text “Hello, world!” in a rectangle defined by the coordinates (10, 10) and (100, 100). By passing 0 for *X* and 0 for *Y*, the top and left edges of the text will be clipped by the rectangle.

```
var
  TheRect: TRect;
begin
  TheRect.Top := 10;
  TheRect.Left := 10;
  TheRect.Bottom := 100;
  TheRect.Right := 100;
  Form1.Canvas.TextRect(TheRect,0,0,'Hello, world!');
end;
```

See also

TextOut method

TextToFloat function SysUtils

Declaration

```
function TextToFloat(Buffer: PChar; var Value: Extended): Boolean;
```

TextToFloat converts the null-terminated string given by *Buffer* to a floating-point value which is returned in the variable given by *Value*.

The return value is *True* if the conversion was successful, or *False* if the string is not a valid floating-point value.

For further details, see the description of the *StrToFloat* function.

TextToShortCut function Menus

Declaration

```
function TextToShortCut(Text: string): TShortCut;
```

The *TextToShortCut* function creates a menu shortcut from a text string. For example, your application can allow the user to specify what they want the shortcut to be in an edit box control, then *TextToShortCut* can use that string to create a menu shortcut.

Note The *TextToShortCut* function executes slowly. Unless you are getting input from the user, your application should use the *ShortCut* function to create a menu shortcut.

Example

This example uses a main menu that contains an Open command, an edit box, and a button. Delphi automatically names the menu item for the Open command *Open1*. When the user enters the desired shortcut text in the edit box and clicks the button, a shortcut is created and the shortcut text appears next to the Open menu item.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NewShortCut: TShortCut;
begin
  NewShortCut := TextToShortCut(Edit1.Text);
  Open1.ShortCut := NewShortCut;
end;
```

See also
ShortCut function, *ShortCut* property, *ShortCutToKey* procedure, *ShortCutToText* function

TextWidth method

Applies to
TCanvas object

Declaration

```
function TextWidth(const Text: string): Integer;
```

The *TextWidth* method returns the width in pixels of the string passed in *Text* when rendered in the current font. You can use *TextWidth* to determine whether a given string will fit in a particular space.

Example
This example determines the width of a specified string, and if the string is too wide to display in an edit box, the edit box is widened to accommodate the string. The string displays in the edit box.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  T: Longint;
  S: string;
begin
  S := 'Object Pascal is the language for me';
  T := Canvas.TextWidth(S);
  if T > Edit1.Width then
    Edit1.Width := T + 10;
  Edit1.Text := S;
end;
```

See also
TextRect method

TFDApplyEvent type Dialogs

Declaration

```
TFDApplyEvent = procedure(Sender: TObject; Wnd: HWND) of object;
```

The *TFDApplyEvent* type points to a method that performs an action when the user chooses the Apply button in the Font dialog box. *TFDApplyEvent* is the type of the *OnApply* event.

TField component

TField components are used to access fields in a record. By default, a set of *TField* components is created automatically each time a dataset component is activated; the resulting set of *TField* components is dynamic, mirroring the actual columns in an underlying physical table at that time.

At design time, you can use the Fields Editor to create a persistent, unchanging set of *TField* components for a dataset. Creating *TField* components with the Fields Editor provides efficient, readable, and type-safe programmatic access to underlying data. It guarantees that each time your application runs, it uses and displays the same columns, in the same order, every time, even if the physical structure of the underlying database has changed. Creating *TField* components at design time guarantees that data-aware components and program code that rely on specific fields always work as expected. If a column on which a persistent *TField* component is based is deleted or changed, then Delphi generates an exception rather than running the application against a nonexistent column or mismatched data.

A *TField* component is an abstract object. The *Fields* property of a dataset is always one of the following *TField* descendants:

Component:	**Used for:**
TStringField	Fixed length text data up to 255 characters
TIntegerField	Whole numbers in the range –2,147,483,648 to 2,147,483,647
TSmallintField	Whole numbers in the range –32,768 to 32,767
TWordField	Whole numbers in the range 0 to 65,535
TFloatField	Real numbers with absolute magnitudes from $5.0*10^{-324}$ to $1.7*10^{308}$ accurate to 15–16 digits
TCurrencyField	Currency values. The range and accuracy is the same as *TFloatField*
TBCDField	Real numbers with a fixed number of digits after the decimal point. Accurate to 18 digits. Range depends on the number of digits after the decimal point. [Paradox only]
TBooleanField	*True* or *False* values
TDateTimeField	Date and time value
TDateField	Date value
TTimeField	Time value
TBlobField	Arbitrary data field without a size limit
TBytesField	Arbitrary data field without a size limit
TVarBytesField	Arbitrary data field up to 65,535 characters, with the actual length stored in the first two bytes
TMemoField	Arbitrary length text
TGraphicField	Arbitrary length graphic, such as a bitmap

Each *TField* component and its properties, methods, and events can be accessed programmatically. At run time, dynamically created components can be accessed through the *Fields* property of the dataset; at design time, use the Fields Editor to select a field component and use the Object Inspector to modify the field's properties.

Most *TField* descendants have the same properties, but some properties, such as *AsBoolean* or *EditMask* only apply to some fields. Use the *AsBoolean*, *AsDateTime*, *AsFloat*, *AsInteger*, or *AsString* properties as appropriate to access or modify the current value of the field. Test the *CanModify* property to see if the field can be changed. Use the *DataSet* property to reference the dataset of the field. Use the *DataType* property to test the type of the field. Set the *DisplayLabel* property to a column heading for a data grid(*TDBGrid*). The *DisplayText* property will format the field for display purposes; *Text* will format it for editing purposes. Set the *DisplayWidth* to control the column width in a data grid. Set the *EditMask* property to limit the characters entered to a selected set. Use the *FieldName* property to get the name of the field in the dataset. Test the *IsNull* property to see if the field has been assigned a value. Set the *ReadOnly* property to prevent or allow the user to change the value. Set the *Visible* property to control whether the field appears in a data grid. Call the *Clear* method to erase any data assigned. Call the *GetData* method to access the data in native format, or *SetData* to assign new data. Use the *OnChange* event to be notified when the value of the field is changed. Use the *OnGetText* event to do your own formatting of the data for display or edit purposes, and the *OnSetText* event to convert the edited data back to native format. Use the *OnValidate* event to validate the data before it is stored into the record.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all components.

Properties

Alignment	▷ DataType	▷ IsIndexField
▷ AsBoolean	DisplayLabel	▷ IsNull
▷ AsDateTime	▷ DisplayName	Name
▷ AsFloat	▷ DisplayText	▷ Owner
▷ AsInteger	DisplayWidth	ReadOnly
▷ AsString	EditMask	Required
Calculated	▷ EditMaskPtr	▷ Size
▷ CanModify	FieldName	Tag
▷ DataSet	▷ FieldNo	▷ Text
▷ DataSize	Index	Visible

Methods

Assign	FocusControl	SetData
AssignValue	GetData	
Clear	IsValidChar	

Events

OnChange	OnSetText	OnValidate
OnGetText		

TFieldGetTextEvent type **DB**

Declaration

```
TFieldGetTextEvent = procedure(Sender: TField; var Text: string; DisplayText: Boolean) of
object;
```

The *TFieldGetTextEvent* points to a method that retrieves the text in the field. It is used by the *OnGetText* event of field components. The *Text* parameter references the text, and the *DisplayText* parameter determines whether the text is formatted for display. If *DisplayText* is *True*, the text is in display format. If *DisplayText* is *False*, the text is not formatted for display.

TFieldNotifyEvent type **DB**

Declaration

```
TFieldNotifyEvent = procedure(Sender: TField) of object;
```

The *TFieldNotifyEvent* type points to a method that handles the validation of data in a field or handles the changing of data in a field. It is the type of the *OnChange* and *OnValidate* events of a field component.

TFieldSetTextEvent type **DB**

Declaration

```
TFieldSetTextEvent = procedure(Sender: TField; const Text: string) of object;
```

The *TFieldSetTextEvent* type points to a method that stores text in a field. It is used by the *OnSetText* event of field components. The *Text* parameter is the text that is being stored in the field.

TFileEditStyle type Dialogs

Declaration

```
TFileEditStyle = (fsEdit, fsComboBox);
```

The *TFileEditStyle* type contains the possible values of the *FileEditStyle* property used by the Open (*TOpenDialog*) and Save (*TSaveDialog*) dialog boxes.

TFileExt type Dialogs

Declaration

```
TFileExt = string[3];
```

The *TFileExt* type is used to hold the three characters of a file-name extension. *TFileExt* is used by the *DefaultExt* property of the Open and Save dialog boxes (*TOpenDialog* and *TSaveDialog*).

TFileListBox component FileCtrl

The *TFileListBox* component is a specialized list box that lists all the files in the current directory. To display files in a different directory, change the value of the *Directory* property.

You can have icons next to the file names to help identify the type of file. For example, an executable file displays a different icon than a word processing document. To make the icons appear, set *ShowGlyphs* to *True*.

You decide which file types you want to appear in the list box using the *Mask* property, which displays only the files that match the *Mask* string. For example, you can choose to display only executable files and source code files.

You can also decide which files display in the file list box by their file attributes. For example, you can choose to display hidden and system files as well as regular files, or you can choose to see read-only files only. Use the *FileType* property to select the file types according to their file attributes.

You can have the file selected in the file list box appear as the text of an edit box if you specify a value for the *FileEdit* property.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for FileListBox component in the online Help, and choose the Using the File List Box Component.

Properties

Align	▷ Handle	ParentFont
▷ BoundsRect	Height	ParentShowHint
▷ Canvas	HelpContext	PopupMenu
▷ ComponentIndex	Hint	🔑 Selected
Color	🔑 IntegralHeight	🔑 ShowGlyphs
▷ Controls	🔑 ItemHeight	ShowHint
Ctl3D	▷ 🔑 ItemIndex	▷ Showing
Cursor	▷ 🔑 Items	TabOrder
▷ 🔑 Directory	Left	TabStop
DragCursor	🔑 Mask	Tag
DragMode	▷ 🔑 MultiSelect	Top
Enabled	Name	▷ 🔑 TopIndex
🔑 FileEdit	▷ Owner	Visible
🔑 FileName	▷ Parent	Width
🔑 FileType	ParentColor	
Font	ParentCtl3D	

Methods

BeginDrag	GetTextLen	ScrollBy
BringToFront	Hide	SendToBack
CanFocus	Invalidate	SetBounds
🔑 Clear	🔑 ItemAtPos	SetFocus
ClientToScreen	🔑 ItemRect	SetTextBuf
Dragging	Refresh	Show
EndDrag	Repaint	Update
Focused	ScaleBy	
GetTextBuf	ScreenToClient	

Events

OnChange	OnEndDrag	OnKeyUp
OnClick	OnEnter	OnMouseDown
OnDblClick	OnExit	OnMouseMove
OnDragDrop	OnKeyDown	OnMouseUp
OnDragOver	OnKeyPress	

See also

TDirectoryListBox component, *TDriveComboBox* component

TFieldClass type DB

Declaration

```
TFieldClass = class of TField;
```

The *TFieldClass* type is the object type of *TField*. Use it to create an object reference to a *TField*.

TFieldDef object

The *TFieldDef object* corresponds to a physical field of a record in a table underlying a dataset. *TFieldDef* objects are created automatically for dataset components. A field definition has a corresponding *TField* component, but not all *TField* components have a corresponding *TFieldDef* objects. For example, calculated field do not have *TFieldDef* objects.

In addition to these properties and methods, this object also has the methods that apply to all objects.

Properties

▷ 🔑 DataType	▷ 🔑 FieldNo	▷ 🔑 Required
▷ 🔑 FieldClass	▷ 🔑 Name	▷ 🔑 Size

Methods

🔑 ClassName	🔑 Create	🔑 Free
🔑 ClassParent	🔑 CreateField	
🔑 ClassType	🔑 Destroy	

TFieldDefs object

A *TFieldDefs* object holds the *TFieldDef* objects that represent the physical fields underlying a dataset.

The *Count* property is the total number of *TFieldDef* objects in *TFieldDefs*. The *Items* property is an array of pointers to the *TFieldDef* objects.

Use the *Find* or *IndexOf* methods to locate an entry in *Items* by name. Call *Clear* to remove all *TFieldDef* objects from *TFieldDefs*. Call *Update* to obtain information about the fields in a dataset without opening it.

In addition to these properties and methods, this object also has the methods that apply to all objects.

Properties

Count	Items

Methods

Add	ClassType	Free
AddFieldDesc	Clear	IndexOf
Assign	Create	Update
ClassName	Destroy	
ClassParent	Find	

TFieldType type — DB

Declaration

```
TFieldType = (ftUnknown, ftString, ftSmallint, ftInteger, ftWord, ftBoolean, ftFloat,
  ftCurrency, ftBCD, ftDate, ftTime, ftDateTime, ftBytes, ftVarBytes, ftBlob, ftMemo,
  ftGraphic);
```

The *TFieldType* type is the set of values of the *DataType* property of a *TField* component or *TFieldDef* component.

TFileName type — SysUtils

Declaration

```
TFileName = string[79];
```

The *TFileName* type is the type for the *FileName* property of Open and Save dialog boxes.

TFileRec type — SysUtils

Declaration

```
TFileRec = record
  Handle: Word;
  Mode: Word;
  RecSize: Word;
  Private: array[1..26] of Byte;
  UserData: array[1..16] of Byte;
  Name: array[0..79] of Char;
end;
```

TFileRec is the internal format for typed and untyped files. *TFileRec* enables you to typecast a file variable to access its internal fields.

Note You would normally never declare a variable of this type.

TFileType type — FileCtrl

Declaration

```
TFileAttr = (ftReadonly, ftHidden, ftSystem, ftVolumeID, ftDirectory, ftArchive, ftNormal);
```

```
TFileType = set of TFileAttr;
```

The *TFileType* type is a set of file attributes. The *FileType* property of a file list box (*TFileListBox*) uses the *TFileType* type.

TFillStyle type — Graphics

Declaration

```
TFillStyle = (fsSurface, fsBorder);
```

The *TFillStyle* type determines the method of filling used by the *FloodFill* method of a canvas (*TCanvas* object).

TFilterComboBox component — FileCtrl

The *TFilterComboBox* component is a specialized combo box that is used to present the user with a choice of file filters. Specify the filters you want to appear in the filter combo box with the *Filter* property. The filter the user selects is the value of the *Mask* property.

Most commonly, a filter combo box is used with a file list box (*TFileListBox*). Your application can have the file filter the user selects in the filter combo box determine which files appear in the file list box. If you place this line of code in an *OnChange* event handler of the filter combo box, any change in the filter combo box is reflected in the file list box:

```
   FileListBox1.Mask := FilterComboBox1.Filter;
```

Another way to accomplish the same task is to set the *FileList* property of the filter combo box to the file list box you want affected with a change of filters.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for FilterComboBox component in the online Help, and choose the topic Using the Filter Combo Box Component.

Properties

	Align		Height		ParentFont
▷	BoundsRect		HelpContext	▷ 🔑	SelLength
	Color		Hint	▷ 🔑	SelStart
▷	ComponentIndex	▷ 🔑	ItemIndex	▷ 🔑	SelText
	Ctl3D	▷ 🔑	Items	▷	Showing

Cursor	Left	TabOrder
DragCursor	▷ 🔑 Mask	TabStop
DragMode	Name	▷ 🔑 Text
Enabled	▷ Owner	Tag
🔑 FileList	▷ Parent	Top
🔑 Filter	ParentColor	Visible
Font	ParentCtl3D	Width

Methods

BeginDrag	GetTextBuf	ScrollBy
BringToFront	GetTextLen	🔑 SelectAll
CanFocus	Hide	SendToBack
🔑 Clear	Invalidate	SetBounds
ClientToScreen	Refresh	SetFocus
Dragging	Repaint	SetTextBuf
EndDrag	ScaleBy	Show
Focused	ScreenToClient	Update

Events

OnChange	OnDragOver	OnExit
OnClick	OnDropDown	OnKeyDown
OnDblClick	OnEndDrag	OnKeyPress
OnDragDrop	OnEnter	OnKeyUp

TFindDialog component

Dialogs

The *TFindDialog* component provides a Find dialog box to your application. Users can use the Find dialog box to search for text in a file.

Display the Find dialog box by calling the *Execute* method.

The text your application is searching for is the value of the *FindText* property.

To determine which search options are available in the Find dialog box, use the *Options* property. For example, you can have a Match Case check box appear in the dialog box or hide it, and you can disable or enable the Whole Word check box.

When the user enters the text to search for in the dialog box and chooses Find Next, the *OnFind* event occurs. Within the *OnFind* event handler, write the code that searches for the text specified as the value of *FindText*. Your code should use the *Options* values to determine how the user wants the search conducted.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all components.

For more information, search for FindDialog component in the online Help, and choose the topic Using the Find Dialog Component.

Properties

▷ ComponentIndex	HelpContext	▷ ☞ Position
Ctl3D	Name	Tag
☞ FindText	☞ Options	
▷ Handle	▷ Owner	

Methods

☞ CloseDialog	☞ Execute

Events

☞ OnFind

TFindItemKind type — Menus

Declaration

```
TFindItemKind = (fkCommand, fkHandle, fkShortCut);
```

The *TFindItemKind* defines the possible values of the *Kind* parameter in the *FindItem* method of a menu component.

TFindOptions type — Dialogs

Declaration

```
TFindOption = (frDown, frFindNext, frHideMatchCase, frHideWholeWord, frHideUpDown,
frMatchCase, frDisableMatchCase, frDisableUpDown, frDisableWholeWord, frReplace,
frReplaceAll, frWholeWord, frShowHelp);
```

```
TFindOptions = set of TFindOption;
```

The *TFindOptions* type defines the set of possible values for the *Options* property of the Find and Replace dialog boxes (*TFindDialog* and *TReplaceDialog* components).

TFloatField component

A *TFloatField* represents a field of a record in a dataset. It is represented as a binary value with a range from (positive or negative) $5.0 * 10^{-324}$ to $1.7 * 10^{308}$. It has an accuracy of 15 to 16 digits. Use *TFloatField* for fields that hold floating-point numbers.

Set the *DisplayFormat* property to control the formatting of the field for display purposes, and the *EditFormat* property for editing purposes. Use the *Value* property to access or change the current field value.

The *TFloatField* component has the properties, methods, and events of the *TField* component.

Properties

☞ Alignment	☞ DisplayLabel	☞ MinValue
▷ ☞ AsBoolean	▷ ☞ DisplayName	Name
▷ ☞ AsDateTime	▷ ☞ DisplayText	▷ Owner
▷ ☞ AsFloat	☞ DisplayWidth	☞ Precision
▷ ☞ AsInteger	☞ EditFormat	☞ ReadOnly
▷ ☞ AsString	☞ EditMask	☞ Required
☞ Calculated	▷ ☞ EditMaskPtr	▷ ☞ Size
▷ ☞ CanModify	☞ FieldName	Tag
☞ Currency	▷ ☞ FieldNo	▷ ☞ Text
▷ ☞ DataSet	☞ Index	▷ ☞ Value
▷ ☞ DataSize	▷ ☞ IsIndexField	☞ Visible
▷ ☞ DataType	▷ ☞ IsNull	
☞ DisplayFormat	☞ MaxValue	

Methods

☞ Assign	☞ FocusControl	☞ SetData
☞ AssignValue	☞ GetData	
☞ Clear	☞ IsValidChar	

Events

☞ OnChange	☞ OnSetText	☞ OnValidate
☞ OnGetText		

TFloatFormat

SysUtils

Declaration

```
TFloatFormat = (ffGeneral, ffExponent, ffFixed, ffNumber, ffCurrency);
```

TFloatFormat defines an enumerated list of formatting codes for the float functions.

Format	Defines
ffGeneral	General number format. The value is converted to the shortest possible decimal string using fixed or scientific format. Trailing zeros are removed from the resulting string, and a decimal point appears only if necessary. The resulting string uses fixed point format if the number of digits to the left of the decimal point in the value is less than or equal to the specified precision, and if the value is greater than or equal to 0.00001. Otherwise the resulting string uses scientific format, and the *Digits* parameter specifies the minimum number of digits in the exponent (between 0 and 4).
ffExponent	Scientific format. The value is converted to a string of the form "-d.ddd...E+dddd". The resulting string starts with a minus sign if the number is negative, and one digit always precedes the decimal point. The total number of digits in the resulting string (including the one before the decimal point) is given by the *Precision* parameter. The "E" exponent character in the resulting string is always followed by a plus or minus sign and up to four digits. The Digits parameter specifies the minimum number of digits in the exponent (between 0 and 4).
ffFixed	Fixed point format. The value is converted to a string of the form "-ddd.ddd...". The resulting string starts with a minus sign if the number is negative, and at least one digit always precedes the decimal point. The number of digits after the decimal point is given by the *Digits* parameter—it must be between 0 and 18. If the number of digits to the left of the decimal point is greater than the specified precision, the resulting value will use scientific format.
ffNumber	Number format. The value is converted to a string of the form "-d,ddd,ddd.ddd...". The *ffNumber* format corresponds to the *ffFixed* format, except that the resulting string contains thousand separators.
ffCurrency	Currency format. The value is converted to a string that represents a currency amount. The conversion is controlled by the *CurrencyString*, *CurrencyFormat*, *NegCurrFormat*, *ThousandSeparator*, and *DecimalSeparator* global variables, all of which are initialized from the Currency Format in the International section of the Windows Control Panel. The number of digits after the decimal point is given by the *Digits* parameter—it must be between 0 and 18.

See also

FloatToDecimal procedure, *FloatToStr* function, *FloatToStrF* function, *FloatToText* function, *FloatToTextFmt* function

TFloatRec SysUtils

Declaration

```
TFloatRec = record
  Exponent: Integer;
  Negative: Boolean;
  Digits: array[0..18] of Char;
end;
```

TFloatRec is the *FloatToDecimal* result record.

TFont object Graphics

A *TFont* object defines the appearance of text. *TFont* encapsulates a Windows HFONT.

A *TFont* object defines a set of characters by specifying their height, font family (typeface) name, and so on. The height is specified by the *Height* property. The typeface is specified by the *Name* property. The size in points is specified by the *Size* property. The color is specified by the *Color* property. The attributes of the font (bold, italic, and so on) are specified by the *Style* property.

When a font is modified, an *OnChange* event occurs.

In addition to these properties, methods, and events, this object also has the methods that apply to all objects.

Properties

▷ 🔑 Color	▷ 🔑 Name	▷ 🔑 Size
▷ 🔑 Handle	▷ 🔑 Pitch	▷ 🔑 Style
▷ 🔑 Height	▷ 🔑 PixelsPerInch	

Methods

Assign	Destroy	Free
Create		

Events

🔑 OnChange

TFontDialog component

Dialogs

The *TFontDialog* component makes a Font dialog box available to your application. The purpose of the dialog box is to allow a user to select a font and set attributes of that font. When the user selects a font and chooses OK in the dialog box, the user's font selection is stored in the dialog box's *Font* property, which you can then process as you want.

Display the Font dialog box by calling the *Execute* method.

You choose which device you want a font change to affect with the *Device* property.

You can use the *Options* property to customize how the Font dialog box appears and behaves. For example, you can specify that a Help button be included in the dialog box or that only True Type fonts appear in the list of fonts.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all components.

For more information, search for FontDialog component in the online Help, and choose the topic Using the Font Dialog Component.

Properties

▷	ComponentIndex		HelpContext	🔑	Options
	Ctl3D	🔑	MaxFontSize	▷	Owner
🔑	Device	🔑	MinFontSize		Tag
🔑	Font		Name		

Methods

🔑	Execute

Events

🔑	OnApply

TFontDialogDevice type — Dialogs

Declaration

```
TFontDialogDevice = (fdScreen, fdPrinter, fdBoth);
```

The *TFontDialogDevice* type lists the values the *Device* property of the Font dialog box (*TFontDialog*) can assume.

TFontDialogOptions type — Dialogs

Declaration

```
TFontDialogOption = (fdAnsiOnly, fdTrueTypeOnly, fdEffects, fdFixedPitchOnly,
fdForceFontExist, fdNoFaceSel, fdNoOEMFonts, fdNoSimulations, fdNoSizeSel, fdNoStyleSel,
fdNoVectorFonts, fdShowHelp, fdWysiwyg, fdLimitSize, fdScalableOnly);
```

```
TFontDialogOptions = set of TFontDialogOption;
```

The *TFontDialogOptions* type is the set of values the *Options* property of the Font dialog box (*TFontDialog*) can have.

TFontName type — Graphics

Declaration

```
TFontName = string(LF_FACESIZE - 1);
```

The *TFontName* type is used by the *Name* property of a font object (*TFont*). The maximum number of characters is 32, so font names longer than 32 characters are truncated.

TFontPitch type — Graphics

Declaration

```
TFontPitch = (fpDefault, fpVariable, fpFixed);
```

The *TFontPitch* type is used by the *Pitch* property of a font object (*TFont*).

TFontStyles type — Graphics

Declaration

```
TFontStyle = (fsBold, fsItalic, fsUnderline, fsStrikeOut);

TFontStyles = set of TFontStyle;
```

The *TFontStyles* type is the set of font styles the *Style* property of a font object (*TFont*) can assume.

TForm component — Forms

The Form component is at the center of Delphi applications. You design your application by putting other components on a form. Forms can be used as windows, dialog boxes, or simply as forms, such as data-entry forms.

To display a form that isn't currently active in your application, call either the *Show* or *ShowModal* method. To close a form, call either *Close* or *CloseQuery*, or use the *ModalResult* property with the *ShowModal* method.

You determine the behavior of the horizontal and vertical scroll bars on the form by setting the properties of the *HorzScrollBar* and *VertScrollBar* objects, which are properties of a form.

You can decide how your form first appears—maximized, minimized, or normal—with the *WindowState* property. You can customize the appearance of your form and determine how the user interacts with it by setting the *BorderStyle* and *BorderIcons* properties. Using the *Icon* property, you determine the icon that appears when the form is minimized.

To find out which control is the active control on the form, use the *ActiveControl* property. To assure that a particular control on the form is in view, use the *ScrollInView* method.

Forms have a number of properties and methods that make it simple to create Multiple Document Interface (MDI) applications. You specify which form is the parent form for your application and which forms are the child forms with the *FormStyle* property. Once you have designated a form as a parent and others as children, you can access a child form with the *MDIChildren* property. The number of child forms open in your application is the value of the *MDIChildCount* property. You can determine which form is the active child form with the *ActiveMDIChild* property. For more information about creating MDI applications, search for “MDI applications, creating” in the online Help, and choose the topic “Multiple Document Interface (MDI) Applications.”

Most MDI applications have a Window menu that lists the open child forms or windows at the bottom of the menu. You can specify which item on the main menu of your application is the Window menu as the value of the *WindowMenu* property, and at run time, the open child forms are automatically listed at the bottom of the specified menu. Usually, Window menus have commands that allow the user to manage the windows or forms in the running application. You can call the *Cascade*, *Tile*, *Previous*, *Next*, and *ArrangeIcons* methods in the *OnClick* event handlers for the appropriate menu commands, which make it very easy to give your users this capability.

If you want your form to display different menus at various times while your application runs, you specify the menu you want to use with the *Menu* property. If you want your application to be able to process key events rather than have them go immediately to the selected control on the form, set the form’s *KeyPreview* property to *True*.

You can use the *OnCreate* event handler of the form to set initial values for properties and do any processing you want to occur before the user begins interacting with the form.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for Form component in the online Help, and choose the topic Using the Form Component.

Properties

▷ 🔑 Active	▷ Controls	▷ Owner
🔑 ActiveControl	Ctl3D	▷ Parent
▷ 🔑 ActiveMDIChild	Cursor	🔑 PixelsPerInch
▷ Align	Enabled	PopupMenu
🔑 AutoScroll	Font	🔑 Position
🔑 BorderIcons	🔑 FormStyle	🔑 PrintScale
🔑 BorderStyle	▷ Handle	🔑 Scaled
▷ Brush	Height	ShowHint
Caption	HelpContext	▷ Showing
▷ Canvas	Hint	TabOrder
▷ ClientHandle	🔑 HorzScrollBar	TabStop
ClientHeight	🔑 Icon	Tag
▷ ClientOrigin	🔑 KeyPreview	▷ 🔑 TileMode
▷ ClientRect	Left	Top
ClientWidth	▷ 🔑 MDIChildCount	🔑 VertScrollBar
Color	▷ 🔑 MDIChildren	Visible
▷ ComponentCount	🔑 Menu	Width
▷ ComponentIndex	▷ 🔑 ModalResult	🔑 WindowMenu

▷	Components		Name	🔑	WindowState
▷	ControlCount	🔑	ObjectMenuItem		

Methods

🔑	ArrangeIcons	🔑	GetFormImage		RemoveComponent
	BringToFront		GetTextBuf		Repaint
	CanFocus		GetTextLen		ScaleBy
🔑	Cascade		Hide		ScreenToClient
	ClientToScreen		HandleAllocated		ScrollBy
	Close		HandleNeeded	🔑	ScrollInView
🔑	CloseQuery		Hide		SendToBack
	ContainsControl		InsertComponent		SetBounds
	Create		InsertControl		SetFocus
🔑	CreateNew		Invalidate		SetTextBuf
	Destroy		Next		Show
	Dragging		Previous	🔑	ShowModal
	FindComponent		Print	🔑	Tile
	Focused		Refresh		Update
	Free	🔑	Release		

Events

🔑	OnActivate		OnDragDrop		OnMouseDown
	OnClick		OnDragOver		OnMouseMove
🔑	OnClose		OnEnter		OnMouseUp
🔑	OnCloseQuery		OnExit	🔑	OnPaint
	OnCreate		OnHide	🔑	OnResize
🔑	OnDestroy		OnKeyDown	🔑	OnShow
	OnDblClick		OnKeyPress		
🔑	OnDeactivate		OnKeyUp		

TFormBorderStyle type — Forms

Declaration

```
TFormBorderStyle = (bsNone, bsSingle, bsSizeable, bsDialog);
```

The *TFormBorderStyle* type defines the possible border styles of a form. It is the type of the form’s *BorderStyle* property.

TFormStyle type

Forms

Declaration

```
TFormStyle = (fsNormal, fsMDIChild, fsMDIForm, fsStayOnTop);
```

The *TFormStyle* type defines the possible values of the *FormStyle* property of a form (*TForm*).

TGetEditEvent type

Grids

Declaration

```
TGetEditEvent = procedure (Sender: TObject; ACol, ARow: Longint; var Value: string) of object;
```

The *TGetEditEvent* points to a method that handles the retrieving of the text displayed in a cell in a draw grid (*TDrawGrid*) or string grid (*TStringGrid*) while the grid is in Edit mode, or the edit mask used to display text. The *ACol* parameter specifies the column of the cell, and the *ARow* parameter specifies the row of the cell. The *Value* parameter is the string displayed in the cell or the edit mask used to display the text.

TGetEditEvent is the type of the *OnGetEditText* and *OnGetEditMask* events of the draw and string grid components.

TGraphic object

Graphics

The *TGraphic* object is the foundation class for the *TBitmap*, *TIcon*, and *TMetafile* objects. If you know which type of graphic (bitmap, icon, or metafile) you will be using, you should store the graphic in its specific type object (*TBitmap*, *TIcon*, or *TMetafile*, respectively). Otherwise, you should use a *TPicture* object which can hold any type of *TGraphic*.

In addition to these properties, methods, and events, this object also has the methods that apply to all objects.

Properties

▷ 🔑 Height	▷ 🔑 Empty	▷ 🔑 Width

Methods

ClassName	Create	🔑 LoadFromFile
ClassParent	Destroy	🔑 SaveToFile
ClassType	Free	

Events

🔑 OnChange

TGraphicField component

A *TGraphicField* represents a field of a record which is represented by a value consisting of an arbitrary set of bytes with indefinite size. The bytes should correspond to graphics data.

Use the *Assign* method to transfer another component to a *TGraphicField*. Use the *LoadFromFile* method to load a field's contents from a file. Use *LoadFromStream* method to load a field from a *Stream*. Use *SaveToFile* method to write a field's contents to a file. Use *SaveToStream* method to write a field's contents to a *Stream*.

The *TGraphicField* component has the properties, methods, and events of the *TField* component.

Properties

☞ Alignment	▷ ☞ DataType	▷ ☞ IsIndexField
▷ ☞ AsBoolean	☞ DisplayLabel	▷ ☞ IsNull
▷ ☞ AsDateTime	▷ ☞ DisplayName	Name
▷ ☞ AsFloat	▷ ☞ DisplayText	▷ Owner
▷ ☞ AsInteger	☞ DisplayWidth	☞ ReadOnly
▷ ☞ AsString	☞ EditMask	☞ Required
☞ Calculated	▷ ☞ EditMaskPtr	☞ Size
▷ ☞ CanModify	☞ FieldName	Tag
▷ ☞ DataSet	▷ ☞ FieldNo	▷ ☞ Text
▷ ☞ DataSize	☞ Index	☞ Visible

Methods

☞ Assign	☞ GetData	☞ SaveToFile
☞ AssignValue	☞ IsValidChar	☞ SaveToStream
☞ Clear	☞ LoadFromFile	☞ SetData
☞ FocusControl	☞ LoadFromStream	

Events

☞ OnChange	☞ OnSetText	☞ OnValidate
☞ OnGetText		

TGraphicsObject object

Graphics

A *TGraphicsObject* object is the base class for the Delphi encapsulation of the three main Windows graphics tools: the *TBrush*, *TFont*, and *TPen* objects.

In addition to these methods and events, this object also has the methods that apply to all objects.

Methods

ClassName	ClassType	Destroy
ClassParent	Create	Free

Events

🔑 OnChange

TGridDrawState type Grids

Declaration

```
TGridDrawState = set of (gdSelected, gdFocused, gdFixed);
```

The *TGridDrawState* type defines the possible states of cell when drawing occurs. The *TGridDrawState* is the type of the *AState* parameter used in the *TDrawCellEvent* method pointer.

TGridOptions type Grids

Declaration

```
TGridOption = (goFixedHorzLine, goFixedVertLine, goHorzLine, goVertLine, goRangeSelect,
goDrawFocusSelected, goRowSizing, goColSizing, goRowMoving, goColMoving, goEditing, goTabs,
goRowSelect, goAlwaysShowEditor, goThumbTracking);
```

```
TGridOptions = set of TGridOption;
```

TGridOptions is the set of values the *Options* property of a *TDrawGrid* or *TStringGrid* component can have.

TGridRect type Grids

Declaration

```
TGridRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Longint);
    1: (TopLeft, BottomRight: TGridCoord);
end;
```

TGridRect defines a rectangular area within a grid control. It is the type of the *Selection* property of the *TDrawGrid* and *TStringGrid* components.

TGroupBox component

StdCtrls

The *TGroupBox* component is a standard Windows group box. Use a group box component to group related controls on a form. The most commonly grouped controls in a group box are radio buttons (*TRadioButton*).

Place the group box on the form, then select the components you want to appear in the group box from the Component palette, and place them in the group box.

The text that identifies the purpose of the grouping appears as the value of the *Caption* property.

Once you place another windowed control within a group box, the group box becomes the parent of the control and is the value of that control's *Parent* property.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for GroupBox component in the online Help, and choose the topic Using the Group Box Component.

Properties

Align	Height	PopupMenu
🔑 Caption	HelpContext	▷ Showing
Color	Hint	TabOrder
▷ Controls	Left	TabStop
Ctl3D	Name	Tag
Cursor	▷ Owner	Top
DragCursor	▷ Parent	Visible
DragMode	ParentColor	Width
Enabled	ParentCtl3D	
Font	ParentFont	

Methods

BeginDrag	Focused	ScaleBy
BringToFront	GetTextBuf	ScreenToClient
CanFocus	GetTextLen	ScrollBy
ClientToScreen	Hide	SendToBack
🔑 ContainsControl	Invalidate	SetBounds
Dragging	Refresh	SetFocus
EndDrag	Repaint	SetTextBuf

Events

OnClick	OnEndDrag	OnMouseMove
OnDblClick	OnEnter	OnMouseUp

OnDragDrop	OnExit
OnDragOver	OnMouseDown

THeader component **ExtCtrls**

The *THeader* component is a sectioned visual control that displays text and allows each section to be resized with the mouse. At design time, resize a section by clicking the right mouse button on a section border and dragging to the new size. At run time, the user can resize the header by clicking and dragging with the left mouse button. The widths of the other sections that are not resized remain unchanged.

The *Sections* property specifies the sections of a header. The *AllowResize* property enables or prevents the user from resizing sections at run time. When a section is resized, an *OnSizing* event occurs. After a section has been resized, an *OnSized* event occurs.

To use a header you should attach code to these event handlers. One use would be to align text under a header. When the header is resized, you would realign the text in the *OnSized* event handler. To move the text as the header is being resized, realign the text in the *OnSizing* event handler.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for GroupBox component in the online Help, and choose the topic Using the Group Box Component.

Properties

	Align		HelpContext	▷ ☞	SectionWidth
	AllowResize		Hint		ShowHint
▷	BoundsRect		Left	▷	Showing
	BorderStyle		Name		TabOrder
▷	ComponentIndex	▷	Owner		TabStop
	Cursor	▷	Parent		Tag
	Enabled		ParentFont		Top
	Font		ParentShowHint		Visible
	Height	☞	Sections		Width

Methods

BeginDrag	GetTextLen	SendToBack
BringToFront	Hide	SetBounds
CanFocus	Invalidate	SetFocus
ClientToScreen	Refresh	SetTextBuf
Dragging	Repaint	Show
EndDrag	ScaleBy	Update

Focused	ScrollBy
GetTextBuf	ScreenToClient

Events

OnEnter	☞ OnSized	☞ OnSizing
OnExit		

THelpContext type — Classes

Declaration

```
THelpContext = -MaxLongInt..MaxLongInt;
```

The *THelpContext* type is used to define Help context numbers.

THelpEvent type — Classes

Declaration

```
THelpEvent = function (Command: Word; Data: Longint): Boolean of object;
```

The *THelpEvent* is used by the *OnHelp* event handler. To find the possible values of the *Command* and *Data* parameters, search for the WinHelp topic in the Help system, which explains the WinHelp API (application programming interface).

THintInfo type — Forms

Declaration

```
THintInfo = record
  HintControl: TControl;
  HintPos: TPoint;
  HintMaxWidth: Integer;
  HintColor: TColor;
  CursorRect: TRect;
  CursorPos: TPoint;
end;
```

The *THintInfo* type is used to define the appearance and behavior of the Help window in a *TShowHintEvent* type *OnShowHint* event handler.

TIcon object — Graphics

A *TIcon* object contains an icon graphic (.ICO file format). *TIcon* encapsulates a Windows HICON.

The height and width in pixels of the icon are specified by the *Height* and *Width* properties, respectively.

To load an icon from a file, call the *LoadFromFile* method. To save an icon to a file, call *SaveToFile*.

To draw an icon on a canvas, call the *Draw* or *StretchDraw* methods of a *TCanvas* object, passing a *TIcon* as a parameter.

When the icon is modified, an *OnChange* event occurs.

In addition to these properties, methods, and events, this object also has the methods that apply to all objects.

Properties

▷ 🔑 Empty	▷ 🔑 Height	▷ 🔑 Width
▷ 🔑 Handle		

Methods

Assign	ClassType	Free
ClassName	Create	🔑 LoadFromFile
ClassParent	Destroy	🔑 SaveToFile

Events

🔑 OnChange

TIdleEvent type **Forms**

Declaration

```
TIdleEvent = procedure (Sender: TObject; var Done: Boolean) of object;
```

The *TIdleEvent* type points to a method that runs when your application is idle. It is the type of the *OnIdle* event of the application (*TApplication*).

The *Boolean* parameter *Done* is *True* by default. When *Done* is *True*, the Windows API *WaitMessage* function is called when *OnIdle* returns. *WaitMessage* yields control to other applications until a new message appears in the message queue of your application. If *Done* is *False*, *WaitMessage* is not called.

Tile method

Applies to

TForm component

Declaration

```
procedure Tile;
```

The *Tile* method arranges the child forms of a parent form in your application so that the forms are all the same size. At the same time, all the forms together completely fill up the client area of the parent form. How the forms arrange themselves depends upon the value of the *TileMode* property.

The *Tile* method applies only to forms that are MDI parent forms (have a *FormStyle* property value of *fsMDIForm*).

Example

This example uses three forms. The first form has its *FormStyle* property set to *MDIForm*. The other two have their *FormStyle* properties set to *MDIChild* and their *Visible* properties set to *True*. Add a main menu component and name one of the menu items *TileForms*. This is code for the *TileFormsClick* handler:

```
procedure TForm1.TileForms1Click(Sender: TObject);
begin
  TileMode := tbVertical;
  Tile;
end;
```

When the user chooses the TileForms command, the child forms tile vertically within the MDI frame form.

See also

ArrangeIcons method, *Cascade* method, *Next* method, *Previous* method

TileMode property

Applies to

TForm component

Declaration

```
property TileMode: TTileMode;
```

Run-time only. The *TileMode* property determines how the child forms within a parent form arrange themselves when the application calls the *Tile* method. These are the possible values the *TileMode* property can have:

Value	Meaning
tbHorizontal	Each form stretches across the width of the parent form
tbVertical	Each form stretches along the height of the parent form

Setting the *TileMode* property is meaningful only in an MDI parent form (has a *FormStyle* property value of *fsMDIForm*).

Example

This example uses three forms. The first form has its *FormStyle* property set to *MDIForm*. The other two have their *FormStyle* properties set to *MDIChild* and their *Visible* properties set to *True*. Add a main menu component and name one of the menu items *TileForms*. This is code for the *TileFormsClick* handler:

```
procedure TForm1.TileForms1Click(Sender: TObject);
begin
  TileMode := tbHorizontal;
  Tile;
end;
```

When the application runs and the user chooses the TileForms command, the child forms tile horizontally within the MDI frame form.

See also

Tile method

TImage component — ExtCtrls

The *TImage* component displays a graphical image on a form. The image that appears is the value of the *Picture* property. If you want the image control to resize to fit the current image, set the *AutoSize* property to *True*. If you want to resize the image to completely fill an image control when the control is larger than the native size of the image, use the *Stretch* property.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all controls.

For more information, search for Image component in the online Help, and choose the topic Using the Image Component.

Properties

	Align		Height		ShowHint
🔑	Autosize		HelpContext	▷	Showing
▷	BoundsRect		Hint	🔑	Stretch
▷	Canvas		Left		TabOrder
🔑	Center		Name		TabStop
▷	ComponentIndex	▷	Owner		Tag
	Cursor	▷	Parent		Top
	DragCursor		ParentShowHint		Visible
	DragMode	🔑	Picture		Width
	Enabled		PopupMenu		

Methods

BeginDrag	Hide	SendToBack
BringToFront	Invalidate	SetBounds
ClientToScreen	Refresh	Show
Dragging	Repaint	Update
EndDrag	ScaleBy	
Focused	ScreenToClient	

Events

OnClick	OnDragOver	OnMouseMove
OnDblClick	OnEndDrag	OnMouseUp
OnDragDrop	OnMouseDown	

Time function — SysUtils

Declaration

```
function Time: TDateTime;
```

The *Time* function returns the current time.

Example

This example uses a label and a button on a form. When the user clicks the button, the current time displays in the caption of the label:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := 'The time is  ' + TimeToStr(Time);
end;
```

See also

Date function, *DecodeTime* procedure, *Now* function, *TimeToStr* function

TimeFormat property

Applies to

TMediaPlayer component

Declaration

```
property TimeFormat: TMPTimeFormats;
```

Run-time only. The *TimeFormat* property determines the format used to specify position information.

TimeFormat determines how the *StartPos*, *Length*, *Position*, *Start*, and *EndPos* properties are interpreted. For example, if *Position* is 180 and *TimeFormat* is *tfMilliseconds*, the current position is 180 milliseconds into the medium. If *Position* is 180 and *TimeFormat* is *tfMSF*, the current position is 180 minutes into the medium.

Not all formats are supported by every device. If you try to set an unsupported format, the assignment is ignored.

The current timing information is always passed in a 4-byte integer. In some formats, the timing information returned is not really one integer, but single bytes of information packed in the long integer.

The following table lists the possible values for the *TimeFormat* property:

Value	Time format
tfMilliseconds	Milliseconds are stored as a 4-byte integer variable.
tfHMS	Hours, minutes, and seconds packed into a 4-byte integer. From least significant to most significant byte, the data values are Hours (least significant byte) Minutes Seconds Unused (most significant byte)
tfMSF	Minutes, seconds, and frames packed into a 4-byte integer. From least significant to most significant byte, the data values are Minutes (least significant byte) Seconds Frames Unused (most significant byte)
tfFrames	Frames are stored as a 4-byte integer variable.
tfSMPTE24	24-frame SMPTE packs values in a 4-byte variable. From least significant to most significant byte, the data values are Hours (least significant byte) Minutes Seconds Frames (most significant byte) SMPTE (Society of Motion Picture and Television Engineers) time is an absolute time format expressed in hours, minutes, seconds, and frames. The standard SMPTE division types are 24, 25, and 30 frames per second.
tfSMPTE25	25-frame SMPTE packs data into a 4-byte variable in the same order as 24-frame SMPTE.
tfSMPTE30	30-frame SMPTE packs data into the 4-byte variable in the same order as 24-frame SMPTE.
tfSMPTE30Drop	30-drop-frame SMPTE packs data into the 4-byte variable in the same order as 24-frame SMPTE.
tfBytes	Bytes are stored as a 4-byte integer variable.
tfSamples	Samples are stored as a 4-byte integer variable.
tfTMSF	Tracks, minutes, seconds, and frames are packed in the 4-byte variable. From least significant to most significant byte, the data values are Tracks (least significant byte) Minutes Seconds Frames (most significant byte) Note that MCI uses continuous track numbering.

Note Functions provided with MCI to help you decode the 4-byte integer specified in a given time format are documented under *MCI Macros for Encoding and Decoding Time Data* in the MMSYSTEM.HLP Help file.

Example

The following code declares a *HMSRec* record with four byte fields. If *TimeFormat* is *tfHMS*, the first field specifies hours, the second field specifies minutes, the third field specifies seconds, and the fourth field corresponds to the unused most-significant byte of the *tfHMS* time format. A *LongInt* variable is typecast to an *HMSRec* record, then the hours, minutes, and seconds of the *Length* of the loaded media are displayed in labels when the user clicks a button.

```
type
  HMSRec = record
    Hours: byte;
    Minutes: byte;
    Seconds: byte;
    NotUsed: byte;
  end;

procedure TForm1.Button1Click(Sender: TObject);
var
  TheLength: LongInt;
begin
  TimeFormat := tfHMS; { Set time format - note that some devices don't support tfHMS }
  TheLength := MediaPlayer1.Length; { Store length of currently loaded media in var }
  with HMSRec(TheLength) do { Typecast TheLength as a HMSRec record }
  begin
    Label1.Caption := IntToStr(Hours);   { Display Hours in Label1 }
    Label2.Caption := IntToStr(Minutes); { Display Minutes in Label2 }
    Label3.Caption := IntToStr(Seconds); { Display Seconds in Label3 }
  end;
end;
```

TimeToStr function SysUtils

Declaration

```
function TimeToStr(Time: TDateTime): string;
```

The *TimeToStr* function converts the *Time* parameter, a variable of type *TDateTime*, to a string. You can change the format of how the string is displayed by changing the values of some of the date and time variables.

Example

This example uses a label and a button on a form. When the user clicks the button, the current time appears as the caption of the label:

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

```
    Label1.Caption := TimeToStr(Time);
  end;
```

See also
DateTimeToStr function, *DateToStr* function, *StrToDateTime* function, *Time* function

TIndexDef object

The *TIndexDef* object describes the index for a table.

Use the *Fields* property to get a list of the fields in the index Use the *Name* property to get the name of the index. Test the flags in the *Options* property for a specific characteristic of the index.

In addition to these properties and methods, this object also has the methods that apply to all objects.

Properties

▷ Expression	▷ Name	▷ Options
▷ Fields		

Methods

ClassName	ClassType	Destroy
ClassParent	Create	Free

TIndexDefs object

The *TIndexDefs* object holds the set of available indexes for a table.

In addition to these properties and methods, this object also has the methods that apply to all objects.

Properties

▷ Count	▷ Items

Methods

Add	ClassType	FindIndexForFields
Assign	Clear	Free
ClassName	Create	IndexOf
ClassParent	Destroy	Update

TIndexOptions type DB

Declaration

```
TIndexOptions = set of (ixPrimary, ixUnique, ixDescending, ixNonMaintained,
  ixCaseInsensitive);
```

The *TIndexOptions* type is the set of values that can be used in creating a new index. It is used by the *AddIndex* method of a dataset component.

TIniFile object IniFiles

The *TIniFile* object permits your application to write and read an .INI file.

Your application can retrieve all the strings in a section of an .INI file by calling the *ReadSection* method; or it can retrieve a single *Boolean*, integer, or string value by calling the *ReadBool*, *ReadInteger*, or *ReadString* methods.

To erase an entire section of an .INI file, use the *EraseSection* method.

Your application can change the settings in an existing .INI file. To change a *Boolean* value, call the *WriteBool* method. To change an integer value, call the *WriteInteger* method. Finally, to change a string value, call the *WriteString* method.

In addition to these methods, this object also has the methods that apply to all objects.

Methods

ClassName	🔑 FileName	🔑 ReadString
ClassParent	Free	🔑 WriteBool
ClassType	🔑 ReadBool	🔑 WriteInteger
Create	🔑 ReadInteger	🔑 WriteString
Destroy	🔑 ReadSection	
🔑 EraseSection	🔑 ReadSectionValues	

TIntegerField component

A *TIntegerField* component represents a field of a record in a dataset. It is represented as a binary value with a range from -2,147,483,648 to 2,147,483,647. Use *TIntegerField* for fields that hold large, signed whole numbers.

Set the *DisplayFormat* property to control the formatting of the field for display purposes, and the *EditFormat* property for editing purposes. Use the *Value* property to access or change the current field value. Set the *MinValue* or the *MaxValue* property to limit the smallest or largest value permitted in a field.

The *TIntegerField* component has the properties, methods, and events of the *TField* component.

Properties

🔑 Alignment	🔑 DisplayFormat	▷ 🔑 IsNull
▷ 🔑 AsBoolean	🔑 DisplayLabel	🔑 MaxValue
▷ 🔑 AsDateTime	▷ 🔑 DisplayName	🔑 MinValue
▷ 🔑 AsFloat	▷ 🔑 DisplayText	Name
▷ 🔑 AsInteger	🔑 DisplayWidth	▷ Owner
▷ 🔑 AsString	🔑 EditFormat	🔑 ReadOnly
🔑 Calculated	🔑 EditMask	🔑 Required
▷ 🔑 CanModify	▷ 🔑 EditMaskPtr	▷ 🔑 Size
▷ 🔑 DataSet	🔑 FieldName	Tag
▷ 🔑 DataSize	▷ 🔑 FieldNo	▷ 🔑 Text
▷ 🔑 DataType	🔑 Index	▷ 🔑 Value
▷ 🔑 AsFloat	▷ 🔑 IsIndexField	🔑 Visible

Methods

🔑 Assign	🔑 FocusControl	🔑 SetData
🔑 AssignValue	🔑 GetData	
🔑 Clear	🔑 IsValidChar	

Events

🔑 OnChange	🔑 OnSetText	🔑 OnValidate
🔑 OnGetText		

Title property

Applies to

TPrinter object; *TApplication*, *TOpenDialog*, *TSaveDialog* components

The *Title* property specifies the text used to title an object, component, or application.

For applications

Declaration

```
property Title: string;
```

The *Title* property determines the text that appears with an icon representing your application when it is minimized. The default value is the project name (the name of the project file without the .PRJ file extension).

You can set the title at run time, or you can enter the value of the *Title* property on the Application page of the Options | Project Options dialog box.

Example

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.Title := 'My Incredible Application';
end;
```

See also

Application variable, *Caption* property, *Icon* property, *Minimize* method

For Open and Save dialog boxes

Declaration

```
property Title: string;
```

The *Title* property determines the text that appears in the dialog box's title bar.

Example

This code displays the Open dialog box with the text "Open Pascal files" in its title bar and lists only Pascal files in the list box:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  OpenDialog1.Filter := 'Pascal files (*.PAS)|*.PAS';
  OpenDialog1.Title := 'Open Pascal files';
  OpenDialog1.Execute;
end;
```

For printer objects

```
property Title: string;
```

Run-time only. The *Title* property determines the text that appears listed in the Print Manager and on network header pages.

Example

This line of code sets the value of the *Title* property for the printer object:

```
Printer.Title := 'My incredible application';
```

See also

Printer variable

TitleFont property

Applies to
TDBGrid component

Declaration

```
property TitleFont: TFont;
```

The *TitleFont* property determines the font used for the titles of the columns in the data grid.

Example
The following code makes the font specified by the font dialog component,*FontDialog1*, the font of the data grid.

```
if FontDialog1.Execute then
  DBGrid1.TitleFont := FontDialog1.Font;
```

See also
Title property

TKey type **Controls**

Declaration

```
TKey = Word;
```

The *TKey* type is used to hold keyboard scan codes in keyboard event handlers and in menu shortcut routines.

See also
OnKeyDown event, *OnKeyUp* event, *ShortCut* function, *ShortCutToKey* procedure, *TKeyEvent* type

TKeyEvent type **Controls**

Declaration

```
TKeyEvent = procedure (Sender: TObject; var Key: Word; Shift: TShiftState) of object;
```

The *TKeyEvent* type points to a method that handles keyboard events. The*Key* parameter is the key on the keyboard and *Shift* is one of these possible states:

State	Meaning
ssShift	The *Shift* key is held down.
ssAlt	The *Alt* key is held down.

State	Meaning
ssCtrl	The *Ctrl* key is held down.
ssLeft	The left mouse button is held down.
ssMiddle	The middle mouse button is held down.
ssDouble	Both the right and left mouse buttons are held down.

TKeyEvent is the type of the *OnKeyDown* and *OnKeyUp* events.

TKeyPressEvent type — Controls

Declaration

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

The *TKeyPressEvent* type points to a method that handles a single character key press. The *Key* parameter is the key on the keyboard.

TKeyPressEvent is the type of the *OnKeyPress* event.

TLabel component — StdCtrls

The *TLabel* component is a nonwindowed control that displays text on a form. Usually this text labels some other control.

The text of a label is the value of its *Caption* property. Within the caption, you can include an accelerator key. When the user presses the accelerator key, the control that is the value of the label's *FocusControl* becomes the active control on the form.

How the text of the caption aligns within the label is determined by the value of the *Alignment* property. You can have the label resize automatically to fit a changing caption if you set the *AutoSize* property to *True*. If you prefer to have the text wrap, set *WordWrap* to *True*.

If you want a label to appear on top of a graphic, but you want to be able to see through the label so that part of the graphic isn't hidden, set the *Transparent* property to *True*.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all controls.

For more information, search for Label component in the online Help, and choose the topic Using the Label Component.

Properties

	Align	🔑	FocusControl		PopupMenu
🔑	Alignment		Font	🔑	ShowAccelChar
🔑	AutoSize		Height		ShowHint
▷	BoundsRect		Hint		Tag

Caption	Left	Top
Color	Name	☞ Transparent
▷ ComponentIndex	☞ Owner	Visible
Cursor	☞ Parent	Width
DragCursor	ParentColor	☞ WordWrap
DragMode	ParentFont	
Enabled	ParentShowHint	

Methods

BeginDrag	GetTextLen	ScreenToClient
BringToFront	Hide	SendToBack
ClientToScreen	Invalidate	SetBounds
Dragging	Refresh	SetTextBuf
EndDrag	Repaint	Show
GetTextBuf	ScaleBy	Update

Events

OnClick	OnDragOver	OnMouseMove
OnDblClick	OnEndDrag	OnMouseUp
OnDragDrop	OnMouseDown	

TLeftRight type — Classes

Declaration

```
TAlignment = (taLeftJustify, taRightJustify, taCenter);
```

```
TLeftRight = taLeftJustify..taRightJustify;
```

TLeftRight is the type of the *Alignment* property of check boxes and radio buttons.

TList object — Classes

The *TList* object is used to maintain lists of objects.

The *List* property is a list of pointers to all the objects in the list. You can access a particular item referenced in the list using the *Items* property. To find the position of an item in the list, use the *IndexOf* method.

You can add, delete, insert, remove, move, and exchange items in the list using the *Add*, *Delete*, *Insert*, *Remove*, *Move*, and *Exchange* methods. Use the *Count* property to determine how many items are in the list.

Use the *First* method to move to the beginning of the list, and use the *Last* method to move to the end of the list.

The number of items the list can maintain is determined by the value of the *Capacity* property. If you need to increase the size of the list, call the *Expand* method. You can remove all **nil** pointers in the list with the *Pack* method.

In addition to these properties and methods, this object also has the methods that apply to all objects.

Properties

▷ 🔑 Capacity	▷ 🔑 Items	▷ 🔑 List
▷ 🔑 Count		

Methods

🔑 Add	🔑 Delete	🔑 IndexOf
ClassName	Destroy	🔑 Insert
ClassParent	🔑 Exchange	🔑 Last
ClassType	🔑 Expand	🔑 Pack
Create	🔑 First	🔑 Remove
🔑 Clear	Free	

TListBox component StdCtrls

The *TListBox* component is a Windows list box. A list box displays a list from which users can select one or more items.

The list of items in the list box is the value of the *Items* property. The *ItemIndex* property indicates which item in the list box is selected.

You can add, delete, and insert items in the list box using the *Add*, *Delete*, and *Insert* methods of the *Items* object, which is of type *TStrings*. For example, to add a string to a list box, you could write this line of code:

```
ListBox1.Items.Add('New item');
```

You can change how the list box appears. If you want the list box to have multiple columns, change the value of the *Columns* property. Sort the list box items with the *Sorted* property.

You can allow users to select more than one item at a time by setting the *MultiSelect* property to *True*. The *ExtendedSelect* property determines how multiple items can be selected. To determine whether a particular item is selected and how many items are selected, check the values of the *Selected* and *SelCount* properties, respectively.

You can make the list box an owner-draw list box by changing the *Style* property.

You can drag and drop objects into a list box. For more information, search for Dragging and Dropping in the Help file.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

Properties

Align	*Hint*	▷ 🔑 *SelCount*
BorderStyle	🔑 *IntegralHeight*	▷ 🔑 *Selected*
▷ *Canvas*	▷ 🔑 *ItemIndex*	*ShowHint*
Color	🔑 *ItemHeight*	*Showing*
🔑 *Columns*	🔑 *Items*	🔑 *Sorted*
▷ *ComponentIndex*	*Left*	🔑 *Style*
Ctl3D	🔑 *MultiSelect*	*TabOrder*
Cursor	*Name*	*TabStop*
DragCursor	▷ *Owner*	*Tag*
DragMode	*Parent*	*Top*
Enabled	*ParentColor*	▷ 🔑 *TopIndex*
ExtendedSelect	*ParentCtl3D*	*Visible*
Font	*ParentFont*	*Width*
Height	*ParentShowHint*	
HelpContext	*PopupMenu*	

Methods

BeginDrag	Hide	SendToBack
BringToFront	🔑 ItemAtPos	SetBounds
🔑 Clear	Invalidate	SetFocus
ClientToScreen	Refresh	SetTextBuf
Dragging	Repaint	Show
EndDrag	ScaleBy	Update
GetTextBuf	ScreenToClient	
GetTextLen	ScrollBy	

Events

OnClick	OnEndDrag	OnKeyUp
OnDblClick	OnEnter	🔑 OnMeasureItem
OnDragDrop	OnExit	OnMouseDown
OnDragOver	OnKeyDown	OnMouseMove
🔑 OnDrawItem	OnKeyPress	OnMouseUp

See also

Creating an owner-draw control, *TComboBox* component, *TDBListBox* component

T

TListBoxStyle type

StdCtrls

Declaration

```
TListBoxStyle = (lbStandard, lbOwnerDrawFixed, lbOwnerDrawVariable);
```

The *TListBoxStyle* type is the type of the *Style* property for a list box (*TListBox* component).

TLocale type

DB

Declaration

```
TLocale = Pointer;
```

The *TLocale* type is the type of a *Locale* or *DBLocale* property. These properties are only used or needed when making direct calls to the Borland Database Engine.

TLoginEvent type

DB

Declaration

```
TLoginEvent = procedure(Database: TDatabase; LoginParams: TStrings) of object;
```

The *TLoginEvent* type is the header for the method that handles an *OnLogin* event for a *TDatabase*. The *Database* parameter is the database. *LoginParams* is a *TStrings* object which holds the username and password, along with any other parameters to be used in opening the *Database*. The username is a string of the form 'USER NAME=John_Doe'. The password is a string of the form 'PASSWORD=His_Password'. The *OnLogin* event handler should add both the username and password to *LoginParams* when called.

TMacroEvent type

DDEMan

Declaration

```
TMacroEvent = procedure(Sender: TObject; Msg : String) of object;
```

The *TMacroEvent* type points to a method that handles the passing of a macro string from a DDE client to a DDE server conversation (*TDDEServerConv*) component. *Msg* contains the macro.

TMacroEvent is the type of the *OnExecuteMacro* event.

TMainMenu component

Menus

The *MainMenu* component encapsulates a menu bar and its accompanying drop-down menus for a form. To begin designing a menu, add a main menu component to your form, and double-click the component. See the topic Menu Designer in the Help system.

The items on the menu bar and in its drop-down menus are specified with the *Items* object, a property of a main menu. The *Items* object is of type *TMenuItem*. Your application can use the *Items* property to access a particular command on the menu.

You can choose to have the menus of one form merge with those of another using the *AutoMerge* property and the *Merge* and *Unmerge* methods.

In addition to these properties and methods, this component also has the properties and methods that apply to all components.

For more information, search for MainMenu component in the online Help, and choose the topic Using the Main Menu Component.

Properties

☞ AutoMerge	☞ Items	▷ Owner
▷ ComponentIndex	Name	Tag

Methods

☞ FindItem	☞ GetHelpContext	☞ Unmerge
Free	☞ Merge	

See also

ShortCut function, *ShortCutToKey* procedure, *ShortCutToText* function, *TextToShortCut* function, *TPopupMenu* component

TMaskEdit component

Mask

A mask edit box is an much like an ordinary edit box (*TEdit* component), except you can require the user to enter only valid characters through the use of an *EditMask* property. You can also use the mask to format the display of data.

The text the user enters in the edit box is the value of the *Text* property, just as it is with any edit box. The text of the edit box with the mask specified in the *EditMask* property applied to it is the value of the *EditText* property.

Your application can tell if the value of *Text* changes by checking the value of the *Modified* property. To limit the number of characters users can enter into the mask edit box, use the *MaxLength* property.

If you want to prevent the user from changing the value of the *Text* property by typing in the edit box, set the *ReadOnly* property to *True*.

You can choose to have the text in a mask edit box automatically selected whenever it becomes the active control with the *AutoSelect* property. At run time, you can select all the text in the edit box with the *SelectAll* method. To find out which text in the edit box the user has selected or to replace selected text, use the *SelText* property. To clear selected text, call the *ClearSelection* method. To select only part of the text or to find out what part of the text is selected, use the *SelStart* and *SelLength* properties.

You can cut, copy, and paste text to and from a mask edit box using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods.

Your application can display a specified character rather than the actual character typed into an edit box. If the edit box is used to enter a password, onlookers won't be able to read the typed text. Specify the special character with the *PasswordChar* property.

If you want the edit box to automatically resize to accommodate a change in font size, use the *AutoSize* property.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for MaskEdit component in the online Help, and choose the topic Using the MaskEdit Component.

Properties

▷	Align		Height	🔑	ReadOnly
🔑	AutoSelect		HelpContext	▷ 🔑	SelLength
🔑	AutoSize		Hint	▷ 🔑	SelStart
🔑	BorderStyle	▷ 🔑	IsMasked	▷ 🔑	SelText
🔑	CharCase		Left		ShowHint
	Color	🔑	MaxLength	▷	Showing
▷	ComponentIndex	▷ 🔑	Modified		TabOrder
	Ctl3D		Name		TabStop
	Cursor	▷	Owner		Tag
	DragCursor	▷	Parent	▷ 🔑	Text
	DragMode		ParentColor		Top
🔑	EditMask		ParentCtl3D		Visible
▷ 🔑	EditText		ParentFont		Width
	Enabled		ParentShowHint		
	Font	🔑	PasswordChar		

Methods

	BeginDrag	🔑	GetSelTextBuf	🔑	SelectAll
	BringToFront		GetTextBuf		SendToBack
	CanFocus		GetTextLen		SetBounds
🔑	Clear		Hide		SetFocus
🔑	ClearSelection		Invalidate		SetSelTextBuf
🔑	ClientToScreen		PasteFromClipboard	🔑	SetTextBuf
🔑	CopyToClipboard		Refresh		Show
🔑	CutToClipboard		Repaint		Update
	Dragging		ScaleBy		ValidateEdit
	EndDrag		ScreenToClient		
	Focused		ScrollBy		

Events

OnChange	OnEnter	OnMouseDown
OnDblClick	OnExit	OnMouseMove
OnDragDrop	OnKeyDown	OnMouseUp
OnDragOver	OnKeyPress	
OnEndDrag	OnKeyUp	

See also

TDBEdit component, *TEdit* component

TMeasureItemEvent type — StdCtrls

Declaration

```
TMeasureItemEvent = procedure(ListBox: TListBox; Index: Integer; var Height: Integer) of
object;
```

The *TMeasureItemEvent* type points to a method that handles the measuring of an item in an owner-draw list box. The *Index* parameter identifies the position of the item in the list box and *Height* is the height of the item in pixels.

TMeasureItemEvent is the type of the *OnMeasureItem* event.

TMeasureTabEvent type — Tabs

Declaration

```
TMeasureTabEvent = procedure(Sender: TObject; Index: Integer; var TabWidth: Integer) of
object;
```

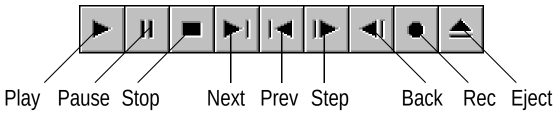
The *TMeasureTabEvent* type points to a method that handles the measuring of a tab in an owner-draw tab set control. Your code is responsible for calculating and returning the tab width, depending on what you have drawn in the tab (if the tab is of *Style tsOwnerDraw*). The *Index* parameter identifies the position of the tab in the tab set control and *TabWidth* is the width of the tab.

TMeasureTabEvent is the type of the *OnMeasureTab* event.

TMediaPlayer component — MPlayer

A *TMediaPlayer* component controls devices that provide a Media Control Interface (MCI) driver. The component is a set of buttons (Play, Stop, Eject, and so on) that controls a multimedia device such as a CD-ROM drive, a MIDI sequencer, or a VCR. A multimedia device may be hardware or software.

The media player component consists of multiple buttons. These buttons can be clicked with the mouse, but are not separate objects or button components.



Button	Value	Action
Play	*btPlay*	Plays the media player
Pause	*btPause*	Pauses playing or recording. If already paused when clicked, resumes playing or recording.
Stop	*btStop*	Stops playing or recording
Next	*btNext*	Skips to the next track, or to the end if the medium doesn't use tracks
Prev	*btPrev*	Skips to the previous track, or to the beginning if the medium doesn't use tracks
Step	*btStep*	Moves forward a number of frames
Back	*btBack*	Moves backward a number of frames
Record	*btRecord*	Starts recording
Eject	*btEject*	Ejects the medium

The multimedia device is played, paused, stopped, and so on when the user clicks the corresponding button on the *TMediaPlayer* component. The device can also be controlled by the control methods that correspond to the buttons (*Play*, *Pause*, *Stop*, *Next*, *Previous*, *Step*, *Back*, *StartRecording*, and *Eject*).

The type of multimedia device (such as *dtWaveAudio* or *dtVideodisc*) is specified by the *DeviceType* property. If the device stores its media in a file, the name of the media file is specified by the *FileName* property. If *DeviceType* is *dtAutoSelect*, the media player attempts to determine the type of device from the extension of the file specified by *FileName*.

To open a multimedia device, call the *Open* method. To have the media player attempt to open the device specified by *DeviceType* automatically when the media player component is created at run time, set the *AutoOpen* property to *True*.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for MediaPlayer component in the online Help, and choose the topic Using the Media Player Component.

Properties

▷ Align	▷ 🔑 ErrorMessage	ShowHint
▷ 🔑 AutoEnable	🔑 FileName	▷ Showing
▷ 🔑 AutoOpen	▷ 🔑 Frames	🔑 Start
🔑 AutoRewind	Height	🔑 StartPos
BoundsRect	HelpContext	TabOrder

▷ ☞ Capabilities	Hint	TabStop
☞ ColoredButtons	Left	Tag
▷ ComponentIndex	▷ ☞ Length	▷ ☞ TimeFormat
Cursor	▷ ☞ Mode	Top
▷ ☞ DeviceID	Name	▷ ☞ TrackLength
☞ DeviceType	▷ ☞ Notify	▷ ☞ TrackPosition
▷ ☞ Display	▷ ☞ NotifyValue	▷ ☞ Tracks
▷ ☞ DisplayRect	▷ Owner	Visible
Enabled	▷ Parent	☞ VisibleButtons
☞ EnabledButtons	ParentShowHint	▷ ☞ Wait
▷ ☞ EndPos	▷ ☞ Position	Width
▷ ☞ Error	☞ Shareable	

Methods

☞ Back	Hide	☞ Save
BeginDrag	Invalidate	ScaleBy
BringToFront	☞ Next	ScreenToClient
CanFocus	☞ Open	SendToBack
ClientToScreen	☞ Pause	SetBounds
☞ Close	☞ PauseOnly	SetFocus
Dragging	☞ Play	SetTextBuf
☞ Eject	☞ Previous	Show
EndDrag	Refresh	☞ StartRecording
Focused	Repaint	☞ Step
GetTextBuf	☞ Resume	☞ Stop
GetTextLen	☞ Rewind	Update

Events

☞ OnClick	OnExit	☞ OnPostClick
OnEnter	☞ OnNotify	

TMemo component

StdCtrls

A *TMemo* component displays text to the user and permits the user to enter text into the application much like a *TEdit* component. The *TMemo* component permits multiple lines to be entered or displayed, unlike *TEdit*.

The text in the memo is the value of the *Text* property. Your application can tell if the value of *Text* changes by checking the value of the *Modified* property. To limit the number of characters users can enter into the memo, use the *MaxLength* property

You can also access the text line by line using the *Lines* property. If you want to work with the text as one chunk, use the *Text* property. If you want to work with individual lines of text, the *Lines* property will suit your needs better.

You can add, delete, insert, and move lines in a memo control using the *Add*, *Delete*, and *Insert* methods of the *Lines* object, which is of type *TStrings*. For example, to add a line to a memo, you could write this line of code:

```
Memo1.Lines.Add('Another line is added');
```

You can cut, copy, and paste text to and from a memo control using the *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods.

If you want the user to be able to read the text in the memo but not to change it, set the *ReadOnly* property to *True*.

Several properties affect how the memo appears and how text is entered. You can choose to supply scroll bars in the memo with the *ScrollBars* property. If you want the memo to automatically resize to accommodate a change in font size, use the *AutoSize* property. If you want the text to break into lines, set *WordWrap* to *True*. If you want the user to be able to use tabs in the text, set *WantTabs* to *True*.

You can choose to have the text in a memo automatically selected whenever it becomes the active control with the *AutoSelect* property. At run time, you can select all the text in the memo with the *SelectAll* method. To find out which text in the memo the user has selected, or to replace selected text, use the *SelText* property. To select only part of the text or to find out what part of the text is selected, use the *SelStart* and *SelLength* properties.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for Memo component in the online Help, and choose the topic Using the Memo Component.

Properties

	Align		Left	▷ ☞	SelStart
☞	Alignment	☞	Lines	▷ ☞	SelText
☞	BorderStyle	☞	MaxLength		ShowHint
	Color	▷ ☞	Modified	▷	Showing
▷	ComponentIndex		Name		TabOrder
	Ctl3D	▷	Owner		TabStop
	Cursor	▷	Parent		Tag
	DragCursor		ParentColor	▷ ☞	Text
	DragMode		ParentCtl3D		Top
	Enabled		ParentFont		Visible
	Font		ParentShowHint	☞	WantReturns
	Height		PopupMenu	☞	WantTabs
	HelpContext	☞	ReadOnly		Width
☞	HideSelection	☞	ScrollBars	☞	WordWrap
	Hint	▷ ☞	SelLength		

Methods

BeginDrag	Focused	ScreenToClient
BringToFront	🔑 GetSelTextBuf	ScrollBy
CanFocus	GetTextBuf	🔑 SelectAll
▷ 🔑 Clear	GetTextLen	SendToBack
▷ 🔑 ClearSelection	Hide	SetBounds
ClientToScreen	Invalidate	SetFocus
🔑 CopyToClipboard	🔑 PasteFromClipboard	🔑 SetSelTextBuf
Create	Refresh	SetTextBuf
🔑 CutToClipboard	RemoveComponent	Show
Dragging	Repaint	Update
EndDrag	ScaleBy	

Events

OnChange	OnEndDrag	OnKeyUp
OnClick	OnEnter	OnMouseDown
OnDblClick	OnExit	OnMouseMove
OnDragDrop	OnKeyDown	OnMouseUp
OnDragOver	OnKeyPress	

TMemoField component

A *TMemoField* represents a field of a record in a dataset. It is represented by a value consisting of an arbitrary set of bytes with indefinite size. The bytes should correspond to text data.

Use the *Assign* method to transfer another component to a *TMemoField*. Use the *LoadFromFile* method to load a field's contents from a file. Use *LoadFromStream* method to load a field from a *Stream*. Use *SaveToFile* method to write a field's contents to a file. Use *SaveToStream* method to write a field's contents to a *Stream*.

The *TMemoField* component has the properties, methods, and events of the *TField* component.

Properties

🔑 Alignment	🔑 DisplayLabel	Name
▷ 🔑 AsBoolean	▷ 🔑 DisplayName	▷ Owner
▷ 🔑 AsDateTime	▷ 🔑 DisplayText	🔑 ReadOnly
▷ 🔑 AsFloat	🔑 DisplayWidth	🔑 Required
▷ 🔑 AsInteger	🔑 EditMask	🔑 Size
▷ 🔑 AsString	▷ 🔑 EditMaskPtr	Tag
🔑 Calculated	🔑 FieldName	▷ 🔑 Text
▷ 🔑 CanModify	▷ 🔑 FieldNo	▷ 🔑 Transliterate

▷ DataSet	Index	Visible
▷ DataSize	▷ IsIndexField	
▷ DataType	▷ IsNull	

Methods

Assign	GetData	SaveToFile
AssignValue	IsValidChar	SaveToStream
Clear	LoadFromFile	SetData
FocusControl	LoadFromStream	

Events

OnChange	OnSetText	OnValidate
OnGetText		

TMenuBreak type **Menus**

Declaration

```
TMenuBreak = (mbNone, mbBreak, mbBarBreak);
```

The *TMenuBreak* type defines the values the *Break* property of a menu item can have.

TMenuItem component **Menus**

A *TMenuItem* component contains the properties, methods, and events for each menu item on a menu (*TMainMenu* or *TPopupMenu*). Each *TMainMenu* or *TPopupMenu* component may contain multiple menu items. As you design a menu with the Menu Designer, you are creating a menu item object for each command on the menu.

When the user chooses a command on a menu, that menu item's *OnClick* event occurs.

The text that appears on a menu is the *Caption* of a menu item. You can also use the caption of the menu item to specify an accelerator key for a menu item or to provide a line that separates a menu into parts. You can assign a shortcut key to a menu item with the *ShortCut* property.

You can use the *Items* property to access a subitem of the current menu item.

If you want a check mark to alternately appear and disappear next to a menu item when the user has selected it, use the *Checked* property. If you want to disable a menu item (make it dim and unavailable to the user), set the *Enabled* property to *False*. You can simulate a user clicking a menu item with the *Click* method. If you are working with a lengthy menu, you can break the menu into two or more columns with the *Break* property.

When you want to merge menus of one form with those of another, use the *GroupIndex* property of menu items, and either the *AutoMerge* property or the *Merge* and *Unmerge* methods of a main menu (*TMainMenu*).

You can insert and delete menu items from a menu at run time with the *Insert* and *Remove* methods.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all components.

Properties

🔑 Break	Enabled	▷ Owner
🔑 Caption	🔑 GroupIndex	▷ Parent
🔑 Checked	HelpContext	🔑 ShortCut
▷ Command	Hint	Tag
▷ ComponentIndex	▷ 🔑 Items	Visible
▷ 🔑 Count	Name	

Methods

🔑 Add	🔑 IndexOf	🔑 Remove
🔑 Click	🔑 Insert	

Events

OnClick

See also

ShortCut function, *ShortCutToKey* procedure, *ShortCutToText* function, *TextToShortCut* function, *TMainMenu* component, *TPopupMenu* component

TMessageEvent type — Forms

Declaration

```
TMessageEvent = procedure (var Msg: TMsg; var Handled: Boolean) of object;
```

The *TMessageEvent* type points to a method that handles the processing of incoming Windows messages. It is the type of the *OnMessage* event handler. The *Msg* parameter identifies the Windows message, and the *Handled* parameter determines whether the message is handled or not.

TMetafile object — Graphics

A *TMetafile* object contains a Windows metafile graphic (.WMF file format).

The height and width in pixels of the metafile are specified by the *Height* and *Width* properties, respectively.

To load a metafile from a file, call the *LoadFromFile* method. To save a bitmap to a file, call *SaveToFile*.

To draw a metafile on a canvas, call the *Draw* or *StretchDraw* methods of a *TCanvas* object, passing a *TMetafile* as a parameter.

When the metafile is modified, an *OnChange* event occurs.

In addition to these properties, methods, and events, this object also has the methods that apply to all objects.

Properties

▷ 🔑 Empty	▷ 🔑 Height	▷ 🔑 Width
▷ 🔑 Handle	▷ 🔑 Inch	

Methods

Assign	ClassType	Free
ClassName	Create	🔑 LoadFromFile
ClassParent	Destroy	🔑 SaveToFile

Events

🔑 OnChange

TMethod SysUtils

Declaration

```
TMethod = record
  Code, Data: Pointer;
end;
```

TMethod declares a record that stores the *Code* and *Data* fields as type *Pointer*.

TModalResult type Forms

Declaration

```
TModalResult = Low(Integer)..High(Integer);
```

The *TModalResult* type is the type of the *ModalResult* property.

TMouseButton type — Controls

Declaration

```
TMouseButton = (mbRight, mbLeft, mbMiddle);
```

The *TMouseButton* type defines the mouse-button constants used by mouse-event handlers to distinguish which button generated the mouse event.

See also

TMouseEvent type

TMouseEvent type — Controls

Declaration

```
TMouseEvent = procedure (Sender: TObject; Button: TMouseButton; Shift:
TShiftState; X, Y: Integer) of object;
```

The *TMouseEvent* type points to a method that handles mouse-button events. The *Button* parameter determines which mouse button the user pressed, *Shift* indicates which shift keys (*Shift*, *Ctrl*, or *Alt*) and mouse buttons were down when the user pressed or released the mouse button that generated the mouse-button event. *X* and *Y* are the screen pixel coordinates of the mouse pointer.

See also

OnMouseDown event, *OnMouseUp* event

TMouseMoveEvent type — Controls

Declaration

```
TMouseMoveEvent = procedure(Sender: TObject; Shift: TShiftState; X, Y: Integer) of object;
```

The *TMouseMoveEvent* type points to a method that handles mouse-move events. The *Button* parameter determines which mouse button the user pressed, *Shift* indicates which shift keys (*Shift*, *Ctrl*, or *Alt*) and mouse buttons were down when the user moved the mouse, and *X* and *Y* are screen pixel coordinates of the new location of the mouse pointer.

See also

OnMouseMove event

TMovedEvent type **Grids**

Declaration

```
TMovedEvent = procedure (Sender: TObject; FromIndex, ToIndex: Longint) of object;
```

The *TMovedEvent* type points to a method that handles the moving of a column or row in a draw grid (*TDrawGrid*) or string grid (*TStringGrid*). The *FromIndex* parameter is the index of the column or row that is being moved, with the first column or row having an index value of 0. The *ToIndex* parameter value is the new location of the column or row after it is moved.

TMovedEvent is the type of *OnColumnMoved* and *OnRowMoved* events of the draw and string grid components.

TMPBtnType type **MPlayer**

Declaration

```
TMPBtnType = (btPlay, btPause, btStop, btNext, btPrev, btStep, btBack, btRecord, btEject);
```

The *TMPBtnType* type defines the buttons of a *TMediaPlayer* component. The buttons are included in a set of the *TButtonSet* type and are used for the *Button* parameter of the *OnClick* and *OnPostClick* events.

TMPDevCapsSet type **MPlayer**

Declaration

```
TMPDevCaps = (mpCanStep, mpCanEject, mpCanPlay, mpCanRecord, mpUsesWindows);

TMPDevCapsSet = set of TMPDevCaps;
```

The *TMPDevCapsSet* type is a set of the capabilities of the open multimedia device used with a *TMediaPlayer* component. *TMPDevCapsSet* is the type of the *Capabilities* property.

TMPDeviceTypes type **MPlayer**

Declaration

```
TMPDeviceTypes = (dtAutoSelect, dtAVIVideo, dtCDAudio, dtDAT, dtDigitalVideo, dtMMMovie,
dtOther, dtOverlay, dtScanner, dtSequencer, dtVCR, dtVideodisc, dtWaveAudio);
```

The *TMPDeviceTypes* type contains the multimedia device types that can be opened by a *TMediaPlayer* component. *TMPDeviceTypes* is the type of the *DeviceType* property.

TMPModes type

MPlayer

Applies to
TMediaPlayer component

Declaration

```
TMPModes = (mpNotReady, mpStopped, mpPlaying, mpRecording, mpSeeking, mpPaused, mpOpen);
```

The *TMPModes* type defines the modes for a multimedia device used with a *TMediaPlayer* component. *TMPModes* is the type of the *Mode* property.

TMPNotifyValues type

MPlayer

Declaration

```
TMPNotifyValues = (nvSuccessful, nvSuperseded, nvAborted, nvFailure);
```

The *TMPNotifyValues* type defines the notification values for a multimedia device used with a *TMediaPlayer* component. *TMPNotifyValues* is the type of the *NotifyValue* property.

TMPTimeFormats type

MPlayer

Declaration

```
TMPTimeFormats = (tfMilliseconds, tfHMS, tfMSF, tfFrames, tfSMPTE24, tfSMPTE25, tfSMPTE30,
  tfSMPTE30Drop, tfBytes, tfSamples, tfTMSF);
```

The *TMPTimeFormats* type defines the time formats for a multimedia device used with a *TMediaPlayer* component. *TMPTimeFormats* is the type of the *TimeFormat* property.

TMsgDlgButtons type

Dialogs

Declaration

```
TMsgDlgBtn = (mbYes, mbNo, mbOK, mbCancel, mbAbort, mbRetry, mbIgnore, mbAll, mbHelp);

TMsgDlgButtons = set of TMsgDlgBtn;
```

The *TMsgDlgButtons* type defines the set of values a button in a message box can have. The *TMsgDlgButtons* type is used by the *MessageDlg* and *MessageDlgPos* functions.

TMsgDlgType type

Dialogs

Declaration

```
TMsgDlgType = (mtWarning, mtError, mtInformation, mtConfirmation, mtCustom);
```

The *TMsgDlgType* type defines the values describing the type of message box. The *TMsgDlgType* is used by the *MessageDlg* and *MessageDlgPos* functions.

TNavigateBtn type

DBCtrls

Declaration

```
TNavigateBtn = (nbFirst, nbPrior, nbNext, nbLast, nbInsert, nbDelete, nbEdit, nbPost,
nbCancel, nbRefresh);
```

The *TNavigateBtn* type defines the possible values in the *TButtonSet* type. It is also used in the *Click* method and the *ENavClick* type.

TNotebook component

ExtCtrls

The *TNotebook* component is a component that can display multiple pages, each with its own set of controls. Notebook components are frequently used with tab set controls (*TTabSet*) to let the user select pages in the notebook by clicking a tab.

The pages available in the notebook control are the strings specified as the value of the *Pages* property. You can access a particular page in the notebook either with the *PageIndex* property or the *ActivePage* property.

If you are using a notebook with a tab set, this is the code that connects the pages of the notebook with the tabs in the tab set, displaying the page strings as the text of the tabs:

```
TabSet1.Tabs := Notebook1.Pages;
```

Then, in the *OnClick* event handler of the notebook, this line of code changes the current page in the notebook control when the user clicks a tab:

```
Notebook1.PageIndex := TabSet1.TabIndex;
```

If you are using a notebook and a tab set together, you usually want the tab set at the bottom of the form and the notebook to take up the remaining space on the form. To align the components this way, use their *Align* properties.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for Notebook component in the online Help, and choose the topic Using the Notebook Component.

Properties

🔑 ActivePage	Enabled	ParentFont
Align	Font	ParentShowHint
▷ BoundsRect	▷ Handle	PopupMenu
Color	Height	ShowHint
▷ ComponentCount	HelpContext	▷ Showing
▷ ComponentIndex	Left	TabOrder
▷ Components	Name	TabStop
▷ ControlCount	▷ Owner	Tag
▷ Controls	🔑 PageIndex	Top
Ctl3D	🔑 Pages	Visible
Cursor	▷ Parent	Width
DragCursor	ParentColor	
DragMode	ParentCtl3D	

Methods

BeginDrag	Focused	Repaint
BringToFront	Free	ScaleBy
CanFocus	GetTextBuf	ScreenToClient
ClientToScreen	GetTextLen	ScrollBy
🔑 ContainsControl	Hide	SendToBack
Create	InsertComponent	SetBounds
Destroy	InsertControl	SetTextBuf
Dragging	Invalidate	Show
EndDrag	Refresh	Update
FindComponent	RemoveComponent	

Events

OnClick	OnEndDrag	OnMouseMove
OnDblClick	OnEnter	OnMouseUp
OnDragDrop	OnExit	🔑 OnPageChanged
OnDragOver	OnMouseDown	

TNotifyEvent type

Classes

Declaration

```
TNotifyEvent = procedure (Sender: TObject) of object;
```

The *TNotifyEvent* type is the type for events that have no parameters. These events simply notify the component that a specific event occurred. For example,*OnClick*, which is type *TNotifyEvent*, notifies the control that a click event occurred on the control.

TNumGlyphs type **Buttons**

Declaration

```
TNumGlyphs: 1..4;
```

The *TNumGlyphs* type defines the range of values (1-4) the *NumGlyphs* property of a bitmap button (*TBitBtn*) or speed button (*TSpeedButton*) can assume.

TOLEContainer component **Toctrl**

The *TOLEContainer* component holds linked or embedded OLE objects. With an OLE container, you can display data from an OLE server application in your Delphi application.

When the user edits the OLE object in your application, the OLE server application is activated and handles any changes to the OLE object. When the user finishes editing the object, the OLE server application can update the object in your application. Along with the following properties, events, and methods, you should use a number of OLE routines to control the OLE container.

The object contained in a *TOLEContainer* component is defined by its OLE class, document, and item. These values are specified in the *ObjClass*, *ObjDoc*, and *ObjItem* properties, respectively.

To initialize an OLE container at run time, assign a pointer that points to an OLE initialization data structure to the *PInitInfo* property. You can obtain this pointer using the *InsertOLEObjectDlg* or *PasteSpecialDlg* functions.

To drop OLE objects onto an OLE container, you should register the form that contains the *TOLEContainer* component with the *RegisterFormAsOLEDropTarget* procedure. Then, in the *OnDragDrop* event handler of the form, the OLE object will be passed in the *Source* parameter.

To determine if an OLE object is active in place, examine the *InPlaceActive* property. If an object is activated in place, the OLE server merges menu items with the *TMainMenu* component of the main form of the OLE container application, depending on the *GroupIndex* property values of the menu items.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

Properties

▷ 🔑	Active	▷	Handle		ParentShowHint
	Align		Height	▷ 🔑	PInitInfo
🔑	AllowInPlace		HelpContext		ShowHint
🔑	AutoActivate		Hint	▷	Showing
🔑	AutoSize	▷ 🔑	InPlaceActive	▷ 🔑	Storage
	BorderStyle		Left		TabOrder

▷ BoundsRect	▷ 🔑 Modified	TabStop
▷ ComponentIndex	Name	Tag
🔑 ConvertDlgHelp	🔑 ObjClass	Top
Ctl3D	🔑 ObjDoc	Visible
Cursor	🔑 ObjItem	Width
DragCursor	▷ Owner	🔑 Zoom
DragMode	▷ Parent	
Enabled	ParentCtl3D	

Methods

BeginDrag	HandleAllocated	ScreenToClient
BringToFront	HandleNeeded	ScrollBy
CanFocus	Hide	SendToBack
ClientToScreen	Invalidate	SetBounds
🔑 CopyToClipboard	🔑 LoadFromFile	SetFocus
Dragging	🔑 OLEObjAllocated	SetTextBuf
EndDrag	Refresh	Show
Focused	Repaint	Update
GetTextBuf	🔑 SaveToFile	
GetTextLen	ScaleBy	

Events

🔑 OnActivate	OnEnter	OnMouseDown
OnDblClick	OnExit	OnMouseMove
OnDragDrop	OnKeyDown	OnMouseUp
OnDragOver	OnKeyPress	🔑 OnStatusLineEvent
OnEndDrag	OnKeyUp	

TOLEDropNotify object **Toctrl**

The *TOLEDropNotify* object is the type of the *Source* parameter of the *OnDragDrop* event of a form when an OLE object is dropped on it. To accept dropped objects, a form must be registered with the *RegisterFormAsOLEDropTarget* procedure.

In order to use the *Source* object as a *TOLEDropNotify* object, *Source* must be typecast as a *TOLEDropNotify* object.

The *DataFormat* property specifies the Clipboard format of the dropped object. The *DataHandle* property specifies a handle to the dropped data.

The *PInitInfo* property corresponds to the *PInitInfo* property of a *TOLEContainer* component. If the dropped object is an OLE object, *PInitInfo* points to an OLE initialization information structure for the OLE object. To initialize an OLE container, assign the value of the *PInitInfo* property of a *TOLEDropNotify* object to the *PInitInfo* property of a *TOLEContainer* component.

In addition to these properties and methods, this object also has the methods that apply to all objects.

Properties

▷ DataFormat	▷ DataHandle	▷ PInitInfo

Methods

ClassName	ClassType	Destroy
ClassParent	Create	Free

Top property

Applies to
All controls; *TFindDialog*, *TReplaceDialog* components

Declaration

```
property Top: Integer;
```

The *Top* property determines the *y* coordinate of the top left corner of a control, relative to the form in pixels. For forms, the value of the *Top* property is relative to the screen in pixels.

For the Find and Replace dialog boxes, Top is a run-time only property. The default value is -1.

Example
The following code moves a button 10 pixels up each time a user clicks it:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button1.Top := Button1.Top - 10;
end;
```

See also
Left property, *SetBounds* method

ToPage property

Applies to
TPrintDialog component

Declaration

```
property ToPage: Integer;
```

The value of the *ToPage* property determines on which page the print job ends. The default value is 0, which means no ending page is specified.

Example

This example uses a print dialog box on a form. The code sets up the print dialog box so that when it appears, the default values of 1 and 1 are the default starting and ending values for the Pages From and To edit boxes:

```
PrintDialog1.Options := [poPageNums];
PrintDialog1.FromPage := 1;
PrintDialog1.ToPage := 1;
```

See also

FromPage property, *Options* property

TOpenDialog component **Dialogs**

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

When the user chooses OK in the dialog box, the user's file name selection is stored in the dialog box's *FileName* property, which you can then use to process as you want.

You can let the user decide which files to make visible in the list box of the Open dialog box with the *Filter* property. The user can then use the List Files of Type combo box to determine which files display in the list box. You set the default filter using the *FilterIndex* property.

You can permit the user to choose multiple file names with the *Options* property so that the *Files* property contains a list of all the selected file names in the list box. You can customize how the Open dialog box appears and behaves with the *Options* property.

If you want a file extension automatically appended to the file name typed in the File Name edit box of the Open dialog box, use the *DefaultExt* property.

In addition to these properties and methods, this component also has the properties and methods that apply to all components.

For more information, search for OpenDialog component in the online Help, and choose the topic Using the Open Dialog Component.

Properties

▷ ComponentIndex	🔑 Filter	🔑 Options
Ctl3D	🔑 FilterIndex	▷ Owner
🔑 DefaultExt	HelpContext	Tag
🔑 FileEditStyle	🔑 HistoryList	🔑 Title
🔑 FileName	🔑 InitialDir	
🔑 Files	Name	

Methods

🔑 Execute

TOpenOptions type

Declaration

```
TOpenOption = (ofReadOnly, ofOverwritePrompt, ofHideReadOnly, ofNoChangeDir,
   ofShowHelp, ofNoValidate, ofAllowMultiSelect, ofExtensionDifferent,
   ofPathMustExist, ofFileMustExist, ofCreatePrompt, ofShareAware, ofNoReadOnlyReturn,
   ofNoTestFileCreate);
```

```
TOpenOptions = set of TOpenOption;
```

The *TOpenOptions* type contains the set of values the *Options* property of the Open dialog box (*TOpenDialog*) can assume.

TopIndex property

Applies to

TDirectoryListBox, TFileListBox, TListBox components

Declaration

```
property TopIndex: Integer;
```

The *TopIndex* property is the index number of the item that appears at the top of the list box. You can use the *TopIndex* property to determine which item is the first item displayed at the top of the list box and to set it to the item of your choosing.

Example

This example uses a list box containing a list of strings, a button, and an edit box on a form. When the user runs the application and clicks the button, the third item in the list becomes the first item, and the index value of that item appears in the edit box. The index value displayed is 2, indicating the third item in the list (the first item in the list has an index value of 0):

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Number: Integer;
begin
  for Number := 1 to 20 do
    ListBox1.Items.Add('Item ' + IntToStr(Number));
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.TopIndex := 2;
  Edit1.Text := IntToStr(ListBox1.TopIndex);
  end;
```

See also
ItemIndex property, *Items* property, *Sorted* property

TopItem property

Applies to
TOutlineNode object

Declaration

```
property TopItem: LongInt;
```

The *TopItem* property specifies the *Index* value of the level 1 parent of the outline item. For an item on level 1, *TopItem* is the same as its index. For an item that is farther down the outline tree than level 1, *TopItem* specifies the index value of the parent at the top of its outline tree branch.

Example
The following code expands the top-level parent of the selected item.

```
  with Outline1 do
    if not Items[Items[SelectedItem].TopItem].Expanded then
      Items[Items[SelectedItem].TopItem].Expanded := True;
```

See also
Level property

TopRow property

Applies to
TDrawGrid, *TStringGrid* components

Declaration

```
property TopRow: Longint;
```

Run-time only. The *TopRow* property determines which row in the grid appears at the top of the grid.

If you have one or more nonscrolling rows in the grid, they remain at the top, regardless of the value of the *TopRow* property. In this case, the row you specify as the top row will be the first row below the nonscrolling rows.

Example

This code uses a string grid and a button on a form. When the user clicks the button, the last row of the string grid becomes the top row:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  StringGrid1.TopRow := StringGrid1.RowCount;
end;
```

See also

FixedRows property, *LeftCol* property, OnTopLeftChange event

TOutline component **Outline**

The *TOutline* component is used for multilevel outlines of data. Use an outline to visually organize information in a hierarchical tree. Each item in an outline is contained in a *TOutlineNode* object.

An item in an outline can be accessed by the *Items* property. The items are indexed from 1 to the number of items. For example, *Items[1]* refers to the first (topmost) item. Since *Items* is the default array property of *TOutline*, an item can also be accessed immediately following the outline name. For example, *Outline1.Items[1]* and *Outline1[1]* refer to the same outline item.

Use the *Add* or *AddObject* methods to add a subitem to an outline. Use the *Insert* or *InsertObject* methods to replace an existing item in the outline. Use *AddChild* and *AddChildObject* to add a child item to the outline. Use *Delete* to remove items.

When adding, removing, or moving outline items, processing time can be sped up by calling *BeginUpdate* first. This prevents the outline items from being reindexed until *EndUpdate* is called.

The currently selected item is specified by the *SelectedItem* property. When the user selects a new item of the outline (by clicking with the mouse or pressing an *Arrow* key), the newly selected item is specified by *SelectedItem*.

The outline items can be represented within an outline by pictures that identify each item. The *OutlineStyle* property determines what type of pictures are used in the outline. You can also choose to display the outline tree with the *OutlineStyle* property.

The pictures displayed in an outline can be specified in the *PictureLeaf*, *PictureMinus*, *PicturePlus*, *PictureOpen*, and *PictureClosed* properties. If you don't specify these properties, an outline displays default pictures.

To display other items than the default pictures and text, set the *Style* property to *otOwnerDraw* and then draw the item in the *OnDrawItem* event handler.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for Outline component in the online Help, and choose the topic Using the Outline Component.

Properties

Align	🔑 ItemHeight	🔑 PictureMinus
BorderStyle	▷ 🔑 Items	🔑 PictureOpen
▷ BoundsRect	🔑 ItemSeparator	🔑 PicturePlus
▷ 🔑 Canvas	Left	PopupMenu
Color	🔑 Lines	▷ 🔑 Row
▷ 🔑 ComponentIndex	Name	🔑 ScrollBars
Ctl3D	🔑 Options	▷ SelectedItem
Cursor	🔑 OutlineStyle	ShowHint
DragCursor	▷ Owner	▷ Showing
DragMode	▷ Parent	Style
Enabled	ParentColor	TabOrder
Font	ParentCtl3D	TabStop
Height	ParentFont	Tag
HelpContext	ParentShowHint	Top
Hint	🔑 PictureClosed	Visible
▷ 🔑 ItemCount	🔑 PictureLeaf	Width

Methods

🔑 Add	Focused	Refresh
🔑 AddChild	🔑 FullCollapse	Repaint
🔑 AddChildObject	🔑 FullExpand	🔑 SaveToFile
🔑 AddObject	🔑 GetDataItem	ScaleBy
BeginDrag	🔑 GetItem	ScreenToClient
🔑 BeginUpdate	GetTextBuf	ScrollBy
BringToFront	🔑 GetTextItem	SendToBack
CanFocus	GetTextLen	SetBounds
🔑 Clear	Hide	SetFocus
ClientToScreen	🔑 Insert	SetTextBuf
Dragging	🔑 InsertObject	🔑 SetUpdateState
EndDrag	Invalidate	Show
🔑 EndUpdate	▷ 🔑 LoadFromFile	Update

Events

OnClick	OnEndDrag	OnKeyUp
🔑 OnCollapse	OnEnter	OnMouseDown
OnDblClick	OnExit	OnMouseMove
OnDragDrop	🔑 OnExpand	OnMouseUp
OnDragOver	OnKeyDown	
🔑 OnDrawItem	OnKeyPress	

TOutlineNode object

Outline

The *TOutlineNode* object contains an item of an *TOutline* component. An outline item is represented as a line, or row, of the outline.

An outline node contains *Text* and *Data* defined by your application. *Text* contains a string, and *Data* contains a pointer to a data structure to be associated with each outline item.

Each item can have from 0 to 16368 subitems, which are subordinate to the parent item in the outline structure.

Each item is identified by a unique *Index*. Index corresponds to the index of the *Items* property of the *TOutline* component. The children of an item are indexed sequentially, first child of an item having an *Index* value of one greater the its parent. For example, if the parent item has an *Index* of 7, its first child has an *Index* of 8, its second child has an *Index* of 9, and so on.

To move an item to a new location within the outline, call the *MoveTo* method.

The *Expanded* property determines if the item is currently expanded. If expanded, all children of an item are displayed in the outline. To set *Expanded* to *True*, call *Expand*. To set *Expanded* to *False*, call *Collapse*.

The *Level* property specifies the level, or column of an item in an outline. The items on the top level have a *Level* of 0. Their children have a level of 1, and so on. To change the level of an item, call the *ChangeLevelBy* method.

The *HasItems* property specifies whether an item has any children or subitems. *GetFirstChild* returns the *Index* value of the first child of an item. Likewise, *GetLastChild*, *GetPrevChild*, and *GetNextChild* return the index values of the last, previous, and next child items respectively.

The *IsVisible* property specifies whether an item is visible in an outline. An item is visible if all of its parents are expanded. You can expand all parents of an item with the *FullExpand* method.

The *TopItem* property specifies the *Index* value of the top-level parent of an item. The *Parent* property returns the actual immediate parent outline node of an item.

The *FullPath* property specifies the full path of parents down to an item. The path consists of the *Text* values of the parents separated by the *ItemSeparator* string specified for the *TOutline* component.

In addition to these properties and methods, this object also has the methods that apply to all objects.

Properties

▷ 🔑 Data	▷ 🔑 Index	▷ 🔑 Text
▷ 🔑 Expanded	▷ 🔑 IsVisible	▷ 🔑 TopItem
▷ 🔑 FullPath	▷ 🔑 Level	
▷ 🔑 HasItems	▷ 🔑 Parent	

Methods

ChangeLevelBy	Create	GetFirstChild
ClassName	Destroy	GetLastChild
ClassParent	Expand	GetNextChild
ClassType	Free	GetPrevChild
Collapse	FullExpand	MoveTo

TOutlineOptions type — Outline

Declaration

```
TOutlineOption = (ooDrawTreeRoot, ooDrawFocusRect, ooStretchBitmaps);
```

```
TOutlineOptions = set of TOutlineOption;
```

The *TOutlineOptions* type determines the display options for a *TOutline* component. *TOutlineStyle* is the type of the *Options* property.

TOutlineStyle type — Outline

Declaration

```
TOutlineStyle = (osText, osPlusMinusText, osPictureText, osPlusMinusPictureText,osTreeText,
osTreePictureText);
```

The *TOutlineStyle* type determines how the items of a *TOutline* component are drawn if the *Style*property is set to *osStandard*. *TOutlineStyle* is the type of the *OutlineStyle* property.

TOutlineType type — Outline

Declaration

```
TOutlineType = (otStandard, otOwnerDraw);
```

The *TOutlineType* type determines whether a *TOutline* component draws itself the standard way, or requires you to write code to draw its items. *TOutlineType* is the type of the *Style* property.

TOwnerDrawState type — StdCtrls

Declaration

```
TOwnerDrawState = set of (odSelected, odGrayed, odDisabled, odChecked, odFocused);
```

The *TOwnerDrawState* type defines the possible values for the *State* parameter in the *TDrawItemEvent* method pointer of an owner-draw list box.

TPaintBox component — ExtCtrls

The *TPaintBox* component provides a way for your application to draw on the form in a specified rectangular area, preventing drawing outside of the boundaries of the paint box. Once a paint box is added to your form, your application can use the *OnPaint* event handler to draw on the paint box's *Canvas*, the drawing surface of the paint box.

If you want to draw on the entire form, you can just use the *OnPaint* event of the form itself. If you want to confine your drawing to rectangular area, you'll find a paint box convenient.

You can align a paint box so that it remains in its relative position on the form, even when the user resizes the form. Use the *Align* property.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all controls.

Properties

Align	Font	ParentShowHint
▷ BoundsRect	Height	PopupMenu
▷ 🔑 Canvas	Hint	ShowHint
▷ ComponentIndex	Left	Tag
Color	Name	Top
Cursor	▷ Owner	Visible
DragCursor	▷ 🔑 Parent	Width
DragMode	ParentColor	
Enabled	ParentFont	

Methods

BeginDrag	GetTextBuf	SendToBack
BringToFront	GetTextLen	SetBounds
ClientToScreen	Invalidate	SetTextBuf
Dragging	Refresh	Update
EndDrag	Repaint	
Focused	ScreenToClient	

Events

OnClick	OnDragOver	OnMouseUp
OnDblClick	OnMouseDown	🔑 OnPaint
OnDragDrop	OnMouseMove	

TPanel component

ExtCtrls

The *TPanel* component is used to place panels on a form on which other controls can be placed.

Panels can be aligned with the form so that they maintain the same relative position to the form even when the form is resized. Align a panel with the *Align* property. Once a panel is aligned with the form, you can use the panel as the foundation of a tool bar, tool palette, or status bar. To make a tool bar or tool palette, add speed buttons (*TSpeedButton*) to the panel along with other controls you find useful.

The text that appears on a panel is the value of the *Caption* property. Align the caption to the left, right, or center of the panel with the *Alignment* property. You can use a panel as a status bar for your application, displaying help hints on it. See the *OnHint* event and the *Hint* property for more information.

To customize the appearance of a panel, use panel's *BevelInner*, *BevelOuter*, *BevelWidth*, and *BorderWidth* properties.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for Panel component in the online Help, and choose the topic Using the Panel Component.

Properties

🔑	Align	▷	Controls	▷	Parent
🔑	Alignment		Ctl3D		ParentColor
🔑	BevelInner		Cursor		ParentCtl3D
🔑	BevelOuter		DragCursor		ParentFont
🔑	BevelWidth		DragMode		ParentShowHint
🔑	BorderStyle		Enabled		PopupMenu
🔑	BorderWidth		Font		ShowHint
▷	BoundsRect		Height	▷	Showing
🔑	Caption		HelpContext		TabOrder
	Color		Hint		TabStop
▷	ComponentCount		Left		Tag
▷	ComponentIndex	▷ 🔑	Locked		Top
▷	Components		Name		Visible
▷	ControlCount	▷	Owner		Width

Methods

BeginDrag	GetTextBuf	ScaleBy
BringToFront	GetTextLen	ScreenToClient
CanFocus	Hide	ScrollBy
ClientToScreen	InsertComponent	SendToBack

ContainsControl	InsertControl	SetBounds
Dragging	Invalidate	SetFocus
EndDrag	Refresh	SetTextBuf
FindComponent	RemoveComponent	Show
Focused	Repaint	Update

Events

OnClick	OnDragOver	OnMouseMove
OnDblClick	OnEndDrag	OnMouseUp
OnDragDrop	OnMouseDown	🔑 OnResize

TPanelBevel type **StdCtrls**

Declaration

```
TPanelBevel = (bvNone, bvLowered, bvRaised);
```

The *TPanelBevel* type contains the values the *BevelInner* and *BevelOuter* properties can assume.

TParam object

The *TParam* object holds information about a parameter of a *TQuery* or *TStoredProc*. In addition to the parameter value, *TParam* stores the field type, name, and (for a stored procedure) the parameter type.

You generally do not need to create a *TParam* explicitly, since *TQuery* or *TStoredProc* will create it as an element of its *Params* property as needed. All you have to do is assign values to the parameters by assigning one of the properties: *AsBCD*, *AsBoolean*, *AsCurrency*, *AsDate*, *AsDateTime*, *AsFloat*, *AsInteger*, *AsSmallint*, *AsString*, *AsTime*, or *AsWord*.

In addition to these properties and methods, this object also has the methods that apply to all objects.

Properties

▷ 🔑 AsBCD	▷ 🔑 AsInteger	▷ 🔑 IsNull
▷ 🔑 AsBoolean	▷ 🔑 AsSmallInt	▷ 🔑 Name
▷ 🔑 AsCurrency	▷ 🔑 AsString	▷ 🔑 ParamType
▷ 🔑 AsDate	▷ 🔑 AsTime	▷ 🔑 Text
▷ 🔑 AsDateTime	▷ 🔑 AsWord	
▷ 🔑 AsFloat	▷ 🔑 DataType	

Methods

▷ 🔑 Assign	ClassType	Free
▷ 🔑 AssignField	Clear	GetData
▷ 🔑 ClassName	Create	GetDataSize
▷ 🔑 ClassParent	Destroy	SetData

TParamBindMode type **DBTables**

Declaration

```
TParamBindMode = (pbByName, pbByNumber);
```

The *TParamBindMode* type defines the possible values of the *ParamBindMode* property of a stored procedure (*TStoredProc*).

TParams object

The *TParams* object holds the parameters for a stored procedure (*TStoredProc*) or parameterized query (*TQuery*) and provides the methods to create and access those parameters. Each parameter is a *TParam* object.

Use the *Items* property to access individual parameters. Call *CreateParam* to create a new parameter. Call *AddParam* to add a new parameter or *RemoveParam* to take one out of the set. Call *Clear* to delete all parameters. Use the *ParamByName* method to find a parameter with a particular name.

In addition to these properties and methods, this object also has the methods that apply to all objects.

Properties

▷ 🔑 Items

Methods

🔑 AddParam	🔑 ClassType	🔑 Destroy
🔑 Assign	🔑 Clear	🔑 Free
🔑 AssignValues	🔑 Count	🔑 ParamByName
🔑 ClassName	🔑 Create	🔑 RemoveParam
🔑 ClassParent	🔑 CreateParam	

TParamType type DBTables

Declaration

```
TParamType = (ptUnknown, ptInput, ptOutput, ptInputOutput, ptResult);
```

The *TParamType* type is the set of values of the *ParamType* property of a *TParam* object

TPasswordEvent type DB

Declaration

```
TPasswordEvent = procedure(Sender: TObject; var Continue: Boolean);
```

The *TPasswordEvent* type is the header for the procedure that handles a password exception event. The value of *Sender* is the *TSession* component of the *DB* unit. *Continue* determines whether the caller will make another attempt to access the database. The procedure should add any available additional passwords and set *Continue* to *True*. If there are no additional passwords available, set *Continue* to *False*. *TPasswordEvent* is used by the *OnPassword event*

TPen object Graphics

A *TPen* object is used to draw lines on a canvas (*TCanvas*). The *TPen* object encapsulates the Windows HPEN.

The color of the pen is specified by the *Color* property. The width in pixels of the line drawn is specified by the *Width* property. The pattern of the line (solid, dotted, and so on) is specified by the *Style* property.

The *Mode* property specifies the color of the line, as it relates to the pixels the line is drawn over. For example, to color the line the color specified by the *Color* property, set *Mode* to *pmCopy*. To color the line the inverse of the screen color, set *Mode* to *pmNot*.

If the pen is modified, an *OnChange* event occurs.

In addition to these properties, methods, and events, this object also has the methods that apply to all objects.

Properties

▷ 🔑 Color	▷ 🔑 Mode	▷ 🔑 Width
▷ 🔑 Handle	▷ 🔑 Style	

Methods

Assign	ClassType	Free
ClassName	Create	
ClassParent	Destroy	

Events

🔑 OnChange

TPenMode type — Graphics

Declaration

```
TPenMode = (pmBlack, pmWhite, pmNop, pmNot, pmCopy, pmNotCopy,pmMergePenNot, pmMaskPenNot,
pmMergeNotPen, pmMaskNotPen, pmMerge,pmNotMerge, pmMask, pmNotMask, pmXor, pmNotXor);
```

The *TPenMode* type specifies the values the *Mode* property of pen object (*TPen*) can assume.

TPenStyle type — Graphics

Declaration

```
TPenStyle = (psSolid, psDash, psDot, psDashDot, psDashDotDot, psClear, psInsideFrame);
```

The *TPenStyle* type specifies the values the *Style* property of pen object (*TPen*) can assume.

TPicture object — Graphics

The *TPicture* object contains a bitmap, icon, or metafile graphic. The type of graphic contained by the *TPicture* is specified in the *Graphic* property.

If the *TPicture* contains a bitmap graphic, the *Bitmap* property specifies the graphic. If the *TPicture* contains an icon graphic, the *Icon* property specifies the graphic. If the *TPicture* contains a metafile graphic, the *Metafile* property specifies the graphic.

The height and width in pixels of the graphic are specified by the *Height* and *Width* properties, respectively.

To load a graphic from a file, call the *LoadFromFile* method. To save a bitmap to a file, call *SaveToFile*. To load or save a picture to the Clipboard, use the *Assign* method of a *TClipboard* object.

To draw a picture on a canvas, call the *Draw* or *StretchDraw* methods of a *TCanvas* object, passing the *Graphic* property of a *TPicture* as a parameter.

When the graphic is modified, an *OnChange* event occurs.

In addition to these properties, methods, and events, this object also has the methods that apply to all objects.

Properties

▷ 🔑 Bitmap	▷ 🔑 Height	▷ 🔑 Metafile
▷ 🔑 Graphic	▷ 🔑 Icon	▷ 🔑 Width

Methods

Assign	ClassType	Free
ClassName	Create	🔑 LoadFromFile
ClassParent	Destroy	🔑 SaveToFile

Events

🔑 OnChange

TPoint type — WinTypes

Declaration

```
TPoint = record
  X: Integer;
  Y: Integer;
end;
```

The *TPoint* type defines a pixel location onscreen, with the origin in the top left corner. *X* specifies the horizontal coordinate of the point, *Y* specifies the vertical coordinate.

TPopupAlignment type — Menus

Declaration

```
TPopupAlignment = (paLeft, paRight, paCenter);
```

The *PopupAlignment* type determines where a pop-up menu (*TPopupMenu*) appears. The *Alignment* property of a pop-up menu is of type *TPopupAlignment*.

TPopupMenu component — Menus

The *TPopupMenu* component encapsulates the properties, methods, and events of a pop-up menu, the menu available to forms and controls when the user selects the component and clicks the right mouse button. To make a pop-up menu available, assign a *TPopupMenu* component to the form's or control's *PopupMenu* property.

To begin designing a pop-up menu, add a pop-up menu component to your form, and double-click the component. For more information, see the topic Menu Designer in the Help system.

The items on the pop-up menu are specified with the *Items* object, a property of a pop-up menu. The *Items* object is of type *TMenuItem*. Your application can use the *Items* property to access a particular item on the menu.

If you want the pop-up menu to appear when the user clicks the right mouse button on the control to which the pop-up menu is assigned, set the *AutoPopup* property to *True*. If you want to use code to control when a pop-up menu appears, use the *Popup* method.

Pop-up menus have an *OnPopup* event handler you can use to specify special processing you want to occur in your application just before a pop-up menu appears.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all components.

For more information, search for PopupMenu component in the online Help, and choose the topic Using the Popup Menu Component.

Properties

☞	Alignment	▷	Components		Name
☞	AutoPopup	▷	Handle	▷	Owner
▷	ComponentCount		HelpContext	☞	PopupComponent
▷	ComponentIndex	☞	Items		Tag

Methods

	FindComponent		Free	☞	Popup
	FindItem				

Events

	OnPopup

See also

ShortCut function, *ShortCutToKey* procedure, *ShortCutToText* function, *TextToShortCut* function, *TMainMenu* component, *TMenuItem* component

TPosition type

Forms

Declaration

```
TPosition = (poDesigned, poDefault, poDefaultPosOnly, poDefaultSizeOnly, poScreenCenter);
```

The *TPosition* type enumerates the values the *Position* property of a form can have.

TPrintDialog component **Dialogs**

The *TPrintDialog* component displays a Print dialog box that permits the user to select which printer to print to, which pages to print, how many copies to print, and if the print job should be collated. If the user chooses the Setup button in the Print dialog box, the Print Setup dialog (*TPrinterSetupDialog* component) appears.

Display the Print dialog box by calling the *Execute* method.

You can customize how the Print dialog box appears and behaves using the *Options* property. For example, you can determine which print options are enabled or disabled, or you can decide whether the option to print to a file appears.

The range of pages to be printed is specified with the *PrintRange* property. If the value of *PrintRange* is *prPageNums*, which allows users to specify a page range, the pages they specify are the values of the *FromPage* and *ToPage* properties. With the *MinPage* and *MaxPage*, your application can limit the range of pages the user can select.

If the user chooses to print to a file, the *PrintToFile* property is *True*. If the user wants the print job to be collated, the *Collate* property is *True*.

In addition to these properties and methods, this component also has the properties and methods that apply to all components.

For more information, search for PrintDialog component in the online Help, and choose the topic Using the Print Dialog Component.

Properties

🔑	Collate	🔑	MaxPage	🔑	PrintRange
▷	ComponentIndex	🔑	MinPage	🔑	PrintToFile
🔑	Copies		Name		Tag
🔑	FromPage	🔑	Options	🔑	ToPage
	HelpContext	▷	Owner		

Methods

🔑	Execute

TPrintDialogOptions type **Dialogs**

Declaration

```
TPrintDialogOption = (poPrintToFile, poPageNums, poSelection, poWarning, poHelp,
poDisablePrintToFile);
```

```
TPrintDialogOptions = set of TPrintDialogOption;
```

The *TPrintDialogOptions* type defines the set of values the *Options* property of the Print dialog box (*TPrintDialog*) can have.

TPrinter object

Printers

The *TPrinter* object encapsulates the printer interface of Windows. Within the *Printers* unit, the variable *Printer* is declared as an instance of *TPrinter*, ready for you to use.

To start a print job, call the *BeginDoc* method. To end a print job that is sent successfully to the printer, call the *EndDoc* method. If a problem occurs and you need to terminate a print job that was not sent to the printer successfully, call the *Abort* method.

You can determine if a job is printing by checking the value of the *Printing* property. If the job aborted, the *Aborted* property is *True*.

The printing surface of a page is represented by the *Canvas* property. You can use the *Brush*, *Font*, and *Pen* properties of the *Canvas* object to determine how drawing or text appears on the page.

The list of installed printers is found in the *Printers* property. The value of the *PrinterIndex* property is the currently selected printer. The list of fonts supported by the current printer is found in the *Fonts* property.

You can determine if a print job prints in landscape or portrait orientation using the *Orientation* property.

You height and width of the current page is found in the *PageHeight* and *PageWidth* properties. The current page is the value of the *PageNumber* property.

The *Title* property determines the text that appears listed in the Print Manager and on network header pages.

Using the *PrintScale* property of a *TForm* component, you determine how the printed image of the form appears.

Whenever you use a *TPrinter* object, you must add *Printers* to the **uses** clause of the unit that implements the properties or methods of a *TPrinter* object.

In addition to these properties and methods, this object also has the methods that apply to all objects.

Properties

▷ 🔑 Aborted	▷ 🔑 Orientation	▷ 🔑 PrinterIndex
▷ 🔑 Canvas	▷ 🔑 PageHeight	▷ 🔑 Printers
▷ 🔑 Fonts	▷ 🔑 PageNumber	▷ 🔑 Printing
▷ Handle	▷ 🔑 PageWidth	▷ 🔑 Title

Methods

▷ 🔑 Abort	ClassType	🔑 GetPrinter
▷ 🔑 BeginDoc	Create	🔑 NewPage
ClassName	Destroy	🔑 SetPrinter
ClassParent	🔑 EndDoc	

TPrinterOrientation type — Printers

Declaration

```
TPrinterOrientation = (poPortrait, poLandscape);
```

The *TPrinterOrientation* type defines the possible values of the *Orientation* property of the printer object (*TPrinter*).

TPrinterSetupDialog component — Dialogs

The *TPrinterSetupDialog* component displays a Printer Setup dialog box in your application. Users can use the dialog box to setup their printer before printing a job.

Display the Printer Setup dialog box by calling the *Execute* method. The Printer Setup dialog box also appears when the user chooses the Setup button in the Print dialog box (*TPrintDialog*).

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all components.

For more information, search for PrinterSetupDialog component in the online Help, and choose the topic Using the Printer Setup Dialog Component.

Properties

▷ ComponentIndex	Name	Tag
HelpContext	▷ Owner	

Methods

🔑 Execute

TPrintRange type — Dialogs

Declaration

```
TPrintRange = (prAllPages, prSelection, prPageNums);
```

The *TPrintRange* type defines the values the *PrintRange property* can have in the Print dialog box (*TPrintDialog*).

TPrintScale type **Forms**

Declaration

```
TPrintScale = (poNone, poProportional, poPrintToFit);
```

The *TPrintScale* type defines the possible values of the *PrintScale* property of the form.

TQuery component **DBTables**

TQuery enables Delphi applications to issue SQL statements to a database engine--either the BDE or an SQL server. *TQuery* provides the interface between an SQL server (or the BDE) and *TDataSource* components. *TDataSource* components then provide the interface to data-aware controls such as *TDBGrid*.

Set the *DatabaseName* property to specify the database to query. Enter a single SQL statement to execute in the *SQL* property. To query dBASE or Paradox tables, use local SQL. To query SQL server tables, use passthrough SQL. The SQL statement can be a *static SQL statement* or a *dynamic SQL statement*.

At run time, an application can supply parameter values for dynamic queries with the *Params* property, the *ParamByName* method, or the *DataSource* property. Use the *Prepare* method to optimize a dynamic query.

A *result set* is the group of records returned by a query to an application. A *TQuery* can return two kinds of result sets:

- “Live” result sets: As with *TTable* components, users can edit data in the result set with data controls. The changes are sent to the database when a *Post* occurs, or when the user tabs off a control.
- “Read only” result sets: Users cannot edit data in the result set with data controls.

If you want the query to provide a live result set, the SQL must conform to certain *syntax requirements*. If the SQL syntax does not conform to these requirements, the query will provide a read-only result set.

Execute the SQL statement at design time by setting the *Active* property to *True* . Execute the SQL statement at run time with the *Open* or *ExecSQL* methods.

Call the *First*, *Next*, *Prior*, *Last*, and *MoveBy* methods to navigate through the result set. Test the *BOF* and *EOF* properties to determine if the cursor is at the beginning or end of the result set, respectively.

Call the *Append*, *Insert*, *AppendRecord* or *InsertRecord* methods to add a record to the underlying database table. Call the *Delete* method to delete the current record. Call the *Edit* method to allow modification of the fields of the current record, and *Post* to post the changes or *Cancel* to discard them.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all components.

Properties

☞ Active	▷ ☞ FieldDefs	▷ ☞ RecordCount
☞ AutoCalcFields	▷ ☞ Fields	☞ RequestLive
▷ ☞ BOF	▷ ☞ Handle	☞ SQL
▷ ☞ CanModify	▷ ☞ Local	▷ ☞ SQLBinary
▷ ☞ Database	▷ ☞ Locale	▷ ☞ State
☞ DatabaseName	▷ ☞ Modified	▷ ☞ StmtHandle
☞ DataSource	Name	Tag
▷ ☞ DBHandle	▷ Owner	▷ ☞ Text
▷ ☞ DBLocale	▷ ☞ ParamCount	☞ UniDirectional
▷ ☞ EOF	☞ Params	☞ UpdateMode
▷ ☞ FieldCount	▷ ☞ Prepared	

Methods

☞ Append	☞ FieldByName	☞ Open
☞ AppendRecord	☞ FindField	☞ ParamByName
☞ Cancel	☞ First	☞ Post
☞ CheckBrowseMode	☞ FreeBookmark	☞ Prepare
☞ ClearFields	☞ GetBookmark	☞ Prior
☞ Close	☞ GetFieldNames	☞ Refresh
☞ CursorPosChanged	☞ GotoBookmark	☞ SetFields
☞ Delete	☞ Insert	☞ UnPrepare
☞ DisableControls	☞ InsertRecord	☞ UpdateCursorPos
☞ Edit	☞ Last	☞ UpdateRecord
☞ EnableControls	☞ MoveBy	
☞ ExecSQL	☞ Next	

Events

AfterCancel	AfterPost	BeforeOpen
AfterClose	BeforeCancel	BeforePost
AfterDelete	BeforeClose	OnCalcFields
AfterEdit	BeforeDelete	OnNewRecord
AfterInsert	BeforeEdit	
AfterOpen	BeforeInsert	

TrackCursor procedure **WinCrt**

Declaration

```
procedure TrackCursor;
```

The *TrackCursor* procedure scrolls the CRT window if necessary to ensure that the cursor is visible.

Example

```
uses WinCrt;

var
  x: integer;

begin
  for x := 1 to 30 do
    Write('Xx');
  TrackCursor;
  Readln;
end;
```

See also

ScrollTo procedure

TrackLength property

Applies to

TMediaPlayer component

Declaration

```
property TrackLength[TrackNum: Integer]: Longint;
```

Run-time and read only. The *TrackLength* property reports the length of the track specified by the *TrackNum* index. The value of *TrackLength* is specified according to the current time format, which is specified in the *TimeFormat* property.

Example
The following code shows the length of the currently playing .WAV audio file (CARTOON.WAV in this example) in the *Caption* of a label.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  with MediaPlayer1 do begin
    TimeFormat := tfMilliseconds;
    DeviceType := dtWaveAudio;
    FileName := 'cartoon.wav';
    Open;
    Label1.Caption := IntToStr(TrackLength[1]);
  end;
end;
```

See also
Length property, *TrackPosition* property, *Tracks* property

TrackPosition property

Applies to
TMediaPlayer component

Declaration

```
property TrackPosition[TrackNum: Integer]: Longint;
```

Run-time and read only. The *TrackPosition* property reports the starting position of the track specified by the *TrackNum* index. The value of *TrackPosition* is specified according to the current time format, which is specified in the *TimeFormat* property.

Example
The following code shows the starting position of the first track of the currently loaded audio CD in the *Caption* of a label.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  with MediaPlayer1 do begin
    TimeFormat := tfMilliseconds;
    DeviceType := dtCDAudio;
    Open;
    Label1.Caption := IntToStr(TrackPosition[1]);
  end;
end;
```

See also
Position property, *TrackLength* property, *Tracks* property

Tracks property

Applies to

TMediaPlayer component

Declaration

```
property Tracks: Longint;
```

Run-time and read only. The *Tracks* property specifies the number of playable tracks on the open multimedia device. *Tracks* is undefined for devices that don't use tracks.

Example

The following code skips to the beginning of the last track on a CD audio device. You must declare the integer variable *I* to run this code.

```
with MediaPlayer1 do
if DeviceType = dtCDAudio then
begin
  Seek(TrackPosition[1]);
  for I := 1 to Tracks-1 do
    Next;
end;
```

See also

Next method, *Previous* method, *TrackLength* property, *TrackPosition* property

TRadioButton component **StdCtrls**

The *TRadioButton* component is a Windows radio button. Use radio buttons to present a set of mutually exclusive options to the user—that is, only one radio button in a set can be selected at any time. When the user selects a radio button, the previously selected radio button becomes unselected.

Radio buttons are frequently grouped in a group box (*TGroupBox*). Add the group box to the form first, then choose the radio buttons from the Component palette and put them in the group box.

The text associated with the radio button that identifies its purpose is the value of the *Caption* property.

When the user selects a radio button, the value of the *Checked* property changes. Also, the *OnClick* event occurs. If you check a radio button, all other radio buttons in the same group become unchecked. By default, all radio buttons that are directly contained by the same windowed control container, such as a *TForm*, *TGroupBox*, or *TPanel*, are grouped. For example, two radio buttons on a form can be checked at the time only if they are contained in separate containers, such as two different group boxes.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for RadioButton component in the online Help, and choose the topic Using the Radio Button Component.

Properties

	Align		Font		ParentFont
🔑	Alignment	▷	Handle		ParentShowHint
🔑	Caption		Height		PopupMenu
🔑	Checked		HelpContext		ShowHint
	Color		Hint	▷	Showing
▷	ComponentIndex		Left		TabOrder
	Ctl3D		Name		TabStop
	Cursor	▷	Owner		Tag
	DragCursor	▷	Parent		Top
	DragMode		ParentColor		Visible
	Enabled		ParentCtl3D		Width

Methods

BeginDrag	GetTextLen	SendToBack
BringToFront	Hide	SetBounds
CanFocus	Invalidate	SetFocus
ClientToScreen	Refresh	SetTextBuf
Dragging	Repaint	Show
EndDrag	ScaleBy	Update
Focused	ScreenToClient	
GetTextBuf	ScrollBy	

Events

OnClick	OnEnter	OnMouseDown
OnDblClick	OnExit	OnMouseMove
OnDragDrop	OnKeyDown	OnMouseUp
OnDragOver	OnKeyPress	
OnEndDrag	OnKeyUp	

See also

TDBRadioGroup component

TRadioGroup component ExtCtrls

A radio group box is a group box that contains radio buttons. A radio group box simplifies the task of grouping radio buttons and getting them to work together as a group.

The radio buttons are added to the group box when strings are entered as the value of the *Items* property. Each string in *Items* makes a radio button appear in the group box with the string appearing as the caption of the radio button. The value of the *ItemIndex* property determines which radio button is currently selected.

You can choose to display the radio buttons in a single column or in multiple columns by setting the value of the *Columns* property.

When the user selects a radio button in the radio button group box, the previously selected radio button is unselected automatically.

You can place other types of controls in a radio group box besides radio buttons.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for RadioGroup in the online Help, and choose the topic Using the RadioGroup Component.

Properties

	Align		Height		ParentShowHint
🔑	Caption		HelpContext		PopupMenu
	Color		Hint		ShowHint
🔑	Columns	🔑	Items	▷ 🔑	Showing
▷	ComponentIndex	🔑	ItemIndex		TabOrder
	Ctl3D		Left		TabStop
	Cursor		Name		Tag
	DragCursor	▷	Owner		Top
	DragMode	▷	Parent		Visible
	Enabled		ParentColor		Width
	Font		ParentCtl3D		
▷	Handle		ParentFont		

Methods

BeginDrag	Focused	ScreenToClient
BringToFront	GetTextBuf	ScrollBy
CanFocus	GetTextLen	SendToBack
ClientToScreen	Hide	SetBounds
ContainsControl	Invalidate	SetFocus
Dragging	Refresh	SetTextBuf
EndDrag	Repaint	
FindComponent	ScaleBy	

Events

OnClick	OnDragOver	OnEnter
OnDragDrop	OnEndDrag	OnExit

See also

TDBRadioGroup component, *TGroupBox* component, *TRadioButton* component

TransIsolation property

Applies to

TDataBase component

Declaration

```
property TransIsolation: TTransIsolation;
```

The *TransIsolation* property specifies the transaction isolation level used by an SQL server. *tiDirtyRead* causes any change to be returned, regardless of whether the record has been committed. *tiReadCommitted* will return only committed versions of the record; uncommitted changes will not be reflected in the result. *tiRepeatableRead* will return only the original record for the duration of the transaction, even if another application has committed a change.

Database servers may support these isolation levels differently or not at all. If the requested isolation level is not supported by the server, then Delphi will use the next highest isolation level, as shown in the following table. For a detailed description of how each isolation level is implemented, see your server documentation.

TransIsolation setting	Oracle	Sybase and Microsoft SQL servers	Informix	InterBase
Dirty read	Read committed	Read committed	Dirty Read	Read committed
Read committed (Default)	Read committed	Read committed	Read committed	Read committed
Repeatable read	Repeatable read (READ ONLY)	Read committed	Repeatable Read	Repeatable Read

Example

```
Database1.TransIsolation := tiReadCommitted;
```

Transliterate property

Applies to

TBatchMove, *TMemoField*, *TStringField* component

Declaration

```
property Transliterate: Boolean;
```

The *Transliterate* property controls whether translations to and from the respective locales of the *Source* and *Destination* properties will be done. *Transliterate* is *True* by default.

Example

```
{ Suppress translations }
BatchMove1.Transliterate := False;
```

See also

DBLocale property

Transparent property

Applies to

TLabel, *TDBText* components

Declaration

```
property Transparent: Boolean;
```

The *Transparent* property determines if a label or database text control is transparent. You could place a transparent label or text control on top of a bitmap, and the control won't hide part of the bitmap. For example, if you have placed a bitmap of the world on a form, you could label the South American continent with a label control, and you would still see the continent in the label space.

Example

This code makes a label transparent:

```
Label1.Transparent := True;
```

See also

BorderStyle property

TRect type **WinTypes**

Declaration

```
TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
end;
```

The *TRect* type defines a rectangle. The coordinates are specified either as four separate integers representing the pixel locations of the left, top, right, and bottom sides, or as two points representing the pixel locations of the top left and bottom right corners. The origin of the pixel coordinate system is in the top left corner of the screen.

TReplaceDialog component **Dialogs**

The *TReplaceDialog* component provides a Replace dialog box your application can use. *TReplaceDialog* contains all the capabilities of the *TFindDialog* component, but it also allows the user to replace found text with a replacement string.

Display the Replace dialog box by calling the *Execute* method.

The text your application is searching for is the value of the *FindText* property. The text that is to replace the found text is the value of the *ReplaceText* property.

To determine which search and replace options are available in the Find dialog box, use the *Options* property. For example, you can have a Match Case check box appear in the dialog box or hide it, and you can disable or enable the Whole Word check box.

When the user enters the text to search for in the dialog box and chooses Find Next, the *OnFind* event occurs. Within the *OnFind* event handler, write the code that searches for the text specified as the value of the *FindText*. Your code can use the *Options* values to determine how the user wants the search conducted.

When the user chooses the Replace or Replace All button in the Replace dialog box, the *OnReplace* event occurs. Within the *OnReplace* event handler, write the code that replaces the found text specified as the value of *ReplaceText*. Your code can use the *Options* values to determine how the user wants the text replaced.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all components.

For more information, search for ReplaceDialog component in the online Help, and choose the topic Using the Replace Dialog Component.

Properties

▷	ComponentIndex	HelpContext	▷	Owner
	Ctl3D	Name		ReplaceText
	FindText	Options		Tag

Methods

CloseDialog	Execute

Events

OnFind	OnReplace

TReport component

Report

The *TReport* component is an interface to Borland's ReportSmith application. Once you place the *TReport* component on a form, you can double-click it to begin running ReportSmith. If you specify an existing report name using the Object Inspector, choose Cancel when the Open Report dialog box appears and ReportSmith minimizes. When you click the ReportSmith icon to restore it, the specified report displays.

You can also control the running of ReportSmith at run time. When you run a report at run time, you are using ReportSmith Runtime.

Specify the report you want to run as the value of the *ReportName* property. You can include the path where the report is located as part of the *ReportName* property, or you can specify the directory with the *ReportDir* property. Call the *Run* method to run the report. To print an existing report, call the *Print* method.

The report component lets you specify several report parameters. You can choose the starting and ending page of the report with the *StartPage* and *EndPage* properties. To limit the size of the report, you can specify a maximum number of records for the report with the *MaxRecords* property. Choose how many copies of the report you want with the *PrintCopies* property.

Specify the report variables you want the report to use with the *InitialValues* property. You can change a report variable with the *SetVariable* and *SetVariableLines* methods, and then recalculate the report and run it using the new report variable by calling the *RecalcReport* method.

You can run a ReportBasic macro using the *RunMacro* method.

To terminate the running of a report, call the *CloseReport* method. You can choose to automatically unload ReportSmith Runtime when a report finishes running if you set the *AutoUnload* property to *True*. If *AutoUnload* is *False*, you must call the *CloseApplication* method to unload ReportSmith Runtime.

For information about using ReportSmith, see the ReportSmith documentation.

In addition to these properties and methods, this component also has the properties and methods that apply to all components.

For more information, search for Report component in the online Help, and choose the topic Using the Report Component.

Properties

🔑 AutoUnload	▷ Owner	🔑 StartPage
▷ ComponentIndex	🔑 Preview	Tag
🔑 EndPage	🔑 PrintCopies	▷ 🔑 VersionMajor
🔑 InitialValues	▷ 🔑 ReportHandle	▷ 🔑 VersionMinor
🔑 MaxRecords	🔑 ReportDir	
Name	🔑 ReportName	

Methods

🔑 CloseApplication	🔑 Print	🔑 RunMacro
🔑 CloseReport	🔑 RecalcReport	🔑 SetVariable
🔑 Connect	🔑 Run	🔑 SetVariableLines

Trunc function **System**

Declaration

```
function Trunc(X: Real): Longint;
```

The *Trunc* function truncates a real-type value to an integer-type value.

X is a real-type expression. *Trunc* returns a *Longint* value that is the value of *X* rounded toward zero.

If the truncated value of *X* is not within the *Longint* range, an error occurs, which you can handle using the *EInvalidOp* exception. If you do not handle it, you will receive a run-time error.

Example

```
var
  S, T: string;
begin
  Str(1.4:2:1, T);
  S := T + ' Truncs to ' + IntToStr(Trunc(1.4)) + #13#10;
  Str(1.5:2:1, T);
  S := S + T + ' Truncs to ' + IntToStr(Trunc(1.5)) + #13#10;
  Str(-1.4:2:1, T);
  S := S + T + ' Truncs to ' + IntToStr(Trunc(-1.4)) + #13#10;
  Str(-1.5:2:1, T);
  S := S + T + ' Truncs to ' + IntToStr(Trunc(-1.5));
  MessageDlg(S, mtInformation, [mbOk], 0);
end;
```

See also
Int function, *Round* function

Truncate method

Applies to
TBlobStream object

Declaration

```
procedure Truncate;
```

The *Truncate* method discards all data in the *TBlobField, TBytesField or TVarBytesField from the current position.*

Example

```
{ Discard all data after the first 1000 bytes }
with BlobStream1 do
  begin
  Seek(0, 1000);
  Truncate;
  end;
```

Truncate procedure **System**

Declaration

```
procedure Truncate(var F);
```

The *Truncate* procedure deletes all records in the file after the current file position *F*. The current file position also becomes end-of-file (*Eof*(*F*) is *True*).

F is a file variable of any type. *Truncate* does not work on text files. *F* must be open.

{$I+} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using **{$I–}**, you must use *IOResult* to check for I/O errors.

Example

```
uses WinCRT;

var
   f: file of Integer;
   i,j: Integer;
 begin
   AssignFile(f,'TEST.INT');
   Rewrite(f);
   for i := 1 to 6 do
     Write(f,i);
   Writeln('File before truncation:');
   Reset(f);
   while not Eof(f) do
   begin
     Read(f,i);
     Writeln(i);
   end;
   Reset(f);
   for i := 1 to 3 do
     Read(f,j); { Read ahead 3 records }
   Truncate(f); { Cut file off here }
   Writeln;
   Writeln('File after truncation:');
```

T

```
    Reset(f);
    while not Eof(f) do
    begin
      Read(f,i);
      Writeln(i);
    end;
    CloseFile(f);
    Erase(f);
  end;
```

See also

Reset procedure, *Rewrite* procedure, *Seek* procedure

TSaveDialog component

Dialogs

The *TSaveDialog* component makes a Save dialog box available to your application. The purpose of the dialog box is to allow a user to specify a file to save. Use the *Execute* method to display the Save dialog box.

When the user chooses OK in the dialog box, the user's file name selection is stored in the dialog box's *FileName* property, which you can then use to process as you want.

You can let the user decide which set of files are visible in the list box of the Save dialog box with the *Filter* property. The user can then use the List Files of Type combo box to determine which files display in the list box. You set the default filter using the *FilterIndex* property.

You can permit the user to choose multiple file names with the *Options* property so that the *Files* property contains a list of all the selected file names in the list box. You can customize how the Save dialog box appears and behaves with the *Options* property.

If you want a file extension automatically appended to the file name typed in the File Name edit box of the Save dialog box, use the *DefaultExt* property.

In addition to these properties and methods, this component also has the properties and methods that apply to all components.

For more information, search for SaveDialog component in the online Help, and choose the topic Using the Save Dialog Component.

Properties

ComponentIndex	🔑 Filter	🔑 Options
Ctl3D	🔑 FilterIndex	Owner
🔑 DefaultExt	HelpContext	Tag
🔑 FileEditStyle	🔑 HistoryList	🔑 Title
🔑 FileName	🔑 InitialDir	
🔑 Files	Name	

Methods

☞ Execute

See also

TOpenDialog component

TScreen component

Forms

The *TScreen* component represents the state of the screen as your application runs. A *Screen* variable of type *TScreen* is already declared, ready for you to use as an instance of *TScreen*.

The screen component lists all forms displayed on the screen in the *Forms* property array. The number of forms is kept as the value of the *FormCount* property. You can find out which form currently has the focus by checking the value of the *ActiveForm* property. Similarly, the control that has the focus is the value of the *ActiveControl* property.

The height and width of the screen device in pixels are the value of the *Height* and *Width* properties. The *PixelsPerInch* property tells you how many pixels are in an inch using the current video driver.

All the fonts supported by the screen are listed in the *Fonts* property. Similarly, all the cursors available are in the *Cursors* property. Using *Cursors*, you can make a custom cursor available to your application.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all components.

Properties

▷ ☞ ActiveControl	▷ ☞ Cursors	▷ Owner
▷ ☞ ActiveForm	▷ ☞ FormCount	▷ ☞ PixelsPerInch
▷ ComponentCount	▷ ☞ Forms	▷ Tag
▷ ComponentIndex	▷ ☞ Fonts	▷ Width
▷ Components	▷ Height	
▷ Cursor	▷ Name	

Methods

FindComponent	InsertComponent	RemoveComponent

Events

☞ OnActiveControlChange	☞ OnActiveFormChange

TScrollBar component

StdCtrls

The *TScrollBar* component is a Windows scroll bar, which is used to scroll the contents of a window, form, or control. In the *OnScroll* event handler, you write the code that determines how the window, form, or control behaves in response to the user scrolling the scroll bar.

You determine how far a thumb tab moves when the user clicks the scroll bar on either side of the thumb tab with the value of the *LargeChange* property. The value of the *SmallChange* property determines how far the thumb tab moves when the user clicks the arrows at the end of the scroll bar or scrolls the scroll bar using the arrow keys on the keyboard.

The *Min* and *Max* property values together determine how many positions are available on the scroll bar for the thumb tab to move when the user scrolls the scroll bar. Your application can set the position of the thumb tab with the *Position* property, or use the property to determine how far the scroll bar has scrolled. You can use the *SetParams* method to set the *Min*, *Max*, and *Position* properties all at once.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for ScrollBar component in online Help, and choose the topic Using the Scroll Bar Component.

Properties

▷ Align	🔑 Kind	🔑 Position
▷ ComponentIndex	🔑 LargeChange	ShowHint
Ctl3D	Left	▷ Showing
Cursor	🔑 Max	🔑 SmallChange
DragCursor	🔑 Min	TabOrder
DragMode	Name	TabStop
Enabled	▷ Owner	Tag
▷ 🔑 Handle	▷ Parent	Top
Height	ParentCtl3D	Visible
HelpContext	ParentShowHint	Width
Hint	PopupMenu	

Methods

BeginDrag	GetTextLen	SetBounds
BringToFront	Hide	SetFocus
CanFocus	Invalidate	🔑 SetParams
ClientToScreen	Refresh	SetTextBuf
Dragging	Repaint	Show
EndDrag	ScaleBy	Update

Focused	ScreenToClient
GetTextBuf	SendToBack

Events

OnChange	OnEndDrag	OnKeyPress
OnClick	OnEnter	OnKeyUp
OnDragDrop	OnExit	🔑 OnScroll
OnDragOver	OnKeyDown	

See also
TScrollBox component, *TForm* component

TScrollBarInc type — Forms

Declaration

```
TScrollBarInc = 1..32767;
```

Description
The *TScrollBarInc* type defines the possible values of the *SmallChange* and *LargeChange* properties of a scroll bar (*TScrollBar*).

TScrollBarKind type — Forms

Declaration

```
TScrollBarKind = (sbHorizontal, sbVertical);
```

The *TScrollBarKind* type defines the two different orientations a scroll bar can have: horizontal and vertical. *TScrollBarKind* is the type of the scroll bar control's *Kind* property.

TScrollBox component — Forms

Scroll box components make it possible to create scrolling areas on a form that are smaller than the entire form. For example, your form might contain a panel component used as a speed bar that is aligned with the top of the form, and a panel component used as a status bar that is aligned with the bottom of the form. You would not want these panel components to scroll when the user scrolls the form. By placing a scroll box between the two panels and aligning it so that it fills the remaining client area, you can place components on the scroll box and then allow the user to scroll only the scroll box on the form.

You determine the behavior of the horizontal and vertical scroll bars on the scroll box by setting the properties of the *HorzScrollBar* and *VertScrollBar* objects, which are properties of a scroll box.

To assure that a particular control on the scroll box is in view, use the *ScrollInView* method.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

Properties

🔑 Align	Enabled	ParentShowHint
🔑 BorderStyle	Font	PopupMenu
▷ Brush	Height	ShowHint
Color	HelpContext	▷ Showing
▷ ComponentCount	Hint	TabOrder
▷ ComponentIndex	🔑 HorzScrollBar	TabStop
▷ Components	Left	Tag
▷ ControlCount	Name	Top
▷ Controls	Owner	🔑 VertScrollBar
Ctl3D	Parent	Visible
Cursor	ParentColor	Width
DragCursor	ParentCtl3D	
DragMode	ParentFont	

Methods

BeginDrag	GetTextLen	ScrollBy
BringToFront	Hide	ScrollInView
CanFocus	InsertComponent	SendToBack
ClientToScreen	InsertControl	SetBounds
🔑 ContainsControl	Invalidate	SetFocus
Dragging	Refresh	SetTextBuf
EndDrag	RemoveComponent	Show
FindComponent	Repaint	Update
Focused	ScaleBy	
GetTextBuf	ScreenToClient	

Events

OnClick	OnEndDrag	OnMouseMove
OnDblClick	OnEnter	OnMouseUp
OnDragDrop	OnExit	🔑 OnResize
OnDragOver	OnMouseDown	

TScrollCode type StdCtrls

Declaration

```
TScrollCode = (scLineUp, scLineDown, scPageUp, scPageDown, scPosition, scTrack, scTop,
scBottom, scEndScroll);
```

Description

The *TScrollCode* type defines the possible states of a scroll bar. It is used by the *TScrollEvent* method pointer.

TScrollEvent type StdCtrls

Declaration

```
TScrollEvent = procedure(Sender: TObject; ScrollCode: TScrollCode; var ScrollPos: Integer) of
object;
```

The *TScrollEvent* type points to a method that handles the scrolling of a scroll bar. The *ScrollCode* parameter is one of these values:

Value	Meaning
scLineUp	User clicked the top or left scroll arrow or pressed the *Up* arrow key
scLineDown	User clicked the bottom or right scroll arrow or pressed the *Down* arrow key
scPageUp	User clicked the area to the left of the thumb tab or pressed the *PgUp* key
scPageDown	User clicked the area to the right of the thumb tab or pressed the *PgDn* key
scPosition	User positioned the thumb tab and released it.
scTrack	User is moving the thumb tab
scTop	User moved the thumb tab to the top or far left on the scroll bar
scBottom	User moved the thumb tab to the bottom or far right on the scroll bar
scEndScroll	User is done moving the thumb tab on the scroll bar

The *ScrollPos* parameter indicates the position of the thumb tab on the scroll bar.

TScrollEvent is the type of the *OnScroll* event.

TScrollStyle type StdCtrls

Declaration

```
TScrollStyle = (ssNone, ssHorizontal, ssVertical, ssBoth);
```

The *TScrollStyle* type defines the different combinations of scroll bars a memo control or a grid can have. *TScrollStyle* is the type of the *ScrollBars* property of *TMemo*, *TDBMemo*, *TDrawGrid*, and *TStringGrid*.

TSearchRec type — SysUtils

Declaration

```
TSearchRec = record
     Fill: array[1..21] of Byte;
     Attr: Byte;
     Time: Longint;
     Size: Longint;
     Name: string[12];
   end;
```

The *TSearchRec* type defines file information searched for by a *FindFirst* or *FindNext* function call. If a file is found, the fields of the *TSearchRec* type parameter are modified to specify the found file.

Attr represents the file attributes the file attributes of the file. Test *Attr* against the following attribute constants or values to determine if a file has a specific attribute:

Constant	Value	Description
faReadOnly	$01	Read-only files
faHidden	$02	Hidden files
faSysFile	$04	System files
faVolumeID	$08	Volume ID files
faDirectory	$10	Directory files
faArchive	$20	Archive files

To test for an attribute, combine the value of the *Attr* field with the attribute constant with the and operator. If the file has that attribute, the result will be greater than 0. For example, if the found file is a hidden file, the following expression will evaluate to *True*: (*SearchRec.Attr* and *faHidden* > 0).

Time contains the time stamp of the file. *Size* contains the size of the file in bytes. *Name* contains the DOS file name and extension.

TSectionEvent type — Headers

Declaration

```
TSectionEvent = procedure(Sender: TObject; ASection, AWidth: Integer) of object;
```

The *TSectionEvent* type is used by the *OnSized* and *OnSizing* events of the *THeader* component. The index of the header section being resized is passed in the *ASection* parameter, and its width in pixels is passed in the *AWidth* parameter.

TSelectCellEvent type — Grids

Declaration

```
TSelectCellEvent = procedure (Sender: TObject; Col, Row: Longint; var CanSelect: Boolean) of
object;
```

The *TSelectEvent* type points to a method that handles the selecting of a cell in a draw grid (*TDrawGrid*) or string grid (*TStringGrid*). The *Col* and *Row* parameters specify the column and row of the selected cell. The value of the *CanSelect* parameter determines whether the user can successfully select the cell in the grid.

TSelectEvent is the type of the *OnSelectCell* event of the draw and string grid components.

TSelectDirOpts type — FileCtrl

Declaration

```
TSelectDirOpt = (sdAllowCreate, sdPerformCreate, sdPrompt);
```

```
TSelectDirOpts = set of TSelectDirOpt;
```

The *TSelectDirOpts* type defines the possible values of the *Options* parameter in the *SelectDirectory* function.

TSession component

You cannot see nor explicitly create a *TSession* component, but you can use its methods and properties to globally affect an application. Delphi creates a *TSession* component named *Session* each time an application runs. Do not attempt to create any other *TSession* or destroy and recreate *Session* itself.

TSession provides global control over database connections for an application. The *Databases* property of *TSession* is an array of all the active databases in the session. The *DatabaseCount* property is an integer specifying the number of active databases in the Session.

KeepConnections is a *Boolean* property that specifies whether to keep inactive database connections for temporary *TDatabase* components. The *DropConnections* method will drop all inactive database connections.

The *NetFileDir* property specifies the directory path of the BDE network control directory. The *PrivateDir* property specifies the path of the directory in which to store temporary files.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all components.

Properties

▷ 🔑 DatabaseCount	▷ 🔑 Locale	▷ 🔑 PrivateDir
▷ 🔑 Databases	▷ Name	▷ Tag
▷ 🔑 Handle	▷ 🔑 NetFileDir	
▷ 🔑 KeepConnections	▷ Owner	

Methods

🔑 AddPassword	🔑 GetAliasParams	🔑 GetTableNames
🔑 CloseDatabase	🔑 GetDatabaseNames	🔑 GetStoredProcNames
🔑 DropConnections	🔑 GetDriverNames	🔑 OpenDatabase
🔑 FindDatabase	🔑 GetDriverParams	🔑 RemoveAllPasswords
🔑 GetAliasNames	🔑 GetPassword	🔑 RemovePassword

Events

🔑 *OnPassword*

TSetEditEvent type

Grids

Declaration

```
TSetEditEvent = procedure (Sender: TObject; ACol, ARow: Longint; const Text: string)
  of object;
```

Description

The *TSetEditEvent* type points to a method that handles the changes the user makes to the text in a cell in a draw grid (*TDrawGrid*) or string grid (*TStringGrid*). The *ACol* parameter specifies the column of the cell, and the *ARow* parameter specifies the row of the cell. The *Text* parameter is the text string the editor has changed.

TSetEditEvent is the type of the *OnSetEditText* event of the draw and string grid components.

TShape component

ExtCtrls

The *TShape* component displays a geometric shape on the form. It is a nonwindowed component.

You determine which geometric shape the shape control assumes by setting the *Shape* property. How the shape is painted depends on the two nested properties of the *Brush* object, *Color* and *Style*.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all controls.

For more information, search for Search component in online Help, and choose the topic Using the Shape Component.

Properties

▷ Align	Height	🔑 Shape
▷ BoundsRect	Hint	ShowHint
🔑 Brush	Left	Tag
ComponentIndex	Name	Top
Cursor	▷ Owner	Visible
DragCursor	▷ Parent	Width
DragMode	ParentShowHint	
Enabled	🔑 Pen	

Methods

BeginDrag	Hide	ScreenToClient
BringToFront	Invalidate	SendToBack
ClientToScreen	Refresh	SetBounds
Dragging	Repaint	Show
EndDrag	ScaleBy	Update

Events

OnDragDrop	OnEndDrag	OnMouseMove
OnDragOver	OnMouseDown	OnMouseUp

TShapeType type

StdCtrls

Declaration

```
TShapeType = (stRectangle, stSquare, stRoundRect, stRoundSquare, stEllipse, stCircle);
```

The *TShapeType* type is used by the *Shape* property of the *TShape* component to determine if the *TShape* component appears as a rectangle, square, rounded rectangle, rounded square, ellipse, or circle.

TShiftState type **Classes**

Declaration

```
TShiftState = set of (ssShift, ssAlt, ssCtrl, ssRight, ssLeft, ssMiddle, ssDouble);
```

The *TShiftState* type is used by key-event and mouse-event handlers to determine the state of the *Alt*, *Ctrl*, and *Shift* keys and the state of the mouse buttons when the event occurred.

See also

TKeyEvent type, *TMouseEvent* type

TShortCut type **Menus**

Declaration

```
TShortCut = Low(Word) .. High(Word);
```

TShortCut types are the menu shortcuts that appear on menus and give the user an alternate way to select a menu commands using the keyboard. The *ShortCut* property is of type *TShortCut*, and the *ShortCutToText* and *ShortCutToKey* routines use parameters of type *TShortCut*.

TShowHintEvent type **Forms**

Declaration

```
TShowHintEvent = procedure (var HintStr: string; var CanShow: Boolean; var HintInfo:
THintInfo) of object;
```

The *TShowHintEvent* type points to a method that displays a Help Hint for a control. It is the type of the *OnShowHint* event.

The *HintStr* parameter is the text of the Help Hint. To obtain the text of a hint for a particular control, call the *GetShortHint* function, assigning the result to *HintStr*. You can change the contents of this string if you want to change the text.

Use the *CanShow* variable to permit or prevent the Help Hint from displaying. If *CanShow* is *True*, the Help Hint displays. If it is *False*, the Help Hint does not appear.

The *HintInfo* parameter is a record that contains information about the appearance and behavior of the help window.

The *HintColor* field of the record contains the name of the control for which hint processing is occurring.

The *HintPos* field determines the default position in screen coordinates of the top-left corner of the hint window. You can change where the window appears by changing this value.

The *HintMaxWidth* field determines the maximum width of the hint window before word wrapping begins. By default, the value is the width of the screen (*Screen.Width*).

The *CursorRect* field determines the rectangle the user's mouse pointer must be in for the hint window to appear. The default value for *CursorRect* is the client rectangle of the control. Your application can change this value so that a single control can divided into several hint regions. When the user moves the mouse pointer moves outside the rectangle, the hint window disappears.

The *CursorPos* field contains the location of the mouse pointer within the control.

TSmallintField component

A *TSmallintField* represents a field of a record in a dataset. It is represented as a binary value with a range from –32,768 to 32,767. Use *TSmallintField* for fields that hold signed whole numbers.

Set the *DisplayFormat* property to control the formatting of the field for display purposes, and the *EditFormat* property for editing purposes. Use the *Value* property to access or change the current field value. Set the *MinValue* or the *MaxValue* property to limit the smallest or largest value permitted in a field.

The *TSmallintField* component has the properties, methods, and events of the *TField* component.

Properties

🔑 Alignment	🔑 DisplayFormat	▷ 🔑 IsNull
▷ 🔑 AsBoolean	🔑 DisplayLabel	🔑 MaxValue
▷ 🔑 AsDateTime	▷ 🔑 DisplayName	🔑 MinValue
▷ 🔑 AsFloat	▷ 🔑 DisplayText	Name
▷ 🔑 AsInteger	🔑 DisplayWidth	▷ Owner
▷ 🔑 AsString	🔑 EditFormat	🔑 ReadOnly
🔑 Calculated	🔑 EditMask	🔑 Required
▷ 🔑 CanModify	▷ 🔑 EditMaskPtr	▷ 🔑 Size
▷ 🔑 DataSet	🔑 FieldName	Tag
▷ 🔑 DataSize	▷ 🔑 FieldNo	▷ 🔑 Text
▷ 🔑 DataType	🔑 Index	▷ 🔑 Value
▷ 🔑 AsFloat	▷ 🔑 IsIndexField	🔑 Visible

Methods

🔑 Assign	🔑 FocusControl	🔑 SetData
🔑 AssignValue	🔑 GetData	
🔑 Clear	🔑 IsValidChar	

Events

🔑 OnChange	🔑 OnSetText	🔑 OnValidate
🔑 OnGetText		

TSpeedButton component — Buttons

TSpeedButton components are buttons that usually have graphical images on their faces that users use to execute commands or set modes. They have some unique capabilities that allow them to work as a set. Speed buttons are commonly used with panels (*TPanel*) to create tool bars and tool palettes.

The graphical image that appears on a speed button is the value of its *Glyph* property. You can use different images to represent the different states of the speed button. For example, you can use one image when the speed button is unselected, another when it is selected, and another when it is disabled. Use the *NumGlyphs* property to specify multiple images.

Use the *Layout*, *Margin*, and *Spacing* properties to arrange the image and text on the speed button.

Speed buttons can work together as a group if you set the *GroupIndex* property. If you want all speed buttons in a group to be able to appear in their “up” state, set the *AllowAllUp* property to *True*. If you want a speed button to initially appear as if it is selected, set its *Down* property to *True*.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all controls.

For more information, search for SpeedButton component in online Help, and choose the topic Using the SpeedButton Component.

Properties

▷ Align	🔑 GroupIndex	ParentFont
▷ 🔑 AllowAllUp	Height	ParentShowHint
▷ BoundsRect	Hint	ShowHint
Caption	🔑 Layout	▷ Showing
▷ ComponentIndex	Left	▷ 🔑 Spacing
Cursor	🔑 Margin	Tag
Down	Name	Top
Enabled	🔑 NumGlyphs	Visible

	Font	▷	Owner	Width
🔑	Glyph	▷	Parent	

Methods

BeginDrag	GetTextBuf	ScreenToClient
BringToFront	GetTextLen	SendToBack
CanFocus	Hide	SetBounds
ClientToScreen	Invalidate	SetTextBuf
Click	Refresh	Show
Dragging	Repaint	Update
EndDrag	ScaleBy	

Events

OnClick	OnMouseDown	OnMouseUp
OnDblClick	OnMouseMove	

TStatusLineEvent type **ToCtrl**

Declaration

```
TStatusLineEvent = procedure(Sender: TObject; Msg: string) of object;
```

TStatusLineEvent is the type of the *OnStatusLineEvent* event of the *TOLEContainer* component. *Msg* contains the string message from the OLE server application.

TStoredProc component

The *TStoredProc* component enables Delphi applications to execute server stored procedures. Set the *DatabaseName* property to specify the database in which the stored procedure is defined. Set the *StoredProcName* to the name of the stored procedure on the server.

A stored procedure has a *Params* array for its input and output parameters, similar to a *TQuery* component. The order of the parameters in the *Params* array is determined by the stored procedure definition. An application can set the values of input parameters and get the values of output parameters in the array similar to *TQuery* parameters. You can also use *ParamByName* to access the parameters by name. If you are not sure of the ordering of the input and output parameters for a stored procedure, use the Parameters Editor.

Before an application can execute a stored procedure, you must prepare the stored procedure, which can be done:

- At design time with the Parameters Editor.
- At run time with the *Prepare* method.

A stored procedure can return either a singleton result or a result set with a cursor, if the server supports it. Execute a stored procedure with the *ExecProc* method, if the stored procedure returns a singleton result (one row), or *Open* method, if the stored procedure returns a result set (multiple rows).

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all components.

Properties

🔑 Active	▷ 🔑 FieldDefs	🔑 Params
🔑 AutoCalcFields	▷ 🔑 Fields	▷ 🔑 Prepared
▷ 🔑 BOF	▷ 🔑 Handle	▷ 🔑 RecordCount
▷ 🔑 CanModify	▷ 🔑 Locale	▷ 🔑 State
▷ 🔑 Database	▷ 🔑 Modified	▷ 🔑 StmtHandle
🔑 DatabaseName	Name	🔑 StoredProcName
▷ 🔑 DBHandle	▷ Owner	Tag
▷ 🔑 DBLocale	🔑 Overload	▷ 🔑 UpdateMode
▷ 🔑 EOF	🔑 ParamBindMode	
▷ 🔑 FieldCount	▷ 🔑 ParamCount	

Methods

🔑 Append	🔑 ExecProc	▷ 🔑 Next
🔑 AppendRecord	🔑 FieldByName	▷ 🔑 Open
🔑 Cancel	🔑 FindField	▷ 🔑 ParamByName
🔑 CheckBrowseMode	🔑 First	▷ 🔑 Post
🔑 ClearFields	🔑 FreeBookmark	🔑 Prepare
🔑 Close	🔑 GetBookmark	🔑 Prior
🔑 CopyParams	🔑 GetFieldNames	🔑 Refresh
🔑 CursorPosChanged	🔑 GetResults	🔑 SetFields
🔑 Delete	🔑 GotoBookmark	🔑 UnPrepare
🔑 DescriptionsAvailabl e	🔑 Insert	🔑 UpdateCursorPos
🔑 DisableControls	🔑 InsertRecord	🔑 UpdateRecord
🔑 Edit	🔑 Last	
🔑 EnableControls	🔑 MoveBy	

Events

AfterCancel	AfterPost	BeforeOpen
AfterClose	BeforeCancel	BeforePost
AfterDelete	BeforeClose	OnCalcFields
AfterEdit	BeforeDelete	OnNewRecord
AfterInsert	BeforeEdit	
AfterOpen	BeforeInsert	

TStringField component

A *TStringField* component represents a field of a record in a dataset. A field of *TStringField* is physically stored as a sequence of up to 255 characters. Use *TStringField* for fields that contain text, such as names and addresses.

Use the *Value* property to access or change the current field value.

The *TStringField* component has the properties, methods, and events of the *TField* component.

Properties

🔑 Alignment	🔑 DisplayLabel	Name
▷ 🔑 AsBoolean	▷ 🔑 DisplayName	▷ Owner
▷ 🔑 AsDateTime	▷ 🔑 DisplayText	🔑 ReadOnly
▷ 🔑 AsFloat	🔑 DisplayWidth	🔑 Required
▷ 🔑 AsInteger	🔑 EditMask	🔑 Size
▷ 🔑 AsString	▷ 🔑 EditMaskPtr	Tag
🔑 Calculated	🔑 FieldName	▷ 🔑 Text
▷ 🔑 CanModify	▷ 🔑 FieldNo	🔑 Transliterate
▷ 🔑 DataSet	🔑 Index	▷ 🔑 Value
▷ 🔑 DataSize	▷ 🔑 IsIndexField	🔑 Visible
▷ 🔑 DataType	▷ 🔑 IsNull	

Methods

🔑 Assign	🔑 FocusControl	🔑 SetData
🔑 AssignValue	🔑 GetData	
🔑 Clear	🔑 IsValidChar	

Events

🔑 OnChange	🔑 OnSetText	🔑 OnValidate
🔑 OnGetText		

TStringGrid component

Grids

The *TStringGrid* component is a grid control designed to simplify the handling of strings and associated objects while maintaining all the functionality of the *TDrawGrid* component.

All the strings within a string grid are contained in the *Cells* property, which you can use to access a particular string within the grid. All the objects associated with the strings in a string grid are contained in the *Objects* property. Use *Objects* to access a particular object.

All the strings and their associated objects for a particular column can be accessed using the *Cols* property. The *Rows* property gives you access to all the strings and their associated objects for a particular row.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for StringGrid component in online Help, and choose the topic Using the String Grid Component.

Properties

	Align	🔑	DefaultRowHeight		ParentColor
🔑	BorderStyle		DragCursor		ParentCtl3D
▷	BoundsRect		DragMode		ParentFont
▷	Canvas	🔑	EditorMode		ParentShowHint
▷ 🔑	Cells		Enabled		PopupMenu
▷	ClientHeight		FixedColor	▷ 🔑	Row
▷	ClientOrigin		FixedCols	🔑	RowCount
▷	ClientRect		FixedRows	🔑	RowHeights
▷	ClientWidth		Font	🔑	Rows
▷ 🔑	Col	▷ 🔑	GridHeight	🔑	ScrollBars
🔑	ColCount	🔑	GridLineWidth	▷ 🔑	Selection
	Color	▷ 🔑	GridWidth		ShowHint
▷ 🔑	Cols	▷	Handle	▷	Showing
▷ 🔑	ColWidths		Height		TabOrder
▷	ComponentCount		HelpContext		TabStop
▷	ComponentIndex		Hint		TabStops
▷	Components		Left		Tag
▷	ControlCount	🔑	LeftCol		Top
▷	Controls		Name	▷ 🔑	TopRow
	Ctl3D	▷ 🔑	Objects		Visible
	Cursor	🔑	Options	▷ 🔑	VisibleColCount
🔑	DefaultColWidth	▷	Owner	▷ 🔑	VisibleRowCount
🔑	DefaultDrawing	▷	Parent		Width

Methods

	BeginDrag		GetTextBuf		ScaleBy
	BringToFront		GetTextLen		ScreenToClient
	CanFocus		Hide		SendToBack
🔑	CellRect		Invalidate		SetBounds
	ClientToScreen	🔑	MouseToCell		SetFocus
	Dragging		Refresh		SetTextBuf
	EndDrag		RemoveComponent		Show
	Focused		Repaint		Update

Events

OnClick	OnEnter	OnMouseDown
🔑 OnColumnMoved	OnExit	OnMouseMove
OnDblClick	🔑 OnGetEditMask	OnMouseUp
OnDragDrop	🔑 OnGetEditText	🔑 OnRowMoved
OnDragOver	OnKeyDown	🔑 OnSelectCell
🔑 OnDrawCell	OnKeyPress	🔑 OnSetEditText
OnEndDrag	OnKeyUp	🔑 OnTopLeftChanged

TStringList object **Classes**

The *TStringList* object maintains a list of strings. Use a string list object when you are managing a list of strings that is not maintained by a control.

You can add, delete, insert, move, and exchange strings using the *Add*, *Delete*, *Insert*, *Move*, and *Exchange* methods. The *Clear* method clears all the strings in the list of strings. The *Count* property contains the number of strings in the list. Each string list object has a *Strings* property that lets you access a particular string by its position in the list of strings. To find the position of a string in the list, use the *IndexOf* method.

If you want to add several strings at once to a list of strings, use the *AddStrings* method. You can assign one strings object to another using the *Assign* method.

To determine if a particular string exists in the list of strings, call the *Find* method. To sort the list of strings, use the *Sort* method. To determine if the list is sorted, check the value of the *Sorted* property. You can decide whether the list can contain duplicate strings using the *Duplicates* property.

Each string can be associated with an object. The objects associated with strings are commonly used to place a bitmap in an owner-draw control. If you want to add a string and an object to string list at the same time, use the *AddObject* method. You can access a particular object by its position in the list of objects using the *Objects* property. To find the position of the object in the list, use the *IndexOfObject* method. To insert an object, call the *InsertObject* method. The *Delete*, *Move*, *Clear*, and *Exchange* methods operate on the object associated with a string as well as on the string itself. For example, calling *Clear* removes all strings and all their associated objects.

You can store strings in a file and then load them all at one using the *LoadFromFile* method. To save the strings to a file, use the *SaveToFile* method.

In addition to these properties, methods, and events, this object also has the methods that apply to all objects.

Properties

▷ 🔑 Count	▷ 🔑 Objects	▷ 🔑 Strings
▷ 🔑 Duplicates	▷ 🔑 Sorted	▷ 🔑 Values

Methods

🔑 Add	🔑 Clear	🔑 Insert
🔑 AddObject	🔑 Delete	🔑 InsertObject
🔑 AddStrings	🔑 EndUpdate	🔑 LoadFromFile
🔑 Assign	🔑 Exchange	🔑 SaveToFile
🔑 BeginUpdate	Free	SetText
ClassName	GetText	🔑 Sort
ClassParent	🔑 IndexOf	
ClassType	🔑 IndexOfObject	

Events

OnChange

TStrings object — Classes

String objects are used by various components to manipulate strings. A string object has no way to store a string, but instead uses the native storage ability of the control that uses it.

For example, the *Items* property of a list box control is of type *TStrings*. The strings that appear in a list box control are stored in a list box string object (*TListBoxStrings)*, which is derived from *TStrings*. When you add or delete items in a list box, you are adding and deleting them from a list box string object.

To maintain a list of strings outside of a control, use a string list object (*TStringList*).

You can add, delete, insert, move, and exchange strings using the *Add*, *Delete*, *Insert*, *Move*, and *Exchange* methods of a string object. The *Clear* method clears all the strings in the list of strings. The *Count* property contains the number of strings in the list. Each string object has a *Strings* property that lets you access a particular string by its position in the list of strings. To find the position of a string in the list, use the *IndexOf* method.

To add several strings at once to a list of strings, use the *AddStrings* method. You can assign one string object to another using the *Assign* method.

Each string can be associated with an object. The objects associated with strings are commonly used to place a bitmap in an owner-draw control. If you want to add a string and an object to string list at the same time, use the *AddObject* method. You can access a particular object by its position in the list of objects using the *Objects* property. To find the position of the object in the list, use the *IndexOfObject* method. To insert an object, call the *InsertObject* method. The *Delete*, *Move*, *Clear*, and *Exchange* methods operate on the object associated with a string as well as on the string itself. For example, calling *Clear* removes all strings and all their associated objects.

You can store strings in a file and then load them all at one using the *LoadFromFile* method. To save the strings to a file, use the *SaveToFile* method.

In addition to these properties and methods, this object also has the methods that apply to all objects.

Properties

▷ 🔑 Count	▷ 🔑 Strings	▷ 🔑 Values
▷ 🔑 Objects		

Methods

🔑 Add	▷ 🔑 Clear	🔑 Insert
🔑 AddObject	▷ 🔑 Delete	🔑 InsertObject
🔑 AddStrings	▷ 🔑 Exchange	🔑 LoadFromFile
🔑 Assign	▷ 🔑 EndUpdate	🔑 Move
🔑 BeginUpdate	Free	🔑 SaveToFile
ClassName	🔑 GetText	SetText
ClassParent	🔑 IndexOf	
ClassType	🔑 IndexOfObject	

TSymbolStr type **DB**

Declaration

```
TSymbolStr = string[DBIMAXNAMELEN];
```

The *TSymbolStr* type is the type of a string of the correct length for a database object name, such as a *Locale* property, a field name, or a password.

TTabbedNotebook component **TabNotBk**

The *TTabbedNotebook* component contains multiple pages, each with its own set of controls. The user selects a page by clicking the page's tab that appears at the top of the control.

The pages available in the tabbed notebook control are the strings specified as the value of the *Pages* property. You can access a particular page in the notebook either with the *PageIndex* property or the *ActivePage* property.

At run time, you can change the active page in the tabbed notebook with the *SetTabFocus* method. If you need to determine the *PageIndex* value of a particular page, call the *GetIndexForPage* method.

You determine how many tabs appear in a row by setting the *TabsPerRow* property. If there are more pages than there are tabs in one row, multiple rows automatically appear in the control. You can specify the font of the text on the tabs with the *TabFont* property.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

Properties

🔑	ActivePage	▷	Handle	▷	Parent
	Align		Height		TabOrder
▷	BoundsRect		HelpContext	🔑	TabsPerRow
▷	ComponentIndex		Hint		TabStop
▷	ControlCount		Left		Tag
▷	Controls		Name		Top
	Cursor	▷	Owner		Visible
	Enabled	🔑	PageIndex		Width
	Font	🔑	Pages		

Methods

	BeginDrag		GetTextBuf		ScrollBy
	BringToFront		GetTextLen		SendToBack
	CanFocus		Hide		SetBounds
	ClientToScreen		InsertControl		SetFocus
🔑	ContainsControl		Invalidate	🔑	SetTabFocus
	Dragging		Refresh		SetTextBuf
	EndDrag		Repaint		Show
	Focused		ScaleBy		Update
🔑	GetIndexForPage		ScreenToClient		

Events

OnEnter	OnExit

TTabChangeEvent type — Tabs

Declaration

```
TTabChangeEvent = procedure(Sender: TObject; NewTab: Integer; var AllowChange: Boolean) of
object;
```

The *TTabChangeEvent* type points to a method that is called when the selected tab (the *TabIndex*) is about to change in a tab set control. The *NewTab* parameter is the tab that is about to become the selected tab. The *AllowChange* variable determines whether the change is permitted. If *AllowChange* is *False*, the user won't be able to select the new tab, in effect disabling it.

TTable component — DBTables

The *TTable* component provides live access to database tables through the Borland Database Engine. *TTable* is the interface between the Borland Database Engine and

TDataSource components. The *TDataSource* components then provide the interface to data-aware controls such as *TDBGrid*.

Set the *DatabaseName* property to specify the database to access. Set the *TableName* property to the table to access. Set the *ReadOnly* property to *True* unless you want to change the contents of the table. Set the *Exclusive* property to *True* if you do not want any other application to access the table while you are using it. Use the *IndexName* property to use the table with a secondary index. Use the *MasterFields* and *MasterSource* properties to create a link to a master table in a master-detail relationship. Call the *GotoCurrent* method to move the cursor to the same position as another *TTable* linked to the same database table.

Set the *Active* property to *True* or call the *Open* method to open a *TTable*, putting it in Browse mode. Set *Active* to *False* or call *Close* close the *TTable*. Call the *First*, *Next*, *Prior*, *Last*, and *MoveBy*, methods to navigate through the table. Call the *SetKey*, *FindKey*, *FindNearest*, *GotoKey*, and *GotoNearest* methods to search the database table for specific values.

Test the *BOF* and *EOF* properties to determine if the cursor has reached the beginning or end of the table, respectively. Call the *Append*, *Insert*, *AppendRecord* or *InsertRecord* methods to add a record to the table. Call the *Delete* method to delete the current record. Call the *Edit* method to allow an application to modify records in the table, and *Post* to send the changes to the database or *Cancel* to discard them.

Use the *EditRangeStart*, *EditRangeEnd*, *SetRangeStart*, *SetRangeEnd*, *ApplyRange* and *SetRange* methods to limit the range of records returned to the application and the *CancelRange* method to remove the limit.

In addition to these properties, methods, and events, this component also has the properties and methods that apply to all components.

Properties

🔑 Active	▷ 🔑 Fields	▷ 🔑 Modified
🔑 AutoCalcFields	▷ 🔑 Handle	Name
▷ 🔑 BOF	▷ 🔑 IndexDefs	▷ Owner
▷ 🔑 CanModify	▷ 🔑 IndexFieldCount	🔑 ReadOnly
▷ 🔑 Database	🔑 IndexFieldNames	▷ 🔑 RecordCount
🔑 DatabaseName	🔑 IndexName	▷ 🔑 State
▷ 🔑 DBHandle	▷ 🔑 IndexFields	🔑 TableName
▷ 🔑 DBLocale	▷ 🔑 KeyExclusive	🔑 TableType
▷ 🔑 EOF	▷ 🔑 KeyFieldCount	Tag
🔑 Exclusive	▷ 🔑 Locale	🔑 UpdateMode
▷ 🔑 FieldCount	🔑 MasterFields	
▷ 🔑 FieldDefs	🔑 MasterSource	

Methods

🔑 AddIndex	🔑 EditKey	🔑 GotoNearest
🔑 ApplyRange	🔑 EditRangeEnd	🔑 Insert

Append	EditRangeStart	InsertRecord
AppendRecord	EmptyTable	Last
BatchMove	EnableControls	MoveBy
Cancel	FieldByName	Next
CancelRange	FindField	Open
CheckBrowseMode	FindKey	Post
ClearFields	FindNearest	Prior
Close	First	Refresh
CreateTable	FreeBookmark	SetFields
CursorPosChanged	GetBookmark	SetKey
Delete	GetFieldNames	SetRange
DeleteIndex	GetIndexNames	SetRangeEnd
DeleteTable	GotoBookmark	SetRangeStart
DisableControls	GotoCurrent	UpdateRecord
Edit	GotoKey	

Events

AfterCancel	AfterPost	BeforeOpen
AfterClose	BeforeCancel	BeforePost
AfterDelete	BeforeClose	OnCalcFields
AfterEdit	BeforeDelete	OnNewRecord
AfterInsert	BeforeEdit	
AfterOpen	BeforeInsert	

TTabOrder type

Controls

Declaration

```
TTabOrder = -1..32767;
```

The *TTabOrder* type defines a subrange of integers that can be used as values for the *TabOrder* property.

TTabSet component

Tabs

The *TTabSet* component presents horizontal tabs users can click to initiate actions. Tab set controls are commonly used with *TNotebook* controls to display pages within the same dialog box.

You create a set of tabs for the tab set control when you specify a list of strings as the value of the *Tabs* property. One tab is created for each string. If you are using a tab set control to work with a notebook control (*TNotebook*), this line of code creates a tab for each page of the notebook control:

```
TabSet1.Tabs := Notebook1.Pages;
```

Then, in the *OnClick* event handler of the tab set control, this line of code changes the current page in the notebook control when the user clicks a tab:

```
Notebook1.PageIndex := TabSet1.TabIndex;
```

To determine which tab is currently selected or to use code to select a tab, use the *TabIndex* property. To find out which tab is the first visible tab in the tab set control or to make a tab the first visible tab, use the *FirstIndex* property.

Several properties affect the appearance of the tab set control. Tabs are usually displayed at the bottom of a form. To display the tabs at the bottom of the form, choose *alBottom* as the *Align* property value. Set the *SelectedColor* and *UnselectedColor* properties to help the user tell the difference between a selected and an unselected tab. The *BackgroundColor* and the *DitherBackground* properties determine how the background of the tab set appears. You can determine how far from the edge of the control the tabs are positioned with the *StartMargin* and *EndMargin* properties. If you want scroll buttons to appear automatically when the tab set control doesn't have enough room to display all the tabs, set *AutoScroll* to *True*.

You can display graphics on tabs as well as text. Use the *Style* property select an owner-draw tab style.

In addition to these properties, methods, and events, this component also has the properties, methods, and events that apply to all windowed controls.

For more information, search for TabSet component in online Help, and choose the topic Using the TabSet Component.

Properties

🔑	Align		Height	🔑	Style
🔑	AutoScroll		HelpContext	🔑	TabHeight
🔑	BackgroundColor		Hint	🔑	TabIndex
▷ 🔑	Canvas		Left		Tabs
▷	ComponentIndex		Name		Tag
	Cursor	▷	Owner		Top
🔑	DitherBackground	▷	Parent	🔑	UnselectedColor
	DragMode		ParentShowHint		Visible
	Enabled	🔑	SelectedColor	▷ 🔑	VisibleTabs
🔑	EndMargin		ShowHint		Width
🔑	FirstIndex	▷	Showing		
	Font	🔑	StartMargin		

Methods

	BeginDrag		GetTextLen		ScreenToClient
	BringToFront		Hide		ScrollBy
	CanFocus		Invalidate	🔑	SelectNext
	ClientToScreen	🔑	ItemAtPos		SendToBack
	Dragging	🔑	ItemRect		SetBounds

EndDrag	Refresh	SetFocus
Focused	Repaint	SetTextBuf
GetTextBuf	ScaleBy	Update

Events

OnChange	OnDragOver	OnEnter
OnClick	🔑 OnDrawTab	OnExit
OnDragDrop	OnEndDrag	🔑 OnMeasureTab

TTabStyle type

Tabs

Declaration

```
TTabStyle = (tsStandard, tsOwnerDraw);
```

The *TTabStyle* type defines the style of the tabs in a tab set control. *TTabStyle* is the type of the a tab set control's *Style* property.

TTextCase type

FileCtrl

Declaration

```
TTextCase = (tcLowerCase, tcUpperCase);
```

The *TTextCase* type defines the values available to the *Text* property of a drive combo box (*TDriveComboBox*).

TTextRec type

SysUtils

Declaration

```
PTextBuf = ^TTextBuf;
TTextBuf = array[0..127] of Char;
TTextRec = record
  Handle: Word;
  Mode: Word;
  BufSize: Word;
  Private: Word;
  BufPos: Word;
  BufEnd: Word;
  BufPtr: PTextBuf;
  OpenFunc: Pointer;
  InOutFunc: Pointer;
  FlushFunc: Pointer;
  CloseFunc: Pointer;
  UserData: array[1..16] of Byte;
  Name: array[0..79] of Char;
  Buffer: TTextBuf;
end;
```

TTextRec is the internal format of a variable of type text. You would never declare a variable of this type. However, you would use *TTextRec* to typecast a text file variable to access the internal data fields, such as the file name.

Note Do not use this unless you are familiar with writing Object Pascal text file device drivers.

TTileMode type **Forms**

Declaration

```
TTileMode = (tbHorizontal, tbVertical);
```

The *TTileMode* type defines the values the *TileMode* property of a form can have.

TTimeField component

A *TTimeField* represents a field of a record in a dataset. It represents a value consisting of a time.

Set the *DisplayFormat* property to control the formatting of the field for display purposes, and the *EditFormat* property for editing purposes. Use the *Value* property to access or change the current field value.

The *TTimeField* component has the properties, methods, and events of the *TField* component.

Properties

🔑 Alignment	▷ 🔑 DataType	▷ 🔑 IsIndexField
▷ 🔑 AsBoolean	🔑 DisplayLabel	▷ 🔑 IsNull
▷ 🔑 AsDateTime	▷ 🔑 DisplayName	Name
▷ 🔑 AsFloat	▷ 🔑 DisplayText	▷ Owner
▷ 🔑 AsInteger	🔑 DisplayWidth	🔑 ReadOnly
▷ 🔑 AsString	🔑 EditMask	🔑 Required
🔑 Calculated	▷ 🔑 EditMaskPtr	▷ 🔑 Size
▷ 🔑 CanModify	🔑 FieldName	Tag
▷ 🔑 DataSet	▷ 🔑 FieldNo	▷ 🔑 Text
▷ 🔑 DataSize	🔑 Index	🔑 Visible

Methods

🔑 Assign	🔑 FocusControl	🔑 SetData
🔑 AssignValue	🔑 GetData	
🔑 Clear	🔑 IsValidChar	

Events

☞ OnChange	☞ OnSetText	☞ OnValidate
☞ OnGetText		

TTimer component **ExtCtrls**

The *TTimer* component causes an *OnTimer* event to occur whenever a specified period of time passes. Within that *OnTimer* event handler, your code specifies what you want to happen each time the *OnTimer* event occurs.

You use the *Interval* property to control the amount of time between timer events.

To activate or deactivate a timer, use its *Enabled* property.

In addition to these properties and events, this component also has the properties and methods that apply to all components.

For more information, search for Timer component in online Help, and choose the topic Using the Timer Component.

Properties

ComponentIndex	☞ Interval	Owner
Enabled	Name	Tag

Events

☞ OnTimer

TTransIsolation type **DB**

Declaration

```
TTransIsolation = (tiDirtyRead, tiReadCommitted, tiRepeatableRead);
```

The *TTransIsolation* type is used by the *TransIsolation* property and it is the set of values that can be used to start a transaction. They control how records which have been modified by another application will be returned to your application by the server.

TVarBytesField component

A *TVarBytesField* represents a field of a record which is represented by a value consisting of an arbitrary set of up to 65535 bytes. The first two bytes are a binary value defining the actual length.

Use the *Assign* method to copy values from another field to a *TVarBytesField*.

The *TVarBytesField* component has the properties, methods, and events of the *TField* component.

Properties

🔑 Alignment	▷ 🔑 DataType	▷ 🔑 IsIndexField
▷ 🔑 AsBoolean	🔑 DisplayLabel	▷ 🔑 IsNull
▷ 🔑 AsDateTime	▷ 🔑 DisplayName	Name
▷ 🔑 AsFloat	▷ 🔑 DisplayText	▷ Owner
▷ 🔑 AsInteger	🔑 DisplayWidth	🔑 ReadOnly
▷ 🔑 AsString	🔑 EditMask	🔑 Required
🔑 Calculated	▷ 🔑 EditMaskPtr	🔑 Size
▷ 🔑 CanModify	🔑 FieldName	Tag
▷ 🔑 DataSet	▷ 🔑 FieldNo	▷ 🔑 Text
▷ 🔑 DataSize	🔑 Index	🔑 Visible

Methods

🔑 Assign	🔑 FocusControl	🔑 SetData
🔑 AssignValue	🔑 GetData	
🔑 Clear	🔑 IsValidChar	

Events

🔑 OnChange	🔑 OnSetText	🔑 OnValidate
🔑 OnGetText		

TVarRec type

System

Declaration

```
const
  vtInteger  = 0;
  vtBoolean  = 1;
  vtChar     = 2;
  vtExtended = 3;
  vtString   = 4;
  vtPointer  = 5;
  vtPChar    = 6;
  vtObject   = 7;
  vtClass    = 8;

type
  TVarRec = record
    case Integer of
      vtInteger:  (VInteger: Longint; VType: Byte);
      vtBoolean:  (VBoolean: Boolean);
      vtChar:     (VChar: Char);
```

```
        vtExtended: (VExtended: PExtended);
        vtString:   (VString: PString);
        vtPointer:  (VPointer: Pointer);
        vtPChar:    (VPChar: PChar);
        vtObject:   (VObject: TObject);
        vtClass:    (VClass: TClass);
    end;
```

TVarRec type is used inside a procedure with a parameter type of **array of const**. The tag field lets the procedure know the simple type of each parameter passed in the open array.

The variable type constants represent the values passed in the tag of the *TVarRec* structure.

TWindowState type **Forms**

Declaration

```
TWindowState = (wsNormal, wsMinimized, wsMaximized);
```

The *TWindowState* type defines the three possible states of a form: normal, minimized, or maximized. *TWindowState* is the type of the *WindowState* property of a form.

TWordArray **SysUtils**

Declaration

```
PWordArray = ^TWordArray;
```

```
TWordArray = array[0..16383] of Word;
```

TWordArray declares a general array of type *Word* that can be used in typecasting.

TWordField component

A *TWordField* represents a field of a record in a dataset. It is represented as a binary value with a range from 0 to 65,535. Use *TWordField* for fields that hold unsigned whole numbers.

Set the *DisplayFormat* property to control the formatting of the field for display purposes, and the *EditFormat* property for editing purposes. Use the *Value* property to access or change the current field value. Set the *MinValue* or the *MaxValue* property to limit the smallest or largest value permitted in a field.

The *TWordField* component has the properties, methods, and events of the *TField* component.

Properties

🔑 Alignment	🔑 DisplayFormat	▷ 🔑 IsNull
▷ 🔑 AsBoolean	🔑 DisplayLabel	🔑 MaxValue
▷ 🔑 AsDateTime	▷ 🔑 DisplayName	🔑 MinValue
▷ 🔑 AsFloat	▷ 🔑 DisplayText	Name
▷ 🔑 AsInteger	🔑 DisplayWidth	▷ Owner
▷ 🔑 AsString	🔑 EditFormat	🔑 ReadOnly
🔑 Calculated	🔑 EditMask	🔑 Required
▷ 🔑 CanModify	▷ 🔑 EditMaskPtr	▷ 🔑 Size
▷ 🔑 DataSet	🔑 FieldName	Tag
▷ 🔑 DataSize	▷ 🔑 FieldNo	▷ 🔑 Text
▷ 🔑 DataType	🔑 Index	▷ 🔑 Value
▷ 🔑 AsFloat	▷ 🔑 IsIndexField	🔑 Visible

Methods

🔑 Assign	🔑 FocusControl	🔑 SetData
🔑 AssignValue	🔑 GetData	
🔑 Clear	🔑 IsValidChar	

Events

🔑 OnChange	🔑 OnSetText	🔑 OnValidate
🔑 OnGetText		

TypeOf function — System

Declaration

```
function TypeOf(X) : Pointer
```

The *TypeOf* function returns a pointer to an object type's virtual method table (VMT).

X is either an object type identifier or an instance of an object type.

TypeOf can be applied only to object types that have a VMT; all other types result in an error.

Example

```
{Note: use TypeOf for the older "Object" object hierarchy}

uses WinCrt;

uses Objects;

type
  PBaseObject = ^TBaseObject;
  TBaseObject = object(TObject)
```

```
    end;

  PChildObject = ^TChildObject;
    TChildObject = object(TBaseObject)
    end;

  var
    P,Q : PObject;   { abstract object pointer }

  begin
    P := New(PBaseObject, Init);
    Q := New(PChildObject, Init);
    if TypeOf(P^) = TypeOf(TBaseObject) then
      writeln('P is a TBaseObject instance')
    else
      writeln('P is not a TBaseObject instance');

  if TypeOf(Q^) = TypeOf(TChildObject) then
      writeln('Q is a TChildObject instance')
    else
      writeln('Q is not a TChildObject instance');

  if TypeOf(Q^) <> TypeOf(P^) then
      writeln('Q is not the same kind of object instance as P');

  Dispose(P, Done);
    Dispose(Q, Done);
  end;
```

See also
SizeOf function

TZoomFactor type **ToCtrl**

Declaration

```
TZoomFactor = (z025, z050, z100, z150, z200);
```

TZoomFactor is the type of the *Zoom* property of the *TOLEContainer* component.

UniDirectional property

Applies to
TQuery component

Declaration

```
property UniDirectional: Boolean;
```

If an application only needs to be able to move forward in the result set of a *TQuery* component, set the *UniDirectional* property to *True*. When *UniDirectional* is *True*, an application requires less memory (because the records do not have to be cached), but the application cannot move backwards in the result set.

UniDirectional is *False* by default.

Unmerge method

Applies to

TMainMenu component

Declaration

```
procedure Unmerge(Menu: TMainMenu);
```

The *Unmerge* method reverses the merging of two menus into one in a non-MDI application. The *Menu* parameter is the merged menu that you no longer want to be merged.

Example

This example uses two forms, each containing a main menu created with the Menu Designer. It also uses a button on *Form1*. When the user clicks the button, *Form2* appears and the main menu of *Form2* merges with that of *Form1*.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Show;
  MainMenu1.Merge(Form2.MainMenu1);
end;
bo
```

Form2 also has a button. When the user clicks the button on *Form2*, the menu of *Form2* is no longer merged with the menu on *Form1*:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  Form1.MainMenu1.Unmerge(MainMenu1);
end;
```

To run this example, create a **uses** clause in the **implementation** section of each unit and add the other unit to it. For example, the **uses** clause in the **implementation** section of *Unit1* would look like this:

```
uses Unit2;
```

See also

AutoMerge property, *Merge* method

UnPrepare method

Applies to
TQuery, *TStoredProc* component

For stored procedures

Declaration

```
procedure UnPrepare;
```

The *UnPrepare* method notifies the server that the stored procedure will no longer be needed, allowing the server to release any resources allocated to the stored procedure.

Example

```
   StoredProc1.UnPrepare;
```

See also
Prepared property, *Prepare* method

For queries

Declaration

```
procedure UnPrepare;
```

The *UnPrepare* method sets the *Prepared* property to *False*. This ensures that the *SQL* property will be translated again before the request is submitted to the server. In addition, the server is notified that it can release any resources allocated for optimization purposes, since a new request will be sent before (or in conjunction with) a call to the *Open* or *ExecSQL* method.

Preparing a query consumes some database resources, so it is good practice for an application to unprepare a query once it is done using it. The *UnPrepare* method unprepares a query. When you change the text of a query at run time, Delphi automatically closes and unprepares the query.

See also
Prepare method

UnselectedColor property

Applies to
TTabSet component

Declaration

```
property UnselectedColor: TColor;
```

The *UnselectedColor* property determines the color of the tabs that aren't currently selected in the tab set control.

Example

This code changes the color of the unselected tabs:

```
   TabSet11.UnselectedColor := clSilver;
```

See also

SelectedColor property

UpCase function System

Declaration

```
function UpCase(Ch: Char): Char;
```

The *UpCase* function converts a *Ch* to uppercase.

Ch is an expression of type *Char*. Character values not in the range a..z are unaffected.

Example

```
uses Dialogs;

var
   s : string;
   i : Integer;
 begin
   { Get string from TEdit control }
   s := Edit1.Text;
   for i := 1 to Length(s) do
     s[i] := UpCase(s[i]);
   MessageDlg('Here it is in all uppercase: ' + s, mtInformation, [mbOk], 0);
 end;
```

See also

StrUpper function

Update method

Applies to

All controls; *TFieldDefs*, *TIndexDefs* objects; *TDirectoryListBox*, *TFileListBox* components

The *Update* method repaints or refreshes a component.

For directory and file list boxes

Declaration

```
procedure Update;
```

The *Update* method updates and refreshes the directory list for the directory and file list box controls.

Example

The following sample code sets the directory of *DirectoryListBox1* to C:\TEMP when the form is created. When *Button1* is pressed, a subdirectory called MYDIR is added to C:\TEMP, but note that it is not updated in *DirectoryListBox1* until *Button2* is pressed and *Update* is called.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MkDir('c:\temp\mydir');
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  DirectoryListBox1.Directory := 'c:\temp';
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  DirectoryListBox1.Update;
end;
```

For all controls

Declaration

```
procedure Update;
```

The *Update* method calls the Windows API *UpdateWindow* function, which processes any pending paint messages.

Example

When this line of code runs, Windows repaints *EditBox1*:

```
Edit1.Update;
```

See also

Invalidate method, *Refresh* method

For TFieldDefs objects

Declaration

```
procedure Update;
```

Update refreshes the *TFieldDef* entries in *Items* to reflect the current state of the fields underlying the dataset. It does so without opening the dataset.

For TIndexDefs objects

Declaration

```
procedure Update;
```

The *Update* method will refresh the entries in *Items* to reflect the current dataset. Use this method to obtain index information without opening the dataset.

See also

Items property

UpdateCursorPos method

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure UpdateCursorPos;
```

UpdateCursorPos sets the current position of the dataset's underlying BDE cursor to the current cursor position of the dataset. The *UpdateCursorPos* method is useful if you make direct calls to the Borland Database Engine.

See also

CursorPosChanged method

UpdateMode property

Applies to

TTable, *TQuery* components (live result sets only)

Declaration

```
property UpdateMode;
```

The *UpdateMode* property determines how Delphi will find records being updated in a SQL database. This property is important in a multi-user environment when users may retrieve the same records and make conflicting changes to them.

When a user posts an update, Delphi uses the original values in the record to find the record in the database. This approach is similar to an optimistic locking scheme. *UpdateMode* specifies which columns Delphi uses to find the record. In SQL terms, *UpdateMode* specifies which columns are included in the WHERE clause of an UPDATE statement. If Delphi cannot find a record with the original values in the columns specified (if another user has changed the values in the database), Delphi will not make the update and will generate an exception.

The *UpdateMode* property may have the following values:

- *WhereAll* (the default): Delphi uses every column to find the record being updated. This is the most restrictive mode.
- *WhereKeyOnly*: Delphi uses only the key columns to find the record being updated. This is the least restrictive mode and should be used only if other users will not be changing the records being updated.
- *WhereChanged*: Delphi uses key columns and columns that have changed to find the record being updated.

Example

Consider a COUNTRY table with columns for NAME (the key), CAPITAL, and CONTINENT. Suppose you and another user simultaneously retrieve a record with the following values:

- NAME = “Philippines”
- CAPITAL = “Nairobi”
- CONTINENT = “Africa”

Both you and the other user notice that the information in this record is incorrect and should be changed. Now, suppose the other user changes CONTINENT to “Asia”, CAPITAL to “Manila”, and posts the change to the database. A few seconds later, you change NAME to “Kenya” and post your change to the database.

If your application has *UpdateMode* set to *WhereKey* on the dataset, Delphi compares the original value of the key column (NAME = “Philippines”) to the current value in the database. Since the other user did not change NAME, your update occurs. You think the record is now [“Kenya”, “Nairobi”, “Africa”] and the other users thinks it is [“Philippines”, “Asia”, “Manila”]. Unfortunately, it is actually [“Kenya”, “Asia”, “Manila”], which is still incorrect, even though both you and the other user think you have corrected the mistake. This problem occurred because you had *UpdateMode* set to its least restrictive level, which does not protect against such occurrences.

If your application had *UpdateMode* set to *WhereAll*, the Delphi would check all the columns when you attempt to make your update. Since the other user changed CAPITAL and CONTINENT, Delphi would not let you make the update. When you retrieved the record again, you would see the new values entered by the other user and realize that the mistake had already been corrected.

UpdateRecord method

Applies to

TTable, *TQuery*, *TStoredProc* components

Declaration

```
procedure UpdateRecord;
```

The *UpdateRecord* method notifies each *TDataSource* component that the current record is about to be posted to the dataset. Each data source in turn notifies all data controls so that they can update the fields of the record from the current values displayed in the controls. *UpdateRecord* is called automatically by *Post*, but an application can also use it separately to bring the current record up to date without posting it.

UpperCase function SysUtils

Declaration

```
function UpperCase(const S: string): string;
```

The *UpperCase* function returns a string containing the same text as *S*, but with all letters converted to uppercase.

Example

This example uses a list box and a button on a form. Use the Items property editor in the Object Inspector to enter a list of strings in the list box. When you run the application and click the button, the strings in the list box become uppercase.

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

See also

AnsiUpperCase function, *LowerCase* function

Val procedure System

Declaration

```
procedure Val(S; var V; var Code: Integer);
```

The *Val* function converts the string value *S* to its numeric representation, as if it were read from a text file with *Read*.

S is a string-type expression; it must be a sequence of characters that form a signed whole number. *V* is an integer-type or real-type variable. *Code* is a variable of type *Integer*.

If the string is invalid, the index of the offending character is stored in *Code*; otherwise, *Code* is set to zero. For a null-terminated string, the error position returned in *Code* is one larger than the actual zero-based index of the character in error.

Val performs range checking differently depending upon the state of **$R** and the type of the parameter *V*.

Setting	Result
{$R+}	An out-of-range value always generates a run-time error.
{$R–}	The values for out-of-range vary depending upon the data type of *V*.

Example

```
uses Dialogs;

var
  I, Code: Integer;
begin
  { Get text from TEdit control }
  Val(Edit1.Text, I, Code);
  { Error during conversion to integer? }
  if code <> 0 then
    MessageDlg('Error at position: ' + IntToStr(Code), mtWarning, [mbOk], 0);
  else
    Canvas.TextOut(10, 10, 'Value = ' + IntToStr(I));
  Readln;
end;
```

See also
Str procedure

ValidateEdit method

Applies to
TDBEdit, *TMaskEdit* components

Declaration

```
procedure ValidateEdit;
```

The *ValidateEdit* method checks the value of the *EditText* property for blank required characters in the edit box. If one is found, the *EDBEditError* exception is raised. If no exception occurs, you can be sure all required characters have been entered in the edit box.

Example

The following code calls *ValidateEdit* from the default *OnExit* event handler of *DBEdit1*.

```
procedure TForm1.DBEdit1Exit(Sender: TObject);
begin
  ValidateEdit;
end;
```

See also

EditText property

ValidParentForm function

Forms

Declaration

```
function ValidParentForm(Control: TControl): TForm;
```

The *ValidParentForm* function returns the form that contains the control specified in the *Control* parameter. If the specified control is not on a form, *ValidParentForm* generates an *EInvalidOperation* exception.

If you prefer that the function return **nil** when the specified control is not on a form, use the *GetParentForm* function.

Example

The following code calls *ValidParentForm* to find the parent form of the *MyForm.MyButton* control. If found, that form is shown modally. If not, an exception occurs.

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  ValidParentForm(MyForm.MyButton).ShowModal;
end;
```

Value property

Applies to

TDBLookupCombo, *TDBLookupList*, *TDBRadioGroup*, *TBCDField*, *TBooleanField*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TIntegerField*, *TSmallintField*, *TStringField*, *TTimeField*, *TWordField* components

For database radio groups

Declaration

```
property Value: string;
```

The value of the *Value* property is the current contents of the field for the current record in the dataset. When the user selects a radio button, the value of the *Value* property changes to the value of the *Items* string for the radio button. The new value of the *Value* property becomes the value of the field for the current record in the dataset.

If the *ReadOnly* property of the database radio group box is *True*, the user won't be able to select a button or change the contents of the field.

Example

The following code concatenates some text to the string in the *Value* property. When the user chooses a radio button in *DBRadioGroup1*, the value of *Value* is stored in the corresponding field in the *DataSource* (if *ReadOnly* is *False*). In the *OnChange* event handler, a label is updated, indicating to the user that this change has occurred.

```
procedure TForm1.DBRadioGroup1Change(Sender: TObject);
begin
  Label1.Caption := 'Field ' + DBRadioGroup1.DataField + ' has changed to ' +
    DBRadioGroup1.Value;
end;
```

See also

ItemIndex property

For database lookup combo and list boxes

Declaration

```
property Value: string;
```

Run-time only. The value of the *Value* property is the contents of the *DataField* for the current record in the primary dataset. As the user moves through the primary dataset, the value of the *Value* property changes.

By explicitly changing the *Value* property value at run time, you change the contents of the field.

Example

The following code changes the *Value* property, and thus, the value of the field in the connected dataset to 'Green'.

```
DBLookupCombo1.Value := 'Green';
```

See also

DisplayValue property

For fields

Declaration

```
property Value: string;      {TStringField}
property Value: Longint;     {TIntegerField, TSmallintField, TWordField}
property Value: Double;      {TBCDField, TCurrencyField, TFloatField}
property Value: Boolean;     {TBooleanField}
property Value: TDateTime    {TDateField, TDateTimeField, TTimeField}
```

Run-time only. *Value* is the actual data in a *TField*. Use *Value* to read data directly from and write data directly to a *TField*.

For *TBCDField*, *TCurrencyField*, and *TFloatField*, *Value* is a *Double*.

For *TBooleanField*, *Value* is a *Boolean*.

For *TDateField*, *TDateTimeField*, and *TTimeField*, *Value* is a *TDateTime*.

For *TIntegerField*, *TSmallintField*, and *TWordField*, *Value* is a *Longint*.

For *TStringField*, *Value* is the string assigned to the field.

Examples

```
StringField1.Value := 'Delphi';
DateField1.Value := StrToDateTime('02/14/95 00:00:00');
```

ValueChecked property

Applies to
TDBCheckBox component

Declaration

```
property ValueChecked: string;
```

Description
If the value of the *ValueChecked* property is equal to the data in the field of the current record of the dataset, the database check box is checked.

You also can enter a semicolon-delimited list of items as the value of *ValueChecked*. If any of the items matches the contents of the field of the current record in the dataset, the check box is checked. For example, you can specify a *ValueChecked* string like this:

```
DBCheckBox1.ValueChecked := 'True;Yes;On';
```

If the string True, Yes, or On is the contents of the field specified as the database check box's *DataField*, the check box is checked. The case of the specified strings is not checked.

If the contents of the field of the current record matches a string specified as the value of the *ValueUnchecked* property, the check box is unchecked. If the contents of the field matches no string in either *ValueChecked* or *ValueUnchecked*, the check box appears gray.

If the *DataField* of the database check box is a logical field, the check box is always checked if the contents of the field is *True*, and it is always unchecked if the contents of the field is *False*. The values of the *ValueChecked* and *ValueUnchecked* properties have no affect on logical fields.

If the user checks a database check box, the string that is the value of the *ValueChecked* property is placed in the database field, as long as the *ReadOnly* property is *False*. If the value is a semicolon-delimited list of items, the first item in the list is inserted as the contents of the field of the current record.

The default value of *ValueChecked* is the string 'True'.

Example

The following code toggles the value of the *ValueChecked* property of *DBCheckBox1* from 'True' to 'False' or from 'False' to 'True'.

```
with DBCheckBox1 do
if (ValueChecked = 'True') or (ValueChecked = 'False') then
  if ValueChecked = 'True' then ValueChecked := 'False'
  else ValueChecked := 'True';
```

See also

ValueUnchecked property

Values property

The *Values* property is used by string and string list objects, and by database radio group boxes.

For string and string list objects

Applies to

TStrings, *TStringList* objects

Declaration

```
property Values[const Name: string]: string;
```

The *Values* property gives you access to a specific string in a list of strings. The strings must have a unique structure before you can use the *Values* property array to access them:

```
Name=Value
```

The *Name* that identifies the string is to the left of the equal sign (=), and the current *Value* of the *Name* identifier is on the right side of the equal sign. There should be no spaces present before and after the equal sign.

Such strings are commonly found in .INI files. For example, here are a few strings taken from a DELPHI.INI file:

```
DisplayGrid=1
```

```
SnapToGrid=1
GridSizeX=8
GridSizeY=8
```

The strings that make up the *Params* property of a database component (*TDatabase*) have the same format. The most common use of the *Values* property is to modify a string within the *Params* property array.

To modify a string in a list of strings that have the required format, identify the string to modify with the *Name* constant parameter, which serves as an index into the list of strings, and assign a new value.

Example

Assume that a string that identifies the password needed to access a database exists in the *Params* string list. You can change the acceptable password using this code:

```
Database1.Params.Values['Password'] := 'TopSecret';
```

If there is no password string, the same code creates one at the bottom of the list of strings and assigns the 'TopSecret' string as its value.

You can also assign the value of the string to a variable. For example, this code assigns the current value of the password string to a variable called *StringValue*:

```
var
  StringValue: string;

StringValue := Database1.Params.Values['Password'];
```

For database radio group boxes

Applies to

TDBRadioGroup component

Declaration

```
property Values: TStrings;
```

Each string in the *Items* property for a database radio group box places a radio button in the group box with an accompanying caption. If the contents of a field for the current record is the same as one of the strings in *Items*, the corresponding radio button is selected. If the user selects one of the radio group buttons and the *ReadOnly* property of the database radio group is *False*, the contents of the field changes to the corresponding *Items* string.

Often, you might not want the same string that serves as the caption of a radio button to become the contents of the field. Or, you might want a different value in the data field (other than the caption of a radio button) to select a radio button. In this case, use the *Values* property. You can specify a string in the *Value* property for each string in the *Items* list. The first string in *Values* corresponds to the first string in the *Items*, and therefore, the first radio button in the group box.

For example, suppose you have two strings in the *Items* property for a database radio group: Yes and No. If there are no strings in the *Values* property, the data field must contain either the value Yes or No to select one of the radio buttons. If the user selects one of these buttons, the string Yes or No becomes the contents of the data field.

If the data field contains values such as Y or N, rather than Yes or No, you can specify Y or N as *Values* strings. This way, the Yes or No radio buttons are selected when a Y or N value appears in the data field. When the user selects one of the radio buttons, Y or N becomes the value of the field of the current record.

Example

This example uses a database radio group box connected to field in a dataset. The field contains the values 'Y', 'N', or 'M'. You want the captions of the radio buttons to be 'Yes', 'No', or 'Maybe', so the code adds these three strings to the *Items* property array. The actual values that are in the field and that can entered into the field are the 'Y', 'N', and 'M' strings, so these are added to the *Values* property array.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  with DBRadioGroup1 do
  begin
    Items.Add('Yes');
    Items.Add('No');
    Items.Add('Maybe');
    Values.Add('Y');
    Values.Add('N');
    Values.Add('M');
  end;
end;
```

When the code runs, three radio buttons appear in the group box. If the current record in the dataset contains any of the values contained in the *Values* property, the appropriate radio button is checked. When the user selects a radio button, the corresponding string in the *Values* property is entered into the field.

See also

ItemIndex property, *Items* property

ValueUnchecked property

Applies to

TDBCheckBox component

Declaration

```
property ValueUnchecked: string;
```

If the value of the *ValueUnchecked* property is equal to the data in the field of the current record of the dataset, the database check box is unchecked.

You also can enter a semicolon-delimited list of items as the value of *ValueUnchecked*. If any of the items matches the contents of the field of the current record in the dataset, the check box is unchecked. For example, you can specify a *ValueUnchecked* string like this:

```
DBCheckBox1.ValueUnchecked := 'False;No;Off';
```

If the string False, No, or Off is the contents of the field specified as the database check box's *DataField*, the check box is unchecked.

If the contents of the field of the current record matches a string specified as the value of the *ValueChecked* property, the check box is checked. If the contents of the field matches no string in either *ValueChecked* or *ValueUnchecked*, the check box appears gray.

If the *DataField* of the database check box is a logical field, the check box is always checked if the contents of the field is *True*, and it is always unchecked if the contents of the field is *False*. The values of the *ValueChecked* and *ValueUnchecked* properties have no affect on logical fields.

If the user checks a database check box, the string that is the value of the *ValueUnchecked* property is placed in the database field, as long as the *ReadOnly* property is *False*. If the value is a semicolon-delimited list of items, the first item in the list is inserted as the contents of the field of the current record.

The default value of *ValueUnchecked* is the string 'False'.

Example

The following code changes *ValueUnchecked* to 'NO'. When the value of the linked field is 'NO', *DBCheckBox1* is unchecked.

```
DBCheckBox1.ValueUnchecked := 'NO';
```

See also

ValueChecked property

VersionMajor property

Applies to

TReport component

Declaration

```
property VersionMajor: Integer;
```

Run-time and read only. The value of the *VersionMajor* property identifies which major version of ReportSmith you are running. For example, if you are using ReportSmith 2.5, the value of *VersionMajor* is 2. The minor version value is reported in the *VersionMinor* property.

See also

VersionMinor property

VersionMinor property

Applies to

TReport component

Declaration

```
property VersionMajor: Integer;
```

Run-time and read only. The value of the *VersionMinor* property identifies which minor version of ReportSmith you are running. For example, if you are using ReportSmith 2.5, the value of *VersionMinor* is 5. The major version value is reported in the *VersionMajor* property.

See also

VersionMajor property

VertScrollBar property

Applies to

TForm, *TScrollBox* components

Declaration

```
property VertScrollBar: TControlScrollBar;
```

The *VertScrollBar* property is the form's or scroll box's vertical scroll bar. The values of *VertScrollBar's* nested properties determines how the vertical scroll bar behaves.

To make a vertical scroll bar appear on a form or scroll box, these nested properties of *VertScrollBar* must be set like this:

- *Visible* must be *x*.
- The value of the *Range* property must be greater than the value of the *ClientHeight* property of the form or the *Height* property of the scroll box.

Example

This example places a vertical scroll bar on the form, as long as the *ClientHeight* of the form is not greater than 500:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  with VertScrollBar do
  begin
    Range := 500;
    Visible := True;
  end;
end;
```

See also
HorzScrollBar property, *Increment* property, *Position* property, *Range* property, *ScrollPos* property, *Visible* property

Visible property

Applies to
All controls; *TBCDField*, *TBlobField*, *TBooleanField*, *TBytesField*, *TControlScrollBar*, *TCurrencyField*, *TDateField*, *TDateTimeField*, *TFloatField*, *TForm*, *TGraphicField*, *TIntegerField*, *TMenuItem*, *TMemoField*, *TSmallintField*, *TStringField*, *TTimeField*, *TVarBytesField*, *TWordField* components

Declaration

```
property Visible: Boolean;
```

The *Visible* property determines whether the component appears onscreen. If *Visible* is *True*, the component appears. If *Visible* is *False*, the component is not visible.

For controls, calling the *Show* method makes the control's *Visible* property *True*, but it also performs other actions to ensure that the user can see the control.

For field components, the *Visible* property determines if a field can be displayed in a *TDBGrid* component. If *Visible* is *False*, the field is not displayed.

The default value is *True* for all components except for forms.

Example
The following code shows how to make a button invisible:

```
  Button1.Visible := False;
```

See also
Hide method, *HorzScrollBar* property, *Show* method, *VertScrollBar* property

VisibleButtons property

Applies to
TDBNavigator, *TMediaPlayer* components

The *VisibleButtons* property determines which buttons of a component are visible, and therefore, which operations the user can perform.

Declaration

```
property VisibleButtons: TButtonSet;
```

The *VisibleButtons* property determines which of the buttons on the media player are visible. If a button is not made visible with *VisibleButtons*, it does not appear on the media player control. By default, all buttons are visible when a media player component is added to a form.

Button	Value	Action
Play	*btPlay*	Plays the media player
Record	*btRecord*	Starts recording
Stop	*btStop*	Stops playing or recording
Next	*btNext*	Skips to the next track, or to the end if the medium doesn't use tracks
Prev	*btPrev*	Skips to the previous track, or to the beginning if the medium doesn't use tracks
Step	*btStep*	Moves forward a number of frames
Back	*btBack*	Moves backward a number of frames
Pause	*btPause*	Pauses playing or recording. If already paused when clicked, resumes playing or recording.
Eject	*btEject*	Ejects the medium

Example

The following line of code causes only the Play and Stop buttons of*MediaPlayer1* to be displayed:

```
MediaPlayer1.VisibleButtons := [btPlay, btStop];
```

See also

ColoredButtons property, *EnabledButtons* property

For database navigator controls

Declaration

```
property VisibleButtons: TButtonSet;
```

The value of the *VisibleButtons* property determines which buttons appear on the database navigator component. By default, all the buttons are visible. By changing the value of the *VisibleButtons* set, you can hide some of the buttons, and therefore, prevent the user from performing certain operations. For example, if you only want the user to view the records in the dataset, you would include only the*nbFirst*, *nbPrior*, *nbNext*, and *nbLast* values in the *VisibleButtons* set.

Button	Value	Action
First	*nbFirst*	Go to the first record
Prior	*nbPrior*	Go to the previous record
Next	*nbNext*	Go to the next record
Last	*nbLast*	Go to the last record
Insert	*nbInsert*	Insert a blank record

Button	Value	Action
Delete	*nbDelete*	Deletes the current record
Edit	*nbEdit*	Permits editing of the current record
Post	*nbPost*	Posts the current record
Cancel	*nbCancel*	Cancels the current edit
Refresh	*nbRefresh*	Refreshes the data in the dataset

The default value is all of these values in the *Options* set.

Example

The following line of code displays only the Prior and Next buttons of *DBNavigator1*:

```
DBNavigator1.VisibleButtons := [nbPrior, nbNext];
```

VisibleColCount property

Applies to

TDrawGrid, *TStringGrid* components

Declaration

```
function VisibleColCount: Integer;
```

Run-time and read only. The *VisibleColCount* contains the number of columns, other than fixed or nonscrolling columns, that are fully displayed in the grid. If another column is partially displayed in the grid, it won't be part of the count.

Example

This example uses a draw grid, two labels, and a button on a form. When the user clicks the button, the number of rows and columns, excluding partial and fixed ones, are reported in the captions of the two labels:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := IntToStr(DrawGrid1.VisibleRowCount) + ' rows';
  Label2.Caption := IntToStr(DrawGrid1.VisibleColCount) + ' columns';
end;
```

See also

ColCount property, *VisibleRowCount* property

VisibleRowCount property

Applies to

TDrawGrid, *TStringGrid* components

Declaration

```
function VisibleRowCount: Integer;
```

Run-time and read only. The *VisibleRowCount* contains the number of rows, other than fixed or nonscrolling rows, that are fully displayed in the grid. If another row is partially displayed in the grid, it won't be part of the count.

Example

This example uses a draw grid, two labels, and a button on a form. When the user clicks the button, the number of rows and columns, excluding partial and fixed ones, are reported in the captions of the two labels:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := IntToStr(DrawGrid1.VisibleRowCount) + ' rows';
  Label2.Caption := IntToStr(DrawGrid1.VisibleColCount) + ' columns';
end;
```

See also

RowCount property, *VisibleColCount* property

VisibleTabs property

Applies to

TTabSet component

Declaration

```
property VisibleTabs: Integer;
```

Read only. The value of the *VisibleTabs* property contains the number of tabs currently visible in the tab set control.

Example

This example queries the *VisibleTabs* property to find out how many tabs are visible in the tab set control and assigns the number to a variable:

```
SeeTabs := TabSet11.VisibleTabs;
```

Wait property

Applies to

TMediaPlayer component

Declaration

```
property Wait: Boolean;
```

The *Wait* property determines whether a media control method (*Back, Close, Eject, Next, Open, Pause, PauseOnly, Play, Previous, StartRecording, Resume, Rewind, Step*, or *Stop*) returns control to the application only after it has been completed. *Wait* is unavailable at design time.

If *Wait* is *True*, the media player component waits until the next media control method has completed before returning control to the application. If *Wait* is *False*, the application won't wait for the next media control method to finish before continuing.

Wait affects only the next media control method called after setting *Wait*. You must reset *Wait* to affect any subsequent call to a media control method.

By default, *Play* and *StartRecording* function as if *Wait* is *False*. You must set *Wait* to *True* before calling *Play* or *StartRecording* to prevent control from returning to the application before playing or recording has finished. By default, all other media control methods function as if *Wait* is *True*.

Note Usually you would set *Wait* to *False* only if the next media control is expected to take a long time, so that your application can execute other code before the media control method has completed. If you set *Wait* to *False*, you might want to set *Notify* to *True* so the application is notified when the media control method completes.

Example

The following code plays a .WAV audio file named NI!.WAV twice. The first call to *Play* doesn't return control to the application until the file is done playing. Note that if you remove the line of code that sets wait to true, the sound is only played once.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with MediaPlayer1 do begin
    FileName := 'ni!.wav';
    AutoRewind := True;
    try
      Open;             { Open Media Player }
      Wait := True;     { Waits until sounds is done playing to return }
      Play;             { Play sound }
      Play;             { Play again }
    finally
      Close;            { Close media player }
    end;
  end;
end;
```

WantReturns property

Applies to

TDBMemo, *TMemo* components

Declaration

```
property WantReturns: Boolean;
```

The *WantReturns* property determines whether return characters the user enters in the memo by pressing *Enter* affect the text in the memo, or go to the form. If *WantReturns* is *True* and the user presses *Enter*, a return character is entered in the memo. If *WantReturns* is *False* and the user presses *Enter*, a return is not entered in the memo, but instead goes to the form. For example, if there is a default button on a form, pressing *Enter* would choose the button instead of affecting the memo's text.

To enter return characters in a memo when *WantReturns* is *False*, press *Ctrl+Enter*.

Example

This example uses a memo and a check box on a form. If the check box is checked, the user can enter return characters into text entered in the memo. If the check box is unchecked, return characters aren't entered into the memo, but go to the form.

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if CheckBox1.Checked then
    Memo1.WantReturns := True
  else
    Memo1.WantReturns := False;
end;
```

See also

KeyPreview property, *WantTabs* property, *WordWrap* property

WantTabs property

Applies to

TDBMemo, *TMemo* components

Declaration

```
property WantTabs: Boolean;
```

The *WantTabs* property determines if tabs are enabled in a memo control. To enable tabs in a memo control, set *WantTabs* to *True*. To turn tabs off, set *WantTabs* to *False*.

Caution If *WantTabs* is *True*, the user can't use the *Tab* key to select the next control on the form. The user can tab into a memo control, but can't tab out.

Example

This example uses a memo and a check box on a form. When the check box is checked, the user can enter tab characters into the memo's text. When the check box is unchecked, the user can't enter tab characters into the text, but can use the *Tab* key to move between the memo and the check box controls.

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if CheckBox1.Checked then
    Memo1.WantTabs := True
  else
    Memo1.WantTabs := False;
end;
```

See also

KeyPreview property, *TabStop* property, *WantReturns* property, *WordWrap* property

WhereX function WinCrt

Declaration

```
function WhereX: Byte;
```

Return value

The *WhereX* function returns the CP's *X*-coordinate of the current cursor location.

The returned value is 1-based, and it corresponds to *Cursor.X + 1*.

Example

```
  uses WinCrt;

  begin
    Write('The number in this sentence is in the #');
    Writeln(WhereX, ' column in this window.');
  end;
```

See also

GotoXY procedure, *WhereY* function

WhereY function WinCrt

Declaration

```
function WhereY: Byte;
```

The *WhereY* function returns the CP's *Y*-coordinate of the current cursor location.

The returned value is 1-based, and it corresponds to *Cursor.Y + 1*.

Example

```
  uses WinCrt;

  begin
    Writeln;
    Writeln;
    Write('This sentence is on the #');
    Writeln(WhereY, ' line in this window.');
  end;
```

See also
GoToXY procedure, *WhereX* function

Width property

Applies to
All controls; *TBitmap*, *TGraphic*, *TIcon*, *TMetafile*, *TPen*, *TPicture* objects; *TForm*, *TScreen* components

Declaration

```
property Width: Integer;
```

The *Width* property determines horizontal size.

For forms and controls

The *Width* property determines the horizontal size of the control or form in pixels. When you increase the *Width* property value, the form or control becomes wider. If you decrease the value, the form or control becomes narrower.

Example
The following code doubles the width of a button:

```
   Button1.Width := Button1.Width * 2;
```

See also
ClientWidth property, *Height* property, *SetBounds* method

For graphic objects

Declaration

```
property Width: Integer;
```

The *Width* property determines the maximum width of the graphics object in pixels.

Example
To set the pen width to a random value from 1 to 10,

```
   Canvas.Pen.Width := 1 + Random(10);
```

See also
Height property

For screen components

Declaration

```
property Width: Integer;
```

Run-time and read only. The *Width* property is the horizontal size of the screen device in pixels.

Example

The following component determines the width of all the forms on the screen and resizes the ones that are wider than the screen width. To run this code, the integer variable *I* must be declared.

```
with Screen do
  for I := 0 to FormCount-1 do
  if Forms[I].Width > Width than Forms[I].Width := Width;
```

See also

Height property

WindowMenu property

Applies to

TForm component

Declaration

```
property WindowMenu: TMenuItem;
```

Most Windows MDI applications contain a Window menu that contains menu items such as Cascade, Arrange Icons, Tile, and so on that let the user manage the windows in the application. Usually this menu lists (at the bottom) the child windows that are currently open in the application. When the user selects one of these windows from the menu, the window becomes the active window in the application.

The *WindowMenu* property determines which menu includes the open child windows (or forms) in your application. Although this menu is commonly called the Window menu, it can be any name of your choosing. It must be an existing menu item, however, and it should be one of the menu items that appears in the menu bar or the child forms won't be included in a menu.

Example

For this code to run, a menu item called *MyWindows* must exist on an MDI form parent form. This line of code designates the *MyWindows* menu to be the Window menu, the menu that lists all open child windows in an MDI application:

```
WindowMenu := MyWindows;
```

See also
FormStyle property, *TMenuItem* component

WindowOrg typed constant **WinCrt**

Declaration

```
const WindowOrg: TPoint = (X: cw_UseDefault; Y: cw_UseDefault);
```

The *WindowOrg* typed constant determines the initial location of the CRT window.

The default location enables Windows to select a suitable location for the CRT window.

You can change the initial location by assigning new values to the x- and y-coordinates before the CRT window is created.

WindowSize typed constant **WinCrt**

Declaration

```
const WindowSize: TPoint = (X: cw_UseDefault; Y: cw_UseDefault);
```

The *WindowSize* typed constant determines the initial size of the CRT window.

The default size enables Windows to select a suitable size for the CRT window.

You can change the initial size by assigning new values to the x- and y-coordinates before the CRT window is created.

WindowState property

Applies to
TForm component

Declaration

```
property WindowState: TWindowState
```

The *WindowState* determines the initial state of the form. These are the possible values:

Value	Meaning
wsNormal	The form appears neither maximized nor minimized
wsMaximized	The form is maximized
wsMinimized	The form is minimized

The default value is *wsNormal*.

Example

The following code responds to the user clicking a button named Shrink by minimizing the form:

```
procedure TForm1.ShrinkClick(Sender: TObject);
begin
  WindowState := wsMinimized;
end;
```

See also

TWindowState type

WindowTitle variable — WinCrt

Declaration

```
var WindowTitle: array[0..79] of Char;
```

The *WindowTitle* variable determines the title of the CRT window.

The default value is the full path of the program's .EXE file.

You can change the title by storing a new string in *WindowTitle* before the CRT window is created.

Here is an example:

```
StrCopy(WindowTitle, 'Hello World');
```

WordRec — SysUtils

Declaration

```
WordRec = record
  Lo, Hi: Byte;
end;
```

WordRec declares a utility record that stores the high and low order bytes of the specified variable as type *Byte*.

See also

Hi function, *Lo* function

WordWrap property

Applies to

TDBMemo, *TDBText*, *TLabel*, *TMemo* components

Declaration

```
property WordWrap: Boolean;
```

The *WordWrap* property determines if text in a label or memo control wraps at the right margin so that it fits in the control. You can give the user access to the lines which aren't visible in a memo control by setting its *ScrollBars* property to add horizontal, vertical, or both scrollbars to the memo control. There should be no reason to use a horizontal scroll bar if *WordWrap* is *True*.

The memo control must be tall enough to display at least one line of text to allow the user to edit its contents, even if *WordWrap* is *True*.

The default value is *False*.

Example

This example allows text a user enters in the *Memo1* control to wrap to the next line, if the control is large enough to hold the text:

```
  Memo1.WordWrap := True;
```

See also

ScrollBars property, *Text* property, *AutoSize* property

Write method

Applies to

TBlobStream object

Declaration

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

The *Write* method copies up to *Count* bytes from *Buffer* to the current position in the field. *Buffer* must have at least *Count* bytes allocated for it. *Write* returns the number of bytes transferred (which may be less than the number requested in *Count*.) Transfers which require crossing a selector boundary in the source will be handled correctly.

Example

```
  BlobStream1.Write(Buf, 4096);
```

See also

TBlobField component, *TBytesField* component, *TVarBytesField* component

Write procedure

System

Declaration

Text files:

```
procedure Write( [ var F: Text; ] P1 [,P2,...,Pn ] );
```

Typed files:

```
procedure Write(F, V1 [V2,...Vn]);
```

The *Write* procedure writes values to a file.

F specifies a text file variable, which must be open for output. If *F* is omitted, the standard file variable *Output* is assumed.

For text files

Each *P* is a *Write parameter* that includes an output expression whose value is to be written to the file. A Write parameter can also contain the specifications of field width, and number of decimal places.

Each output expression must be of type *Char*, *Integer*, *Real*, string, packed string, or *Boolean*.

For typed files

Each *V* is a variable of the same type as the component type of *F*.

For each variable written, the current file position is advanced to the next component.

If the current file position is at the end of the file, the file is expanded.

{$I+} lets you handle run-time errors using exceptions. For more information on handling run-time library exceptions, see Handling RTL Exceptions in the Help system.

If you are using **{$I–}**, you must use *IOResult* to check for I/O errors.

See also

Read procedure, *Readln* procedure, *Writeln* procedure

WriteBool method

Applies to

TIniFile object

Declaration

```
procedure WriteBool(const Section, Ident: string; Value: Boolean);
```

The *WriteBool* method writes a Boolean value in an .INI file.

The *Section* constant identifies the section of the .INI file where the value is written. For example, the WIN.INI for Windows contains a [Desktop] section.

The *Ident* parameter is the name of the identifier for which you want to change the value. The *Value* parameter contains the new value.

Example

This example creates an .INI file for a game with two entries in the Options section, and one entry in the Configuration section.

Before you try this example, you must add *IniFiles* to the **uses** clause of your unit.

```
var
  GameIni: TIniFile;
begin
  Gamini := TIniFile.Create('FUNGAME.INI');
  with GameIni do
  begin
    WriteBool('Options', 'Sound', True);
    WriteInteger('Options', 'Level', 3);
    WriteString('Configuration', 'Name', 'Teresa Ace');
    Free;
  end;
end;
```

See also

ReadBool method, *WriteInteger* method, *WriteString* method

WriteBuf procedure — WinCrt

Declaration

```
procedure WriteBuf(Buffer: PChar; Count: Word);
```

The *WriteBuf* procedure writes a block of characters to the CRT window.

Buffer points to the first character in the block. *Count* contains the number of characters to write.

If the value of the *AutoTracking* typed constant is *True*, the CRT window scrolls if necessary to ensure that the cursor is visible after writing the block of characters.

Example

```
uses WinCrt;

var
  MyBuffer: PChar;

begin
  GetMem(MyBuffer, 80);
  MyBuffer := 'This is an example of WriteBuf';
  WriteBuf(MyBuffer, 30);
```

```
   end;
```

See also

AutoTracking typed constant, *WriteChar procedure*

WriteChar procedure — WinCrt

Declaration

```
procedure WriteChar(Ch: Char);
```

The *WriteChar* writes the character *Ch* to the *WinCrt* window at the current cursor position by calling *WriteBuf*(*@Ch, 1*).

Example

```
   uses WinCrt;

   begin
     Write('ABCDE');
     WriteChar('F');
   end;
```

See also

WriteBuf procedure*,* *Writeln* procedure

Writeln procedure — System

Declaration

```
procedure Writeln([ var F: Text; ] P1 [, P2, ...,Pn ] );
```

The *Writeln* procedure is an extension to the *Write* procedure, as it is defined for text files.

After executing *Write*, *Writeln* writes an end-of-line marker (carriage-return/linefeed) to the file. *Writeln*(*F*) with no parameters writes an end-of-line marker to the file. (*Writeln* with no parameter list corresponds to *Writeln*(*Output*).)

The file must be open for output.

Example

```
   uses WinCrt;

   var
      s : string;
    begin
      Write('Enter a line of text: ');
      Readln(s);
      Writeln('You typed: ',s);
```

W

```
      Writeln('Hit <Enter> to exit');
      Readln;
    end;
```

See also
Write procedure

WriteInteger method

Applies to
TIniFile object

Declaration

```
procedure WriteInteger(const Section, Ident: string; Value: Longint);
```

The *WriteInteger* method writes an integer value in an .INI file.

The *Section* constant identifies the section of the .INI file where the value is written. For example, the WIN.INI for Windows contains a [Desktop] section.

The *Ident* constant is the name of the identifier for which you want to change the value. The *Value* parameter contains the new value.

Example
This example creates an .INI file for a game with two entries in the Options section, and one entry in the Configuration section.

Before you try this example, you must add *IniFiles* to the **uses** clause of your unit.

```
var
  GameIni: TIniFile;
begin
  GameIni := TIniFile.Create('FUNGAME.INI');
  GameIni.WriteBool('Options', 'Sound', True);
  GameIni.WriteInteger('Options', 'Level', 3);
  GameIni.WriteString('Configuration', 'Name', 'Teresa Ace');
  GameIni.Free;
end;
```

See also
ReadInteger method, *WriteBool* method, *WriteString* method

WriteString method

Applies to
TIniFile object

Declaration

```
procedure WriteString(const Section, Ident, Value: string);
```

The *WriteString* method writes a string in an .INI file.

The *Section* constant identifies the section of the .INI file where the string is written. For example, the WIN.INI for Windows contains a [Desktop] section.

The *Ident* constant is the name of the identifier for which you want to change the value. The *Value* constant holds the new string value.

Example

This example creates an .INI file for a game with two entries in the Options section, and one entry in the Configuration section.

Before you try this example, you must add *IniFiles* to the **uses** clause of your unit.

```
var
  GameIni: TIniFile;
begin
  GameIni := TIniFile.Create('FUNGAME.INI');
  GameIni.WriteBool('Options', 'Sound', True);
  GameIni.WriteInteger('Options', 'Level', 3);
  GameIni.WriteString('Configuration', 'Name', 'Teresa Ace');
  GameIni.Free;
end;
```

See also

ReadSection method, *ReadString* method, *WriteBool* method, *WriteInteger* method

Zoom property

Applies to

TOLEContainer component

Declaration

```
property Zoom : TZoomFactor
```

Run-time only. *Zoom* specifies how much to magnify or shrink the picture of an OLE object within an OLE container. *Zoom* defaults to *z100*. Setting *Zoom* to a different value causes the picture of the OLE object in the OLE container to be scaled accordingly. If you zoom in to make the picture of the OLE object larger than the OLE container, the extra portion of the picture will be visually clipped to the size of the OLE container. The OLE object itself won’t be affected, however. These are the possible values:

Value	Meaning
z025	Zoom to 25% of OLE object’s original size
z050	Zoom to 50% of OLE object’s original size

Value	Meaning
z100	Zoom to 100% of OLE object's original size
z150	Zoom to 150% of OLE object's original size
z200	Zoom to 200% of OLE object's original size

Example

The following code should be attached to the *OnClick* event handlers of the Zoom In and Zoom Out buttons. When the *ZoomInBtn* is clicked, the image in *OLEContainer1* is magnified. When the *ZoomOutBtn* is clicked, the image in *OLEContainer1* is reduced.

```
procedure TForm1.ZoomInBtnClick(Sender: TObject);
begin
  OLEContainer1.Zoom := Succ(OLEContainer1.Zoom)
end;

procedure TForm1.ZoomOutBtnClick(Sender: TObject);
begin
  OLEContainer1.Zoom := Pred(OLEContainer1.Zoom)
end;
```

Index

Visual Component Library Reference

Borland®
Delphi™

Borland International, Inc., 100 Borland Way
P.O. Box 660001, Scotts Valley, CA 95067-0001

Borland may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1995 Borland International. All rights reserved. All Borland product names are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

Printed in the U.S.A.

1E0R395
9596979899-987654321
W1

Contents